Normalizing LDDMM metrics with autodiff

Benjamin Charlier, **Jean Feydy**, Joan Alexis Glaunès, Alain Trouvé

June 8, 2018 – SIAM IS 2018, Bologna

École Normales Supérieures de Paris et Paris-Saclay

Overview

`conda install pytorch`

Facebook

- Seamless GPU support,
- Automatic differentiation.

Overview

`conda install pytorch` — Facebook

- Seamless GPU support,
- Automatic differentiation.

`pip install pykeops` — + B. Charlier, J. Glaunès

- Scale up to 3D models.

Overview

`conda install pytorch` — Facebook

- Seamless GPU support,
- Automatic differentiation.

`pip install pykeops` — + B. Charlier, J. Glaunès

- Scale up to 3D models.

Enjoy your maths! — + A. Trouvé

- Modelling prior $\perp$ Implementation.

Automatic differentiation?

- Gradients of your programs, for free!

Automatic differentiation?

- Gradients of your programs, for free!
- Gained traction thanks to Deep Learning:

Automatic differentiation?

- Gradients of your programs, for free!
- Gained traction thanks to Deep Learning:
    - Theano (2008–2017)

Automatic differentiation?

- Gradients of your programs, for free!
- Gained traction thanks to Deep Learning:
  - Theano (2008–2017)
  - TensorFlow (2015–)

Automatic differentiation?

- Gradients of your programs, for free!
- Gained traction thanks to Deep Learning:
  - Theano (2008–2017)
  - TensorFlow (2015–)
  - PyTorch (2017–)

Automatic differentiation?

- Gradients of your programs, for free!
- Gained traction thanks to Deep Learning:
  - Theano (2008–2017)
  - TensorFlow (2015–)
  - PyTorch (2017–)
- Used for shape analysis since 2017 – see [KAS17].

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

What is a gradient?

$$\text{Let} \qquad f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R} \qquad \text{be smooth,}$$

$$\text{So that} \quad \mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R} \quad \text{is linear:}$$

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star$

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star \simeq (\mathrm{d}\mathbf{f}(\mathbf{x}))^\mathsf{T}$

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star \simeq (\mathrm{d}\mathbf{f}(\mathbf{x}))^\mathsf{T}$

i.e. $\partial f(x) : \mathrm{d}y^\star \in \mathbb{R} \mapsto \mathrm{d}x^\star \in \mathbb{R}^n$.

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star \simeq (\mathrm{d}\mathbf{f}(\mathbf{x}))^\mathsf{T}$

i.e. $\partial f(x) : \mathrm{d}y^\star \in \mathbb{R} \mapsto \mathrm{d}x^\star \in \mathbb{R}^n$.

$$\partial f(x).\mathrm{d}y^\star = \begin{pmatrix} \partial_1 f(x) \\ \vdots \\ \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}y^\star \end{pmatrix} = \begin{pmatrix} \mathrm{d}x^\star \end{pmatrix} \quad \text{so that} \quad \nabla f(x) = \partial f(x).1$$

Autodiff is simple – no magic!

This definition lets us **compose gradients**:

Autodiff is simple – no magic!

This definition lets us **compose gradients**:

$$f = h \circ g$$

Autodiff is simple – no magic!

This definition lets us **compose gradients**:

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x)
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x))
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

Autodiff is simple – no magic!

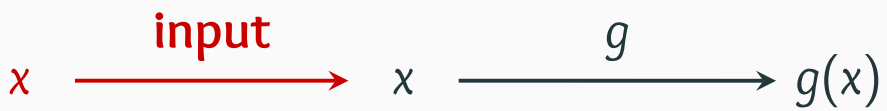
This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

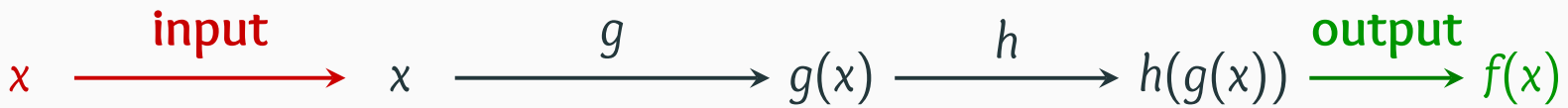
This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

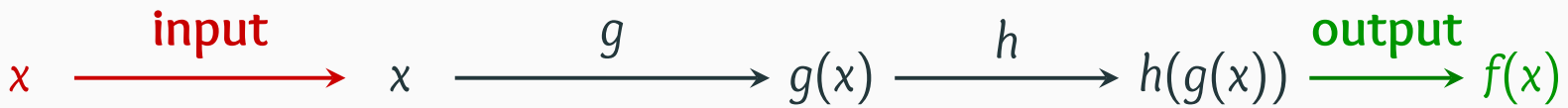
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

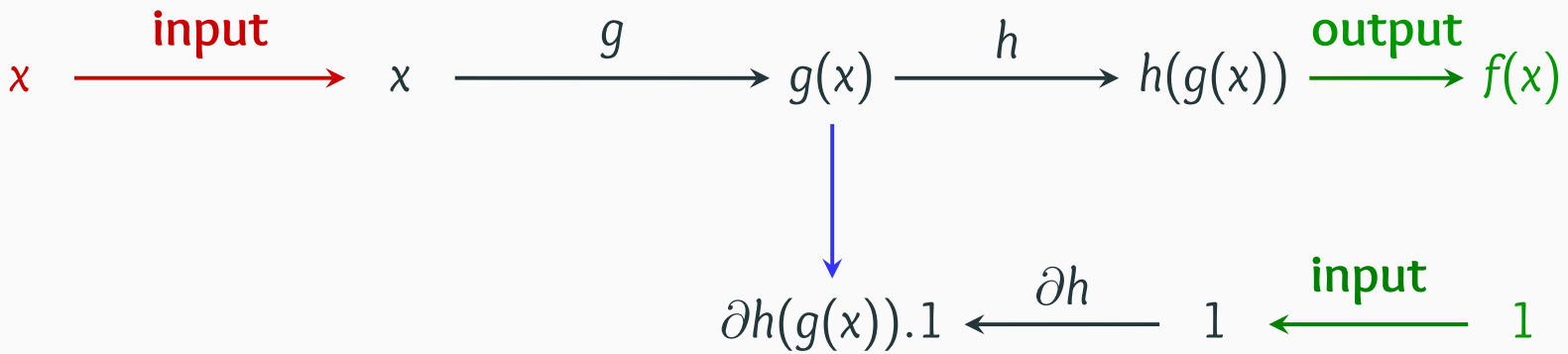
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

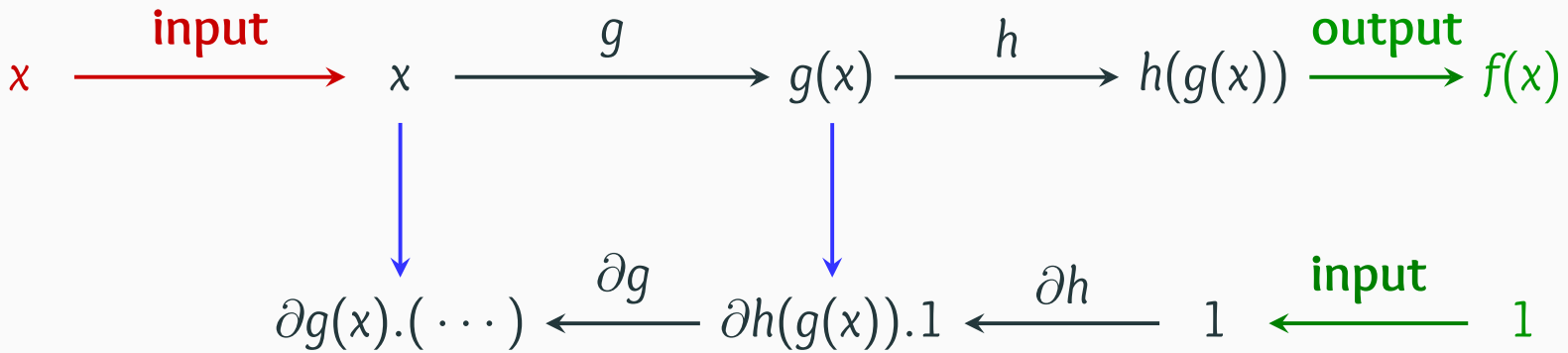
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

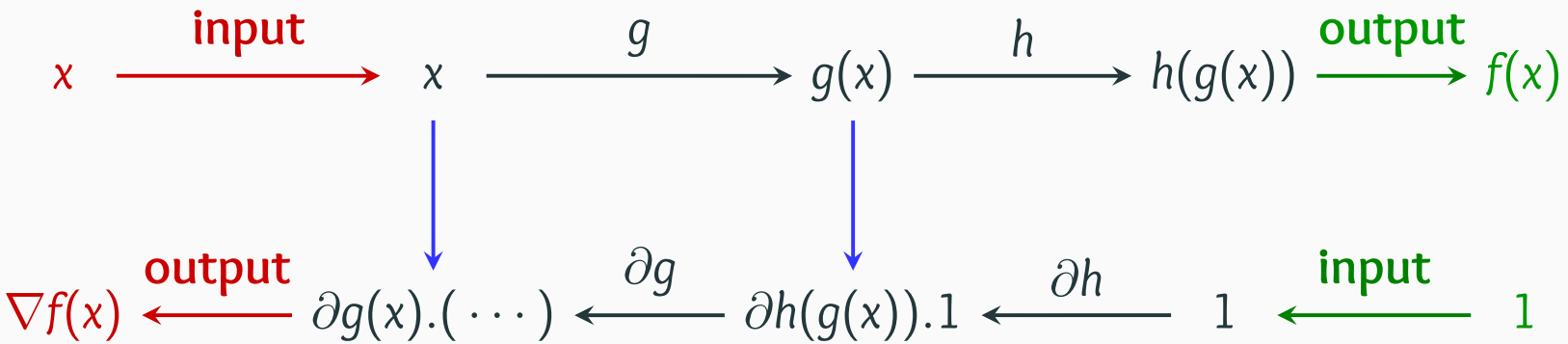
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is easy to use

PyTorch [PGC+17]:

- Straightforward replacement of Matlab/Numpy.

Autodiff is easy to use

PyTorch [PGC+17]:

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.

Autodiff is easy to use

PyTorch [PGC+17]:

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.
- Seamless **GPU** support.

Autodiff is easy to use

PyTorch [PGC+17]:

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.
- Seamless **GPU** support.

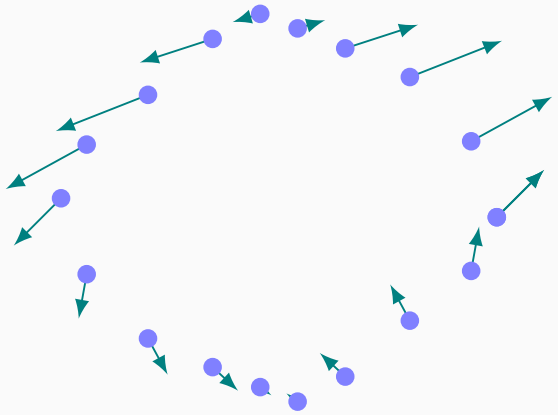
Autodiff is easy to use

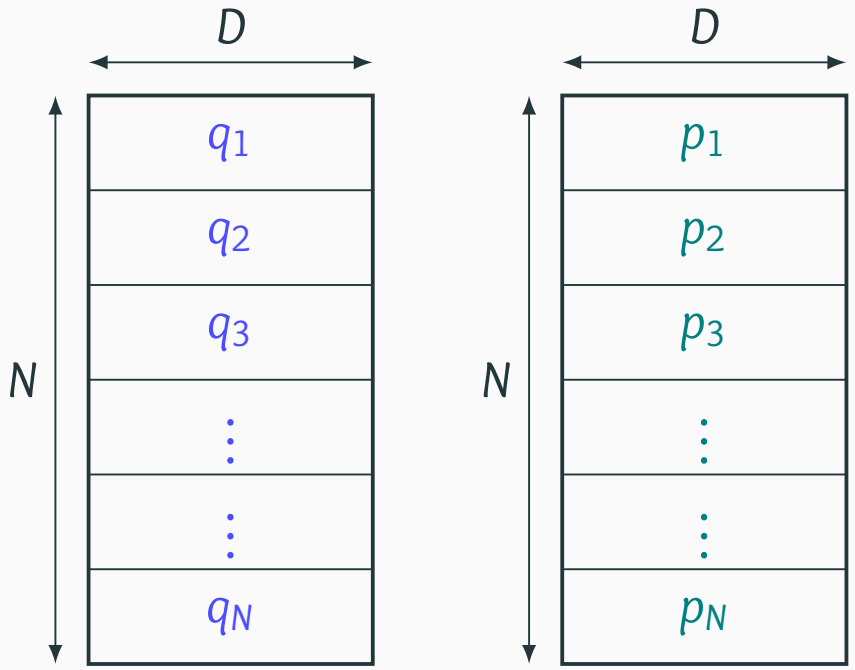
PyTorch [PGC+17]:

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.
- Seamless **GPU** support.

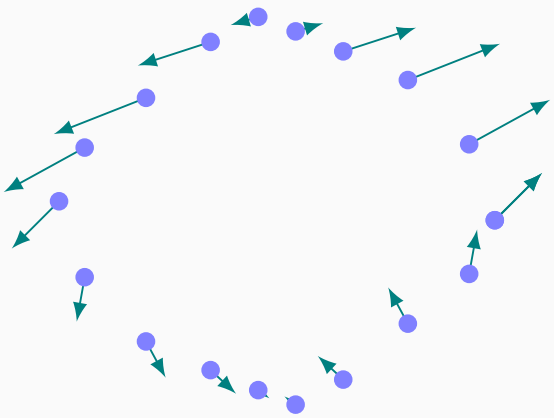
Let's see how it goes **in practice**!

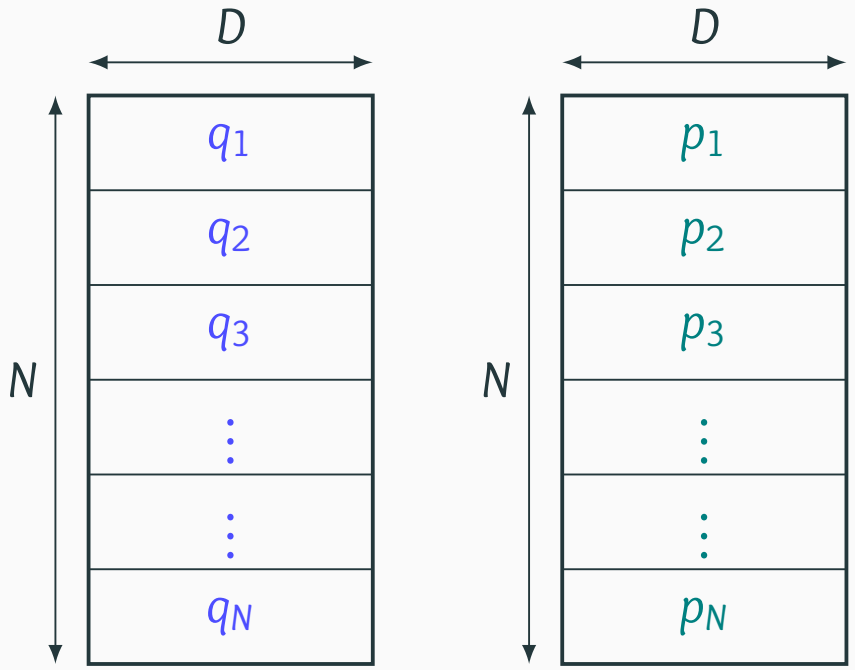
A typical formula: the kernel square norm





A typical formula: the kernel square norm

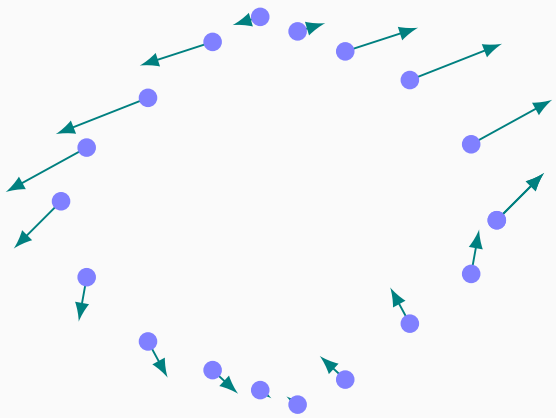


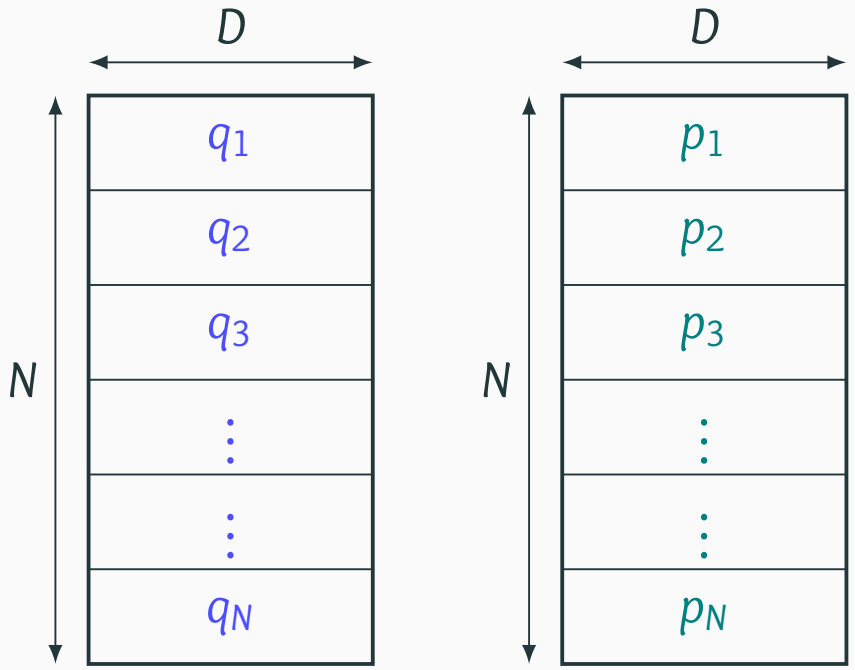


In shape analysis, algorithms often rely on the **kernel dot product:**

$$H(q,p) = \frac{1}{2}\sum_{i,j} \exp\left(-\tfrac{1}{\sigma^2}\|q_i - q_j\|^2\right) \langle p_i, p_j \rangle_2$$

A typical formula: the kernel square norm





In shape analysis, algorithms often rely on the **kernel dot product:**

$$H(q,p) = \frac{1}{2}\sum_{i,j} \exp\left(-\tfrac{1}{\sigma^2}\|q_i - q_j\|^2\right) \langle p_i, p_j \rangle_2$$

$$= \frac{1}{2}\sum_{i} \left\langle p_i, \sum_j k(q_i - q_j)\, p_j \right\rangle_2 = \frac{1}{2}\langle p, K_q p \rangle_2 \,.$$

PyTorch, in practice

```
import torch            # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dev      = torch.device("cuda" if use_gpu else "cpu")
```

PyTorch, in practice

```
import torch            # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dev      = torch.device("cuda" if use_gpu else "cpu")

# Create arbitrary arrays on the CPU or GPU:
N = 5000 ; D = 3
q = torch.randn( N, D, device=dev, requires_grad=True )
p = torch.randn( N, D, device=dev, requires_grad=True )
s = torch.tensor([.5], device=dev )
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 )  # |q_i-q_j|^2
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

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

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

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

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

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

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

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

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

H = 6029309.0

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

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

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

H = 6029309.0

```
# Automatic differentiation is straitghtforward:
[dq,dp] = grad( H, [q,p] )
```

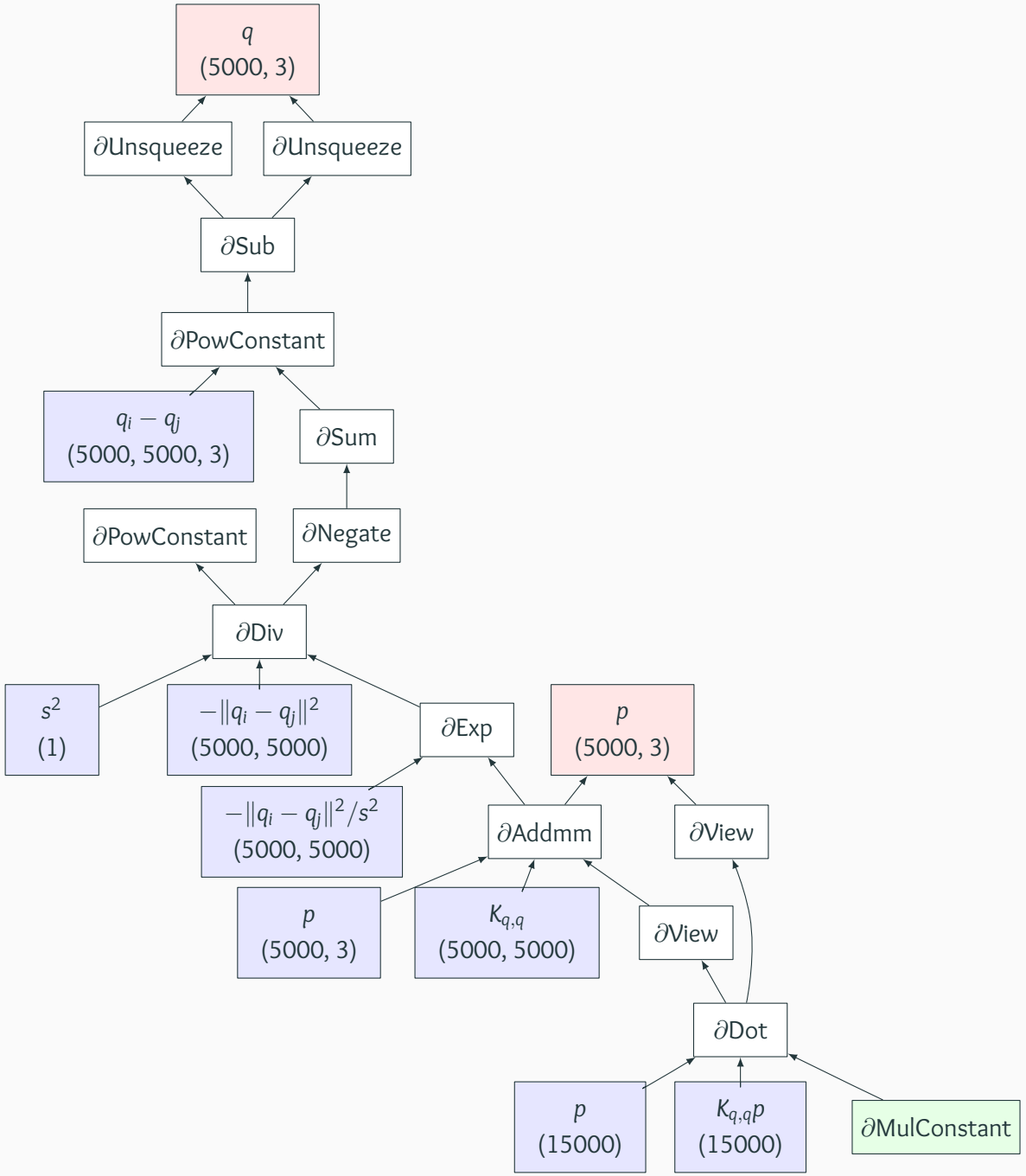
PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

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

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

H = 6029309.0

```
# Automatic differentiation is straitghtforward:
[dq,dp] = grad( H, [q,p] )
```

dq.shape = q.shape ; dp.shape = p.shape


q
(5000, 3)
∂Unsqueeze
∂Unsqueeze
∂Sub
∂PowConstant
$q_i - q_j$
(5000, 5000, 3)
∂Sum
∂PowConstant
∂Negate
∂Div
s^2
(1)
$-\|q_i - q_j\|^2$
(5000, 5000)
∂Exp
p
(5000, 3)
$-\|q_i - q_j\|^2/s^2$
(5000, 5000)
∂Addmm
∂View
p
(5000, 3)
$K_{q,q}$
(5000, 5000)
∂View
∂Dot
p
(15000)
$K_{q,q}p$
(15000)
∂MulConstant

PyTorch, in practice

```
import torch            # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dev      = torch.device("cuda" if use_gpu else "cpu")
```

PyTorch, in practice

```
import torch            # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dev      = torch.device("cuda" if use_gpu else "cpu")

# Create arbitrary arrays on the CPU or GPU:
N = 50000 ; D = 3
q = torch.randn( N, D, device=dev, requires_grad=True )
p = torch.randn( N, D, device=dev, requires_grad=True )
s = torch.tensor([.5], device=dev )
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 )  # |q_i-q_j|^2
```

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 )  # |q_i-q_j|^2
```

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

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 )  # |q_i-q_j|^2
```

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

Large N-by-N matrices do **not** fit into RAM/GPU memories.

PyTorch, in practice

```
# Re-indexing:
q_i  =  q[:,None,:]  # shape (N,D) -> (N,1,D)
q_j  =  q[None,:,:]  # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 )  # |q_i-q_j|^2
```

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

Large N-by-N matrices do **not** fit into RAM/GPU memories.

We need to compute + sum the kernel values

on-the-fly.

KeOps:
Online Map-Reduce Operators,
with autodiff,
without memory overflows.

KeOps:
Online Map-Reduce Operators,
with autodiff,
without memory overflows.

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

(Thank you Benjamin!)

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots\right)\right],$$

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: p^1, p^2, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots\right)\right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$
- Vector **y-variables**, indexed by j : $y_j^1, y_j^2, \ldots$

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$
- Vector **y-variables**, indexed by j : $y_j^1, y_j^2, \ldots$

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$
- Vector **y-variables**, indexed by j : $y_j^1, y_j^2, \ldots$

With **KeOps** you will get:

- a **Linear** memory footprint.

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M}\left[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \right],$$

with :

- "**Reduction**" : Sum, Max, Min, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$
- Vector **y-variables**, indexed by j : $y_j^1, y_j^2, \ldots$

With **KeOps** you will get:

- a **Linear** memory footprint.
- High order **derivatives** – thank you Joan!

KeOps' low-level interface: `generic_sum`

With x_i, y_j points in $\mathbb{R}^3$ and b_j a 2D-signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

KeOps' low-level interface: `generic_sum`

With x_i, y_j points in $\mathbb{R}^3$ and b_j a 2D-signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

```
from pykeops.torch import generic_sum

gaussian_conv = generic_sum(
  "Exp(-G*SqDist(X,Y)) * B", # Custom formula
  "A = Vx(2)", # Output,  2D, indexed by i
  "G = Pm(1)", # 1st arg, 1D, parameter
  "X = Vx(3)", # 2nd arg, 3D, indexed by i
  "Y = Vy(3)", # 3rd arg, 3D, indexed by j
  "B = Vy(1)") # 4th arg, 2D, indexed by j
```

KeOps' low-level interface: `generic_sum`

With x_i, y_j points in $\mathbb{R}^3$ and b_j a 2D-signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

```
from pykeops.torch import generic_sum

gaussian_conv = generic_sum(
  "Exp(-G*SqDist(X,Y)) * B", # Custom formula
  "A = Vx(2)", # Output,  2D, indexed by i
  "G = Pm(1)", # 1st arg, 1D, parameter
  "X = Vx(3)", # 2nd arg, 3D, indexed by i
  "Y = Vy(3)", # 3rd arg, 3D, indexed by j
  "B = Vy(1)") # 4th arg, 2D, indexed by j

# Simply apply your routine to CPU/GPU torch tensors!
a = gaussian_conv( 1/sigma**2, x, y, b )
```

KeOps' high-level interface: `kernel_product`

With x_i, y_j points in $\mathbb{R}^D$, b_j a vector-valued signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

KeOps' high-level interface: `kernel_product`

With x_i, y_j points in $\mathbb{R}^D$, b_j a vector-valued signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

```
from pykeops.torch import Kernel, kernel_product

k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma":           1/sigma**2     }
```

KeOps' high-level interface: `kernel_product`

With x_i, y_j points in $\mathbb{R}^D$, b_j a vector-valued signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

```
from pykeops.torch import Kernel, kernel_product

k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma":           1/sigma**2     }

a = kernel_product( k, x, y, b )
```

KeOps' high-level interface: `kernel_product`

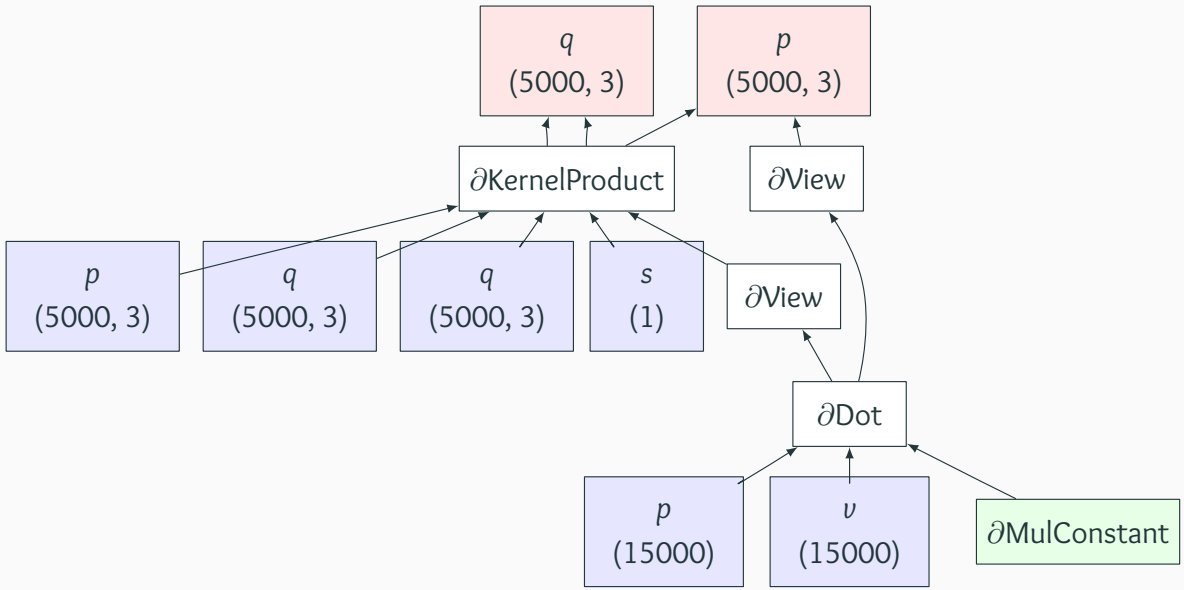
With x_i, y_j points in $\mathbb{R}^D$, b_j a vector-valued signal, u_i, v_j directions in $\mathbb{R}^D$:

$$a_i \;=\; \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot \langle u_i, v_j \rangle^2 \cdot b_j$$

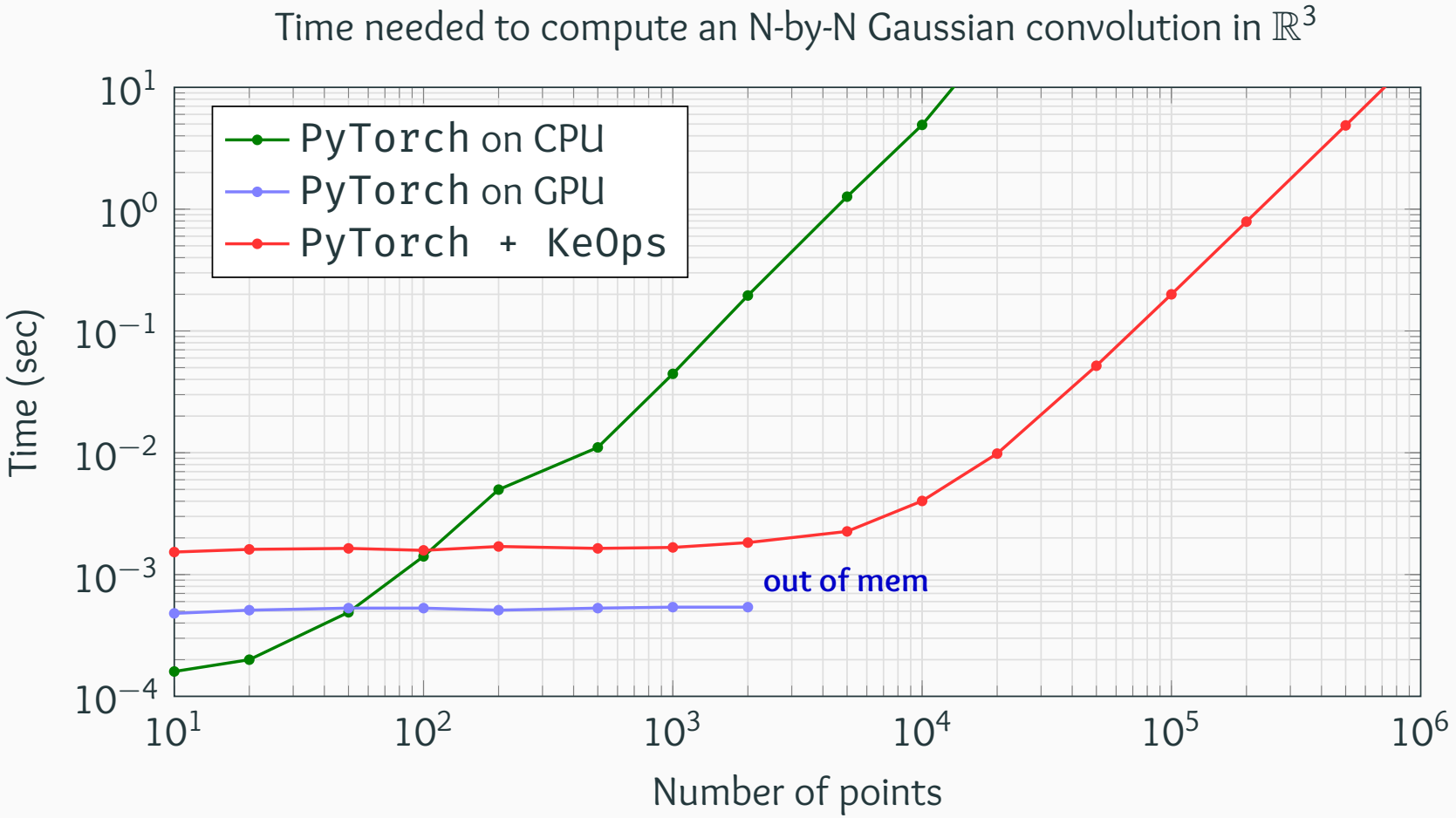
```
from pykeops.torch import Kernel, kernel_product

k = { "id"    : Kernel("gaussian(x,y) * linear(u,v)**2"),
      "gamma":          (  1/sigma**2  ,     None      ) }

a = kernel_product( k, (x,u), (y,v), b )
```

Use KeOps as any other PyTorch module

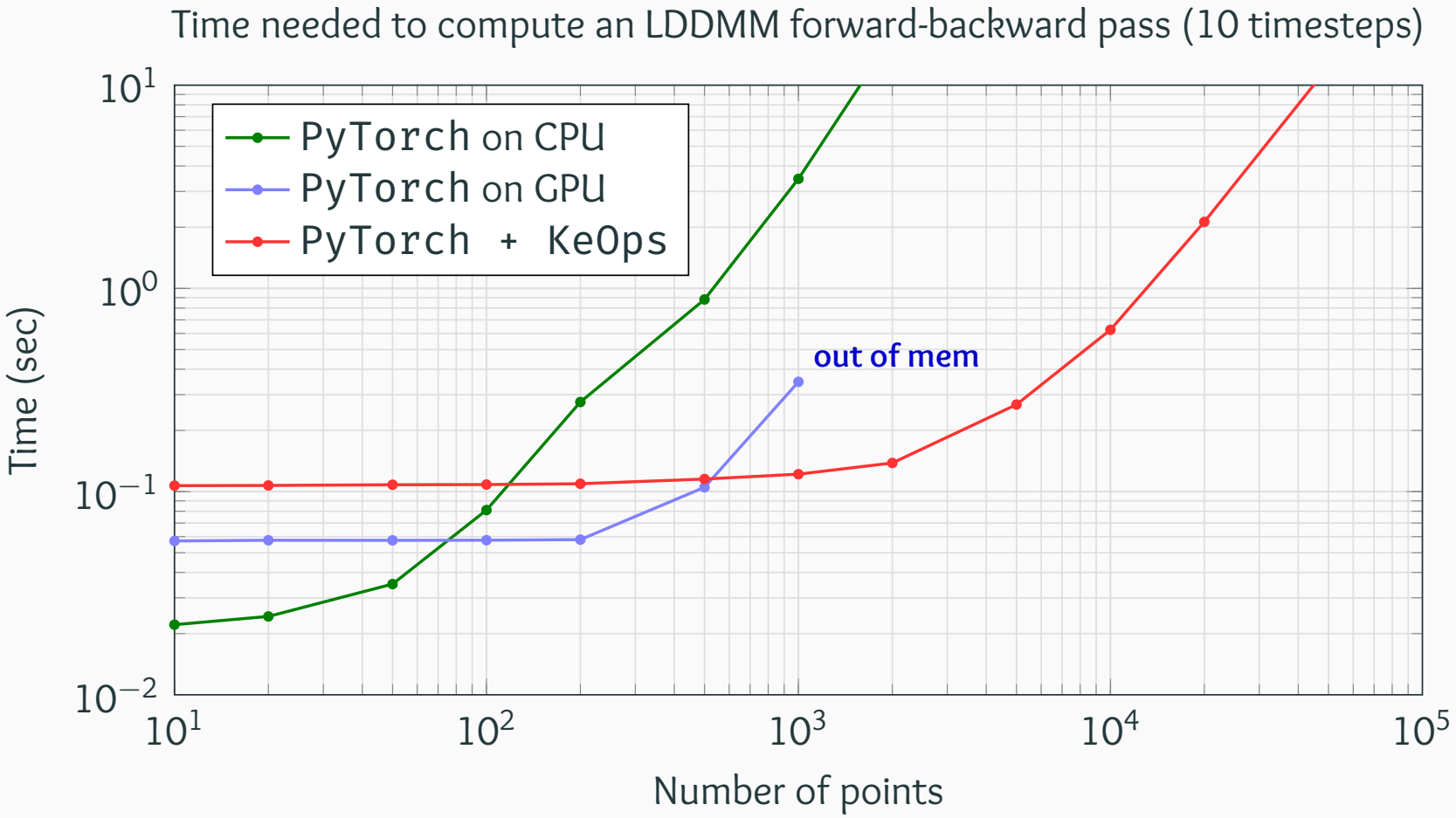
```
def H(q,p) :
  # Compute the kernel convolution
  v = kernel_product( k, q, q, p)
  # Then, output the kernel norm H(q,p): .5*<p,v>
  return .5 * torch.dot( p.view(-1), v.view(-1) )
```



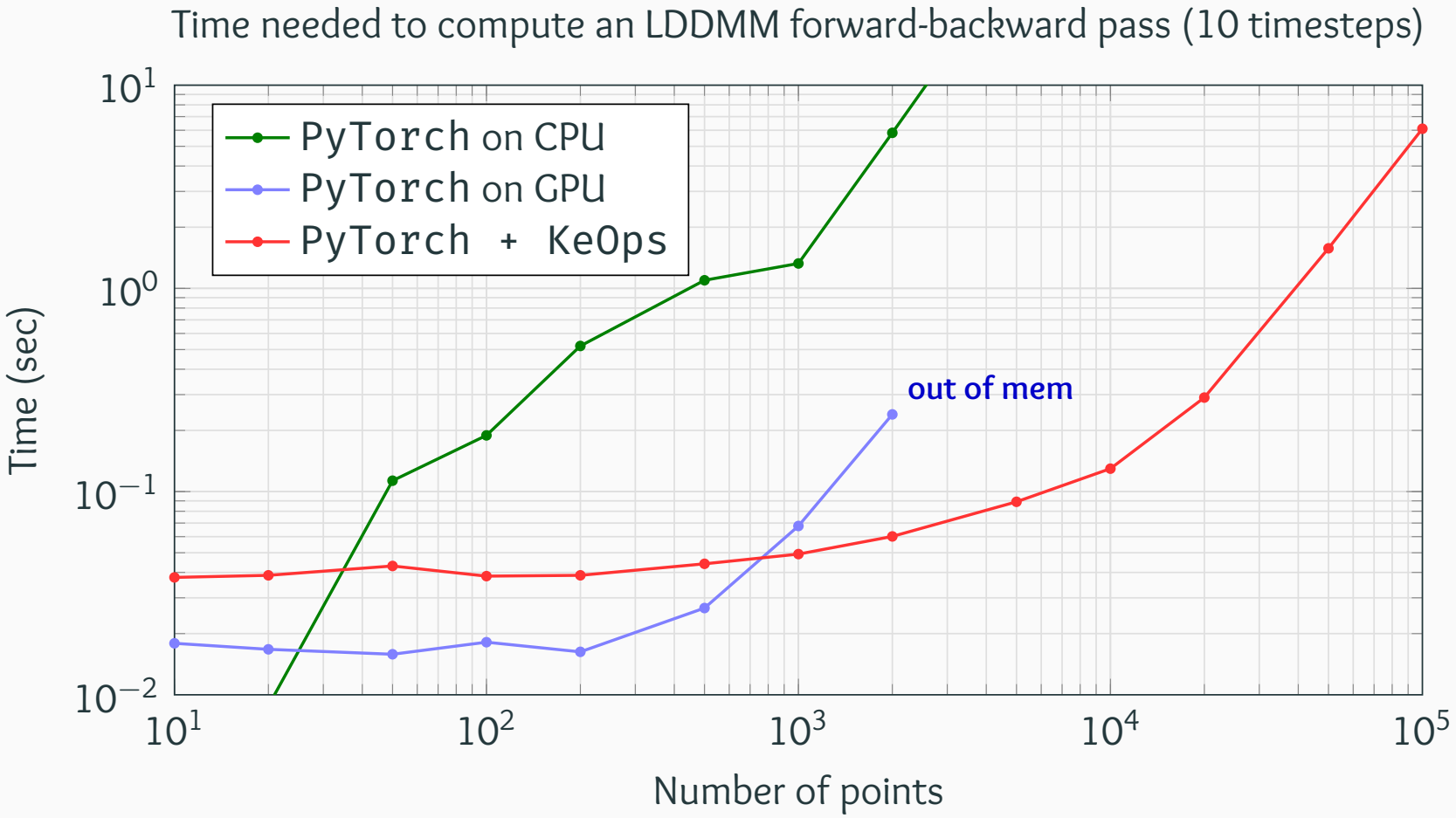
It just works : on my laptop (GTX 960M, March 2015)



It just works : on my laptop (GTX 960M, March 2015)

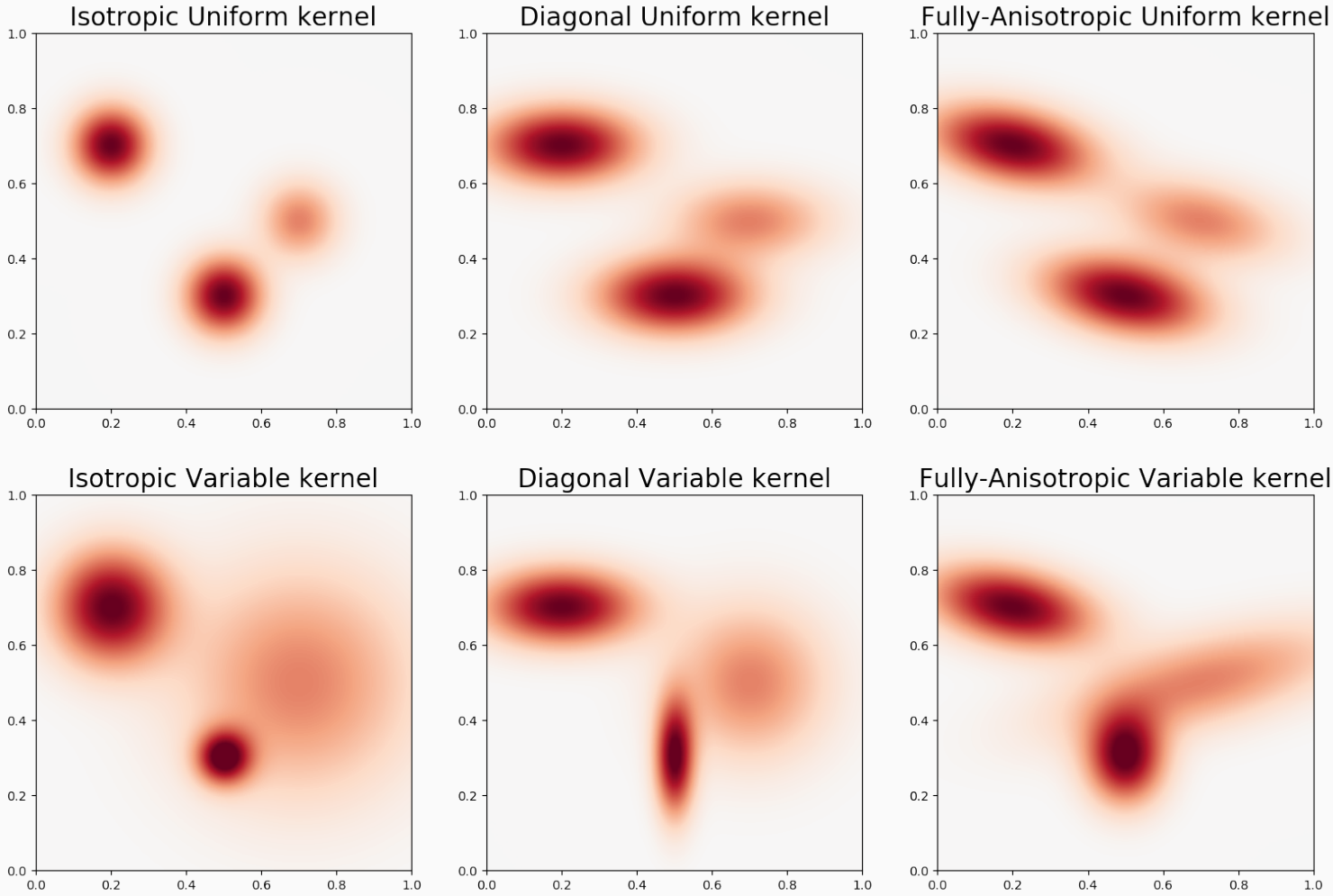


It just works : on a good GPU (Tesla P100, April 2016)



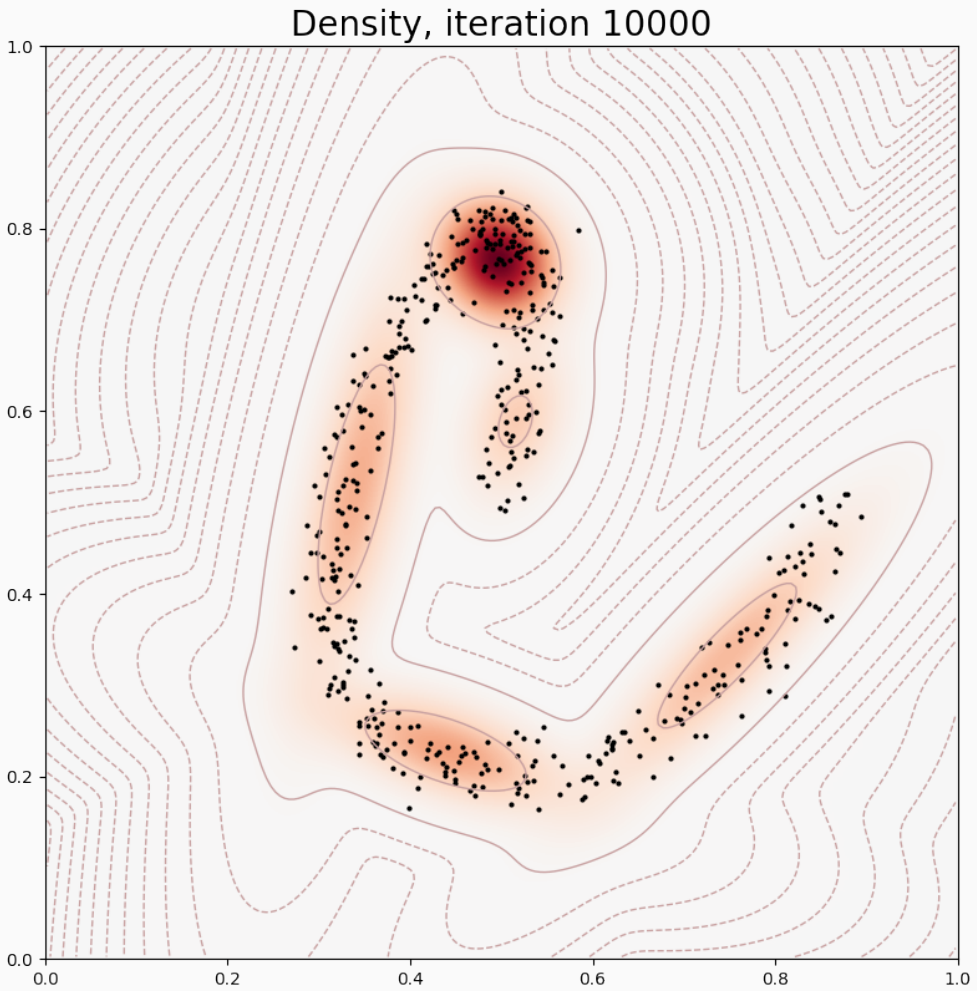
Examples provided in the documentation

Using **anisotropic** kernels:



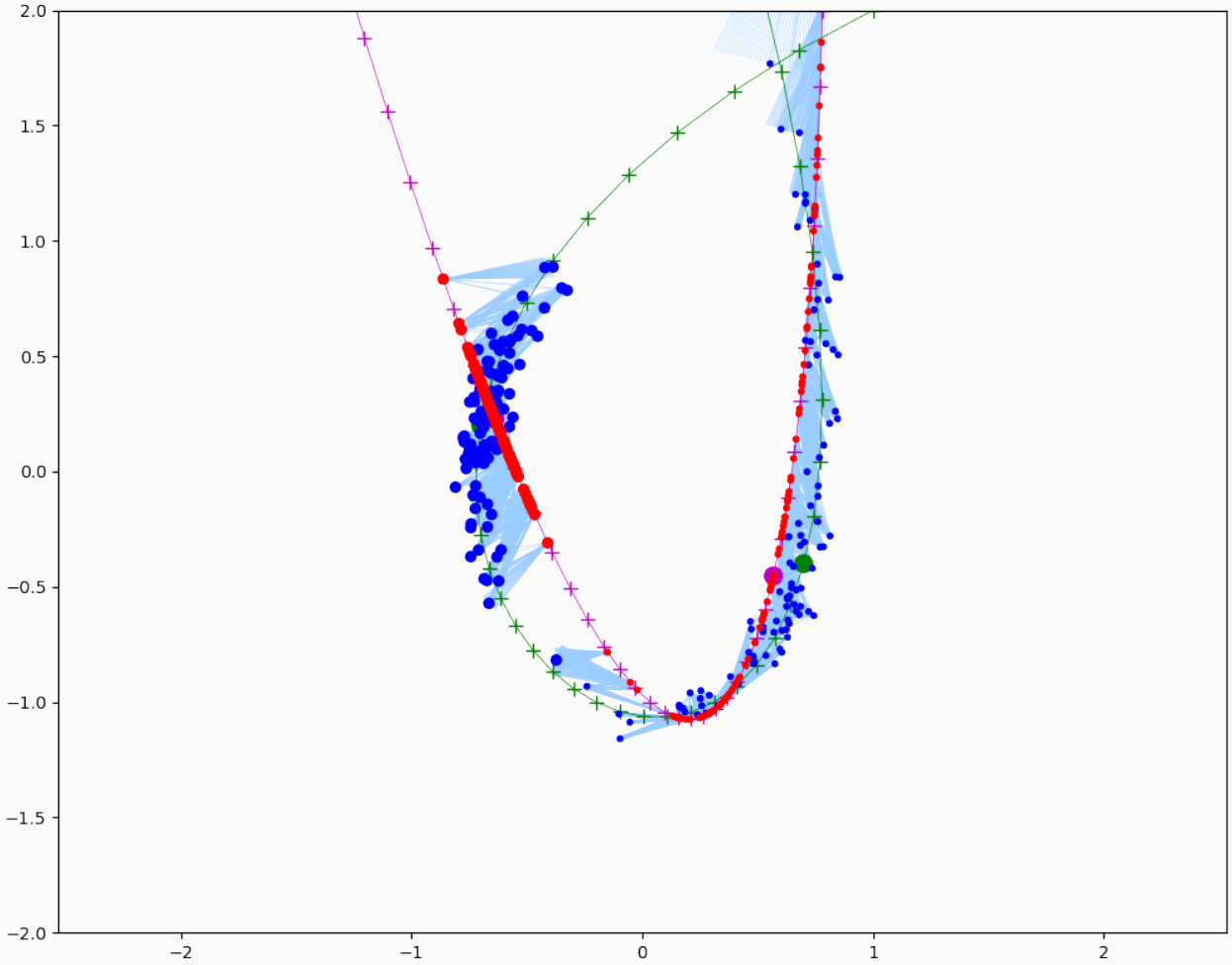
Examples provided in the documentation

Fitting a **Gaussian mixture** model:



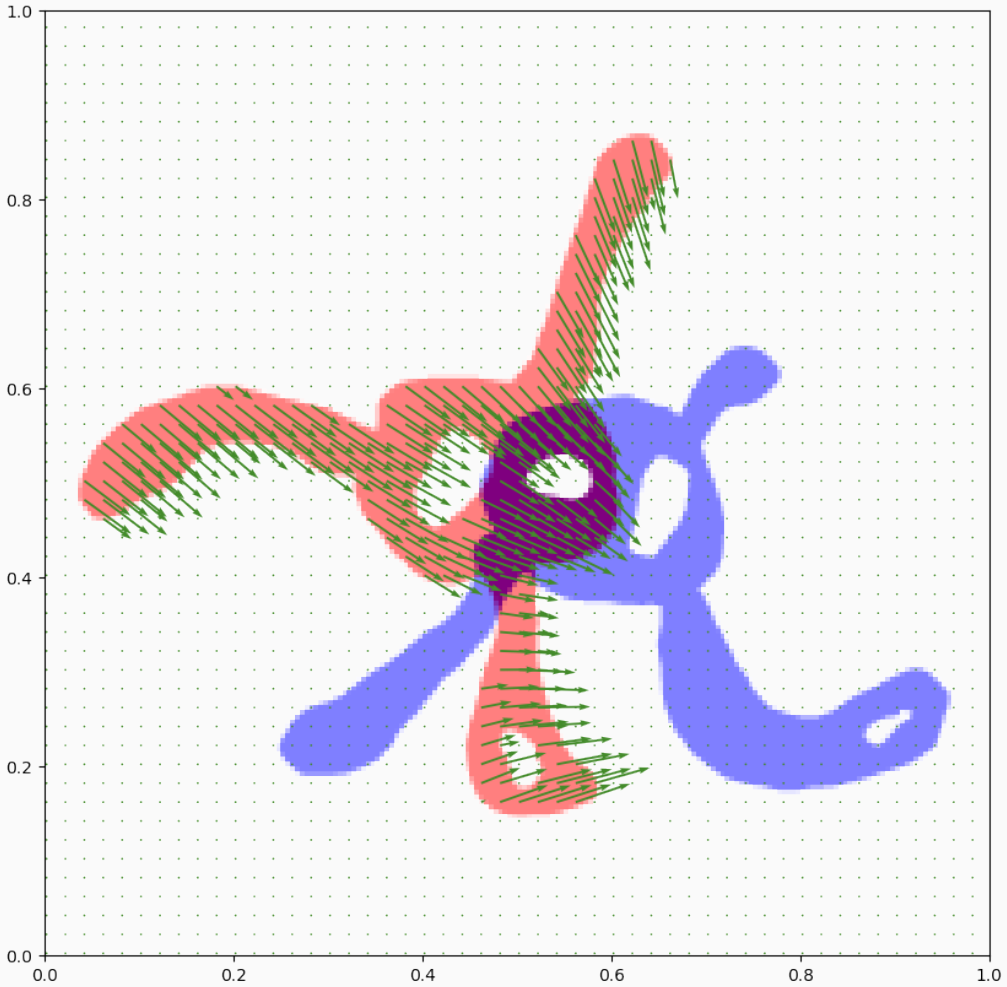
Examples provided in the documentation

Fitting an arbitrary **generative** model:



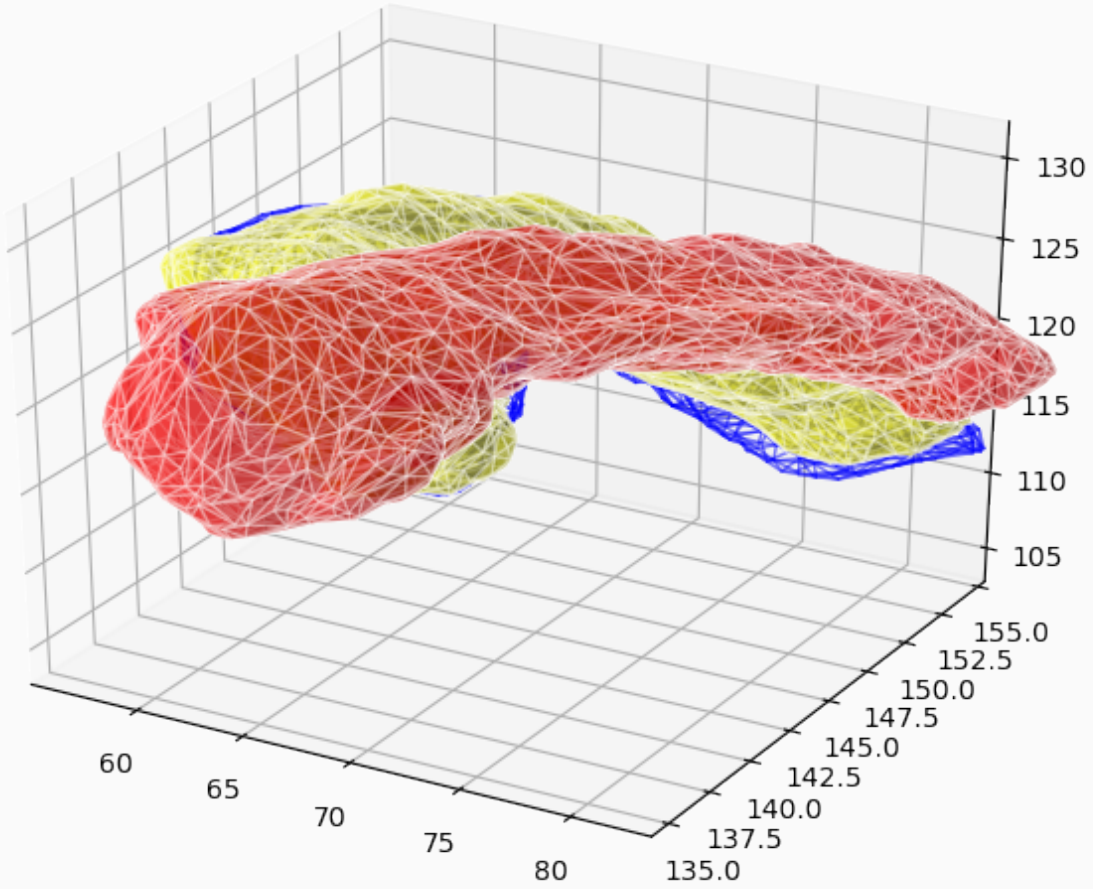
Examples provided in the documentation

Computing an **Optimal Transport** plan:



Examples provided in the documentation

Surface registration with **LDDMM:**

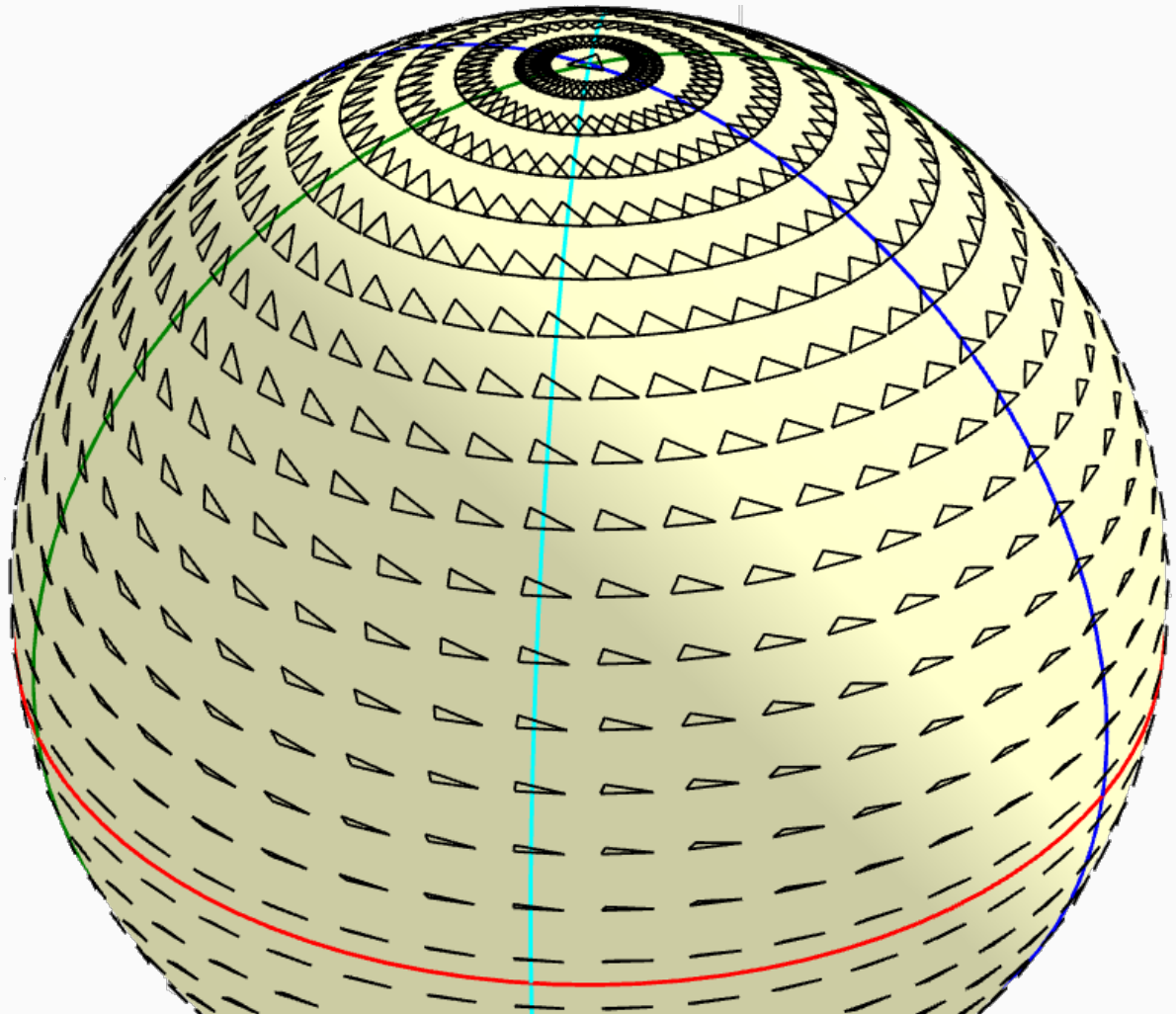


PyTorch
\+ KeOps

The right tool for **shape analysis**

Why do we study shape spaces?

In order to help statisticians, we strive to build **path distances** between shapes (say, point clouds).



Geodesics on a Riemannian manifold

We'd like to define

$$\mathrm{d}(a,b)^2 \;=\; \min_{\substack{q_0=a,\\ q_1=b}} \int_0^1 \underbrace{\left\langle \dot{q}_t \,,\, g_{q_t}\, \dot{q}_t \right\rangle}_{\text{local metric}} \mathrm{d}t$$

Geodesics on a Riemannian manifold

We'd like to define

$$\mathrm{d}(a,b)^2 \;=\; \min_{\substack{q_0=a,\\ q_1=b}} \int_0^1 \underbrace{\left\langle \dot{q}_t \,,\, g_{q_t}\, \dot{q}_t \right\rangle}_{\text{local metric}} \mathrm{d}t$$

Calculus of variations shows that q is a **critical point** of this energy

$$\iff \quad \tfrac{\mathrm{d}}{\mathrm{d}t}\left(g_{q_t}\dot{q}_t\right) \;=\; \tfrac{1}{2}\partial_q \left\langle\, v, g_q\, v \,\right\rangle (q_t, \dot{q}_t).$$

Geodesics on a Riemannian manifold

We'd like to define

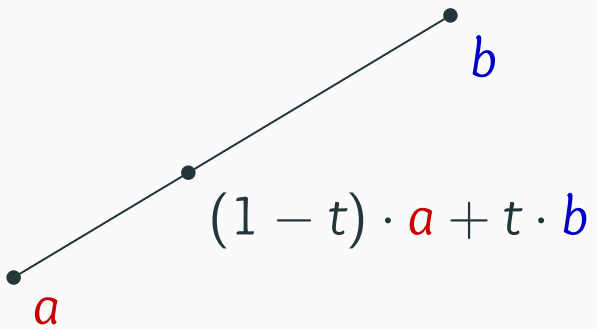
$$\mathrm{d}(a,b)^2 \;=\; \min_{\substack{q_0=a,\\ q_1=b}} \int_0^1 \underbrace{\left\langle \dot{q}_t \,,\, g_{q_t}\, \dot{q}_t \right\rangle}_{\text{local metric}} \mathrm{d}t$$

Calculus of variations shows that q is a **critical point** of this energy

$$\iff \quad \tfrac{\mathrm{d}}{\mathrm{d}t}\left(g_{q_t}\dot{q}_t\right) \;=\; \tfrac{1}{2}\partial_q \left\langle\, v, g_q\, v \,\right\rangle (q_t, \dot{q}_t).$$

If g_q is constant, we retrieve

$$\ddot{q}_t \;=\; 0.$$



Geodesics on a Riemannian manifold

We'd like to define

$$\mathrm{d}(a,b)^2 \;=\; \min_{\substack{q_0=a,\\ q_1=b}} \int_0^1 \underbrace{\left\langle \dot{q}_t \,,\, g_{q_t}\, \dot{q}_t \right\rangle}_{\text{local metric}} \mathrm{d}t$$

Calculus of variations shows that q is a **critical point** of this energy

$$\iff \quad \tfrac{\mathrm{d}}{\mathrm{d}t}\left(g_{q_t}\dot{q}_t\right) \;=\; \tfrac{1}{2}\partial_q \left\langle\, v, g_q\, v \,\right\rangle (q_t, \dot{q}_t).$$

Geodesics on a Riemannian manifold

We'd like to define

$$\mathrm{d}(a,b)^2 \;=\; \min_{\substack{q_0=a,\\ q_1=b}} \int_0^1 \underbrace{\left\langle \dot{q}_t \,,\, g_{q_t}\, \dot{q}_t \right\rangle}_{\text{local metric}} \mathrm{d}t$$

Calculus of variations shows that q is a **critical point** of this energy

$$\iff \quad \tfrac{\mathrm{d}}{\mathrm{d}t}\left(g_{q_t}\dot{q}_t\right) \;=\; \tfrac{1}{2}\partial_q \left\langle\, v, g_q\, v \,\right\rangle (q_t, \dot{q}_t).$$

In the general case, we use Hamilton's **change of variables**

$$p_t = g_{q_t}\dot{q}_t \qquad \text{i.e.} \qquad \dot{q}_t = K_{q_t} p_t \;\text{ with }\; K_q \;=\; g_q^{-1}.$$

Geodesics on a Riemannian manifold

We'd like to define

$$\mathrm{d}(a,b)^2 \;=\; \min_{\substack{q_0=a,\\ q_1=b}} \int_0^1 \underbrace{\big\langle\, \dot q_t \,,\, g_{q_t}\, \dot q_t \big\rangle}_{\text{local metric}} \,\mathrm{d}t$$

Calculus of variations shows that q is a **critical point** of this energy

$$\iff \quad \tfrac{\mathrm{d}}{\mathrm{d}t}\big(g_{q_t}\dot q_t\big) \;=\; \tfrac{1}{2}\partial_q \big\langle\, v, g_q\, v \,\big\rangle\,(q_t, \dot q_t).$$

In the general case, we use Hamilton's **change of variables**

$$p_t = g_{q_t}\dot q_t \qquad \text{i.e.} \qquad \dot q_t = K_{q_t} p_t \;\text{ with }\; K_q \;=\; g_q^{-1}.$$

The **geodesic equation** becomes

$$\dot p_t \;=\; -\tfrac{1}{2}\partial_q \big\langle\, p, K_q\, p \,\big\rangle (q_t, p_t).$$

Riemannian model $\Leftrightarrow$ Shooting routine

In the phase space (q_t, p_t), geodesics flow along the symplectic gradient of the **Hamiltonian** (aka. kinetic energy)

$$H(q,p) \;=\; \tfrac{1}{2}\langle p, K_q p\rangle \quad \text{i.e.} \quad \begin{cases} \dot{q}_t &= +\frac{\partial H}{\partial p}(q_t, p_t) \\ \dot{p}_t &= -\frac{\partial H}{\partial q}(q_t, p_t) \end{cases}$$

Riemannian model $\Leftrightarrow$ Shooting routine

In the phase space (q_t, p_t), geodesics flow along the symplectic gradient of the **Hamiltonian** (aka. kinetic energy)

$$H(q,p) = \tfrac{1}{2}\langle p, K_q p\rangle \quad \text{i.e.} \quad \begin{cases} \dot{q}_t &= +\frac{\partial H}{\partial p}(q_t, p_t) \\ \dot{p}_t &= -\frac{\partial H}{\partial q}(q_t, p_t) \end{cases}$$

```
def H(q,p) : # Encodes the geometry of our model
  return ⟨ p, K(q) p ⟩
```

Riemannian model $\Leftrightarrow$ Shooting routine

In the phase space (q_t, p_t), geodesics flow along the symplectic gradient of the **Hamiltonian** (aka. kinetic energy)

$$H(q,p) = \tfrac{1}{2}\langle p, K_q p \rangle \quad \text{i.e.} \quad \begin{cases} \dot{q}_t &= +\frac{\partial H}{\partial p}(q_t, p_t) \\ \dot{p}_t &= -\frac{\partial H}{\partial q}(q_t, p_t) \end{cases}$$

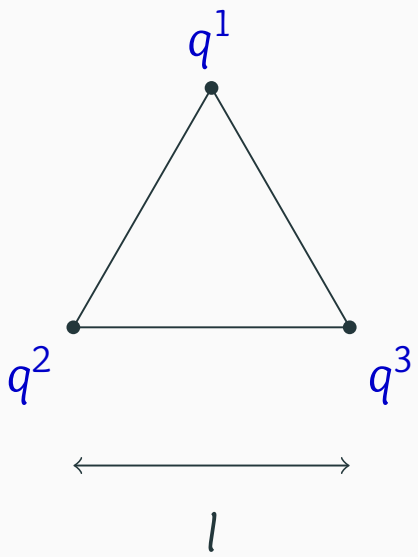
```
def H(q,p) : # Encodes the geometry of our model
  return ⟨ p, K(q) p ⟩

def Shoot(q0,p0) : # Simple ODE integrator
  q,p = q0,p0
  for t in range(10) :
    [dq,dp] = grad( H(q,p), [q,p], create_graph=True)
    q,p  =  ( q + .1 * dp ,
              p - .1 * dq )
  return q,p # = q1,p1
```

Understanding kernel cometrics

$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$

Understanding kernel cometrics

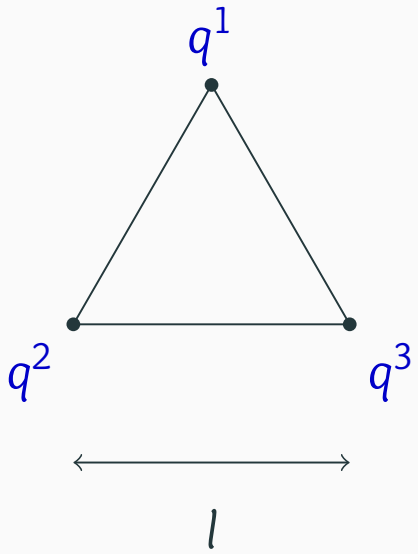
$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$



Understanding kernel cometrics

$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$

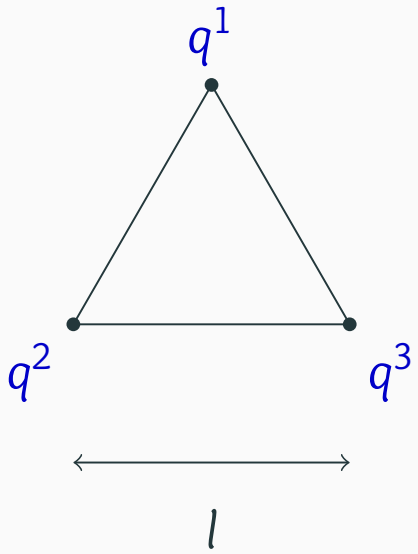
$$k(0) = 1, \quad k(l) = c \in (0,1)$$



Understanding kernel cometrics

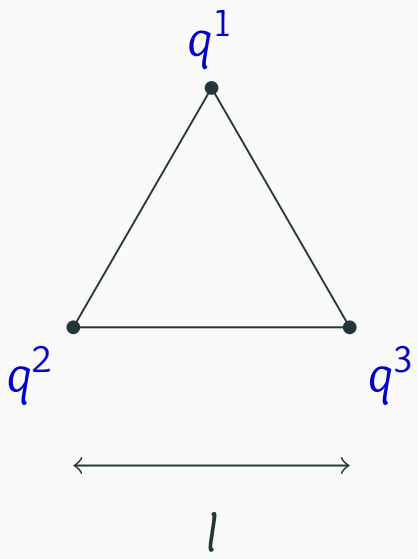
$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$

$$k(0) = 1, \quad k(l) = c \in (0,1)$$



Understanding kernel cometrics

$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$

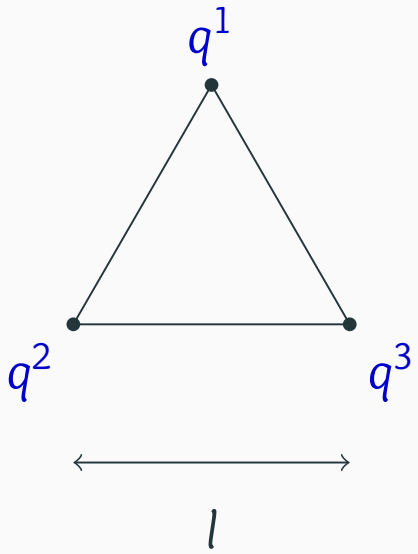


$$k(0) = 1, \quad k(l) = c \in (0,1)$$

$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

Understanding kernel cometrics

$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$



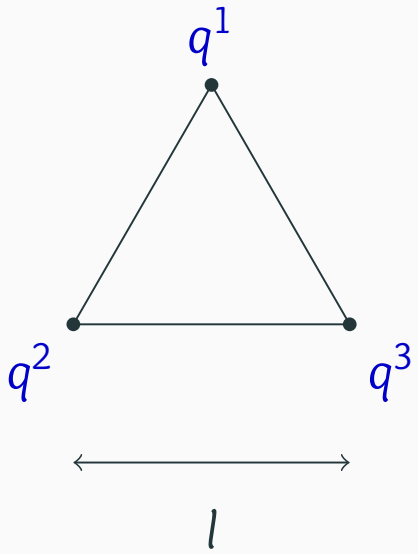
$$k(0) = 1, \quad k(l) = c \in (0,1)$$

$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

$$= \quad 1 + 2\cdot c \ \text{ on } (1) \qquad 1 - c \ \text{ on } (1)^{\perp}$$

Understanding kernel cometrics

$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$



$$k(0) = 1, \quad k(l) = c \in (0,1)$$

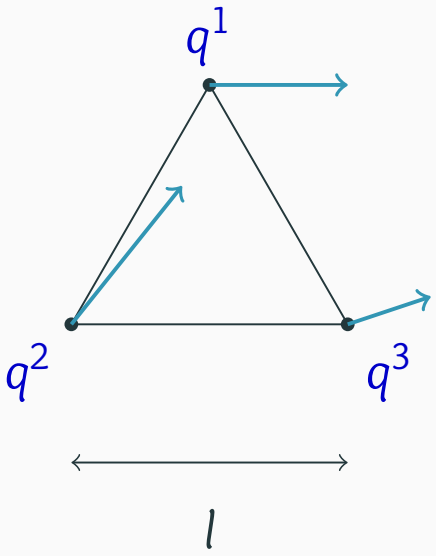
$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

$$= \quad 1 + 2\cdot c \ \text{ on } (1) \qquad 1 - c \ \text{ on } (1)^{\perp}$$

$$g_q = \quad \tfrac{1}{1+2\cdot c} \ \text{ on } (1) \qquad \tfrac{1}{1-c} \ \text{ on } (1)^{\perp}$$

Understanding kernel cometrics

$(K_q)_{i,j} = k(q_i - q_j)$, so that $H(q,p) = \frac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j\right\rangle_2$.



$$k(0) = 1, \quad k(l) = c \in (0,1)$$

$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

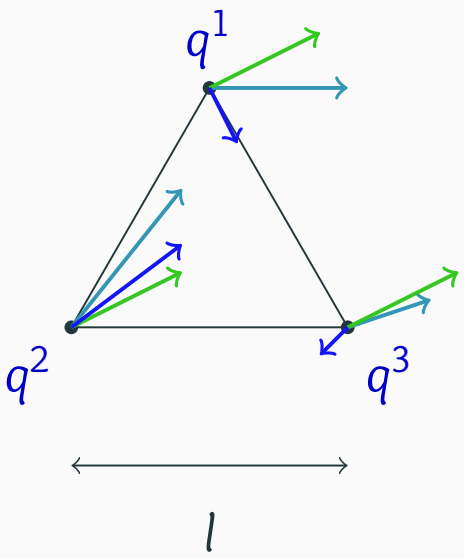
$$= \quad 1 + 2\cdot c \ \text{ on } (1) \qquad 1 - c \ \text{ on } (1)^\perp$$

$$g_q = \quad \tfrac{1}{1+2\cdot c} \ \text{ on } (1) \qquad \tfrac{1}{1-c} \ \text{ on } (1)^\perp$$

$$\langle v, g_q v\rangle =$$

Understanding kernel cometrics

$(K_q)_{i,j} = k(q_i - q_j)$, so that $H(q,p) = \frac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j\right\rangle_2$.



$$k(0) = 1, \quad k(l) = c \in (0,1)$$

$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

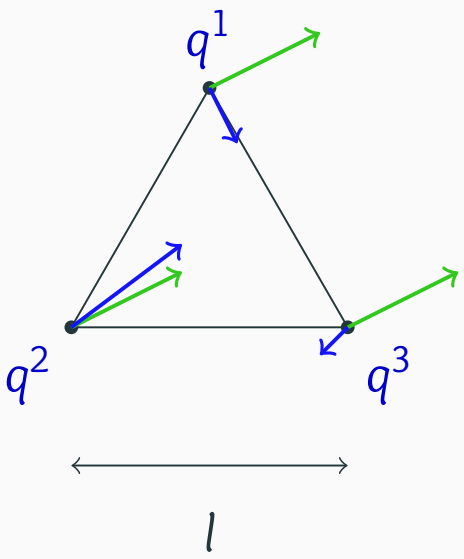
$$= \quad 1 + 2\cdot c \ \text{ on } (1) \qquad 1 - c \ \text{ on } (1)^{\perp}$$

$$g_q = \quad \tfrac{1}{1+2\cdot c} \ \text{ on } (1) \qquad \tfrac{1}{1-c} \ \text{ on } (1)^{\perp}$$

$v = v_{\text{mean}} + v_{\text{var}} \qquad \langle v, g_q v\rangle =$

Understanding kernel cometrics

$(K_q)_{i,j} = k(q_i - q_j)$, so that $H(q,p) = \frac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j\right\rangle_2$.



$$k(0) = 1, \quad k(l) = c \in (0,1)$$

$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

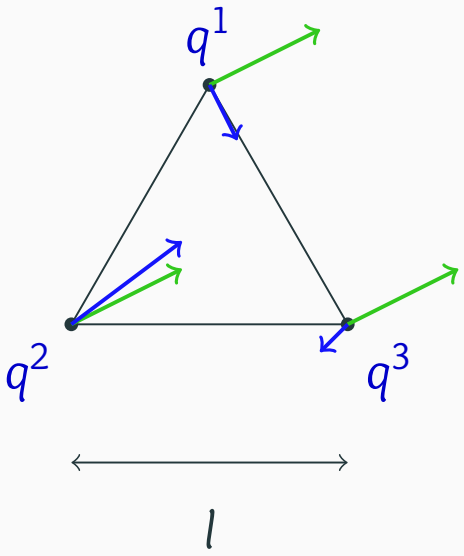
$$= \quad 1 + 2\cdot c \ \text{ on } (1) \qquad 1 - c \ \text{ on } (1)^{\perp}$$

$$g_q = \quad \tfrac{1}{1+2\cdot c} \ \text{ on } (1) \qquad \tfrac{1}{1-c} \ \text{ on } (1)^{\perp}$$

$v = v_{\text{mean}} + v_{\text{var}} \qquad \langle v, g_q v\rangle =$

Understanding kernel cometrics

$$(K_q)_{i,j} = k(q_i - q_j), \quad \text{so that } H(q,p) = \tfrac{1}{2}\sum_{i,j} k(q_i - q_j)\left\langle p_i, p_j \right\rangle_2.$$



$$k(0) = 1, \quad k(l) = c \in (0,1)$$

$$K_q = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

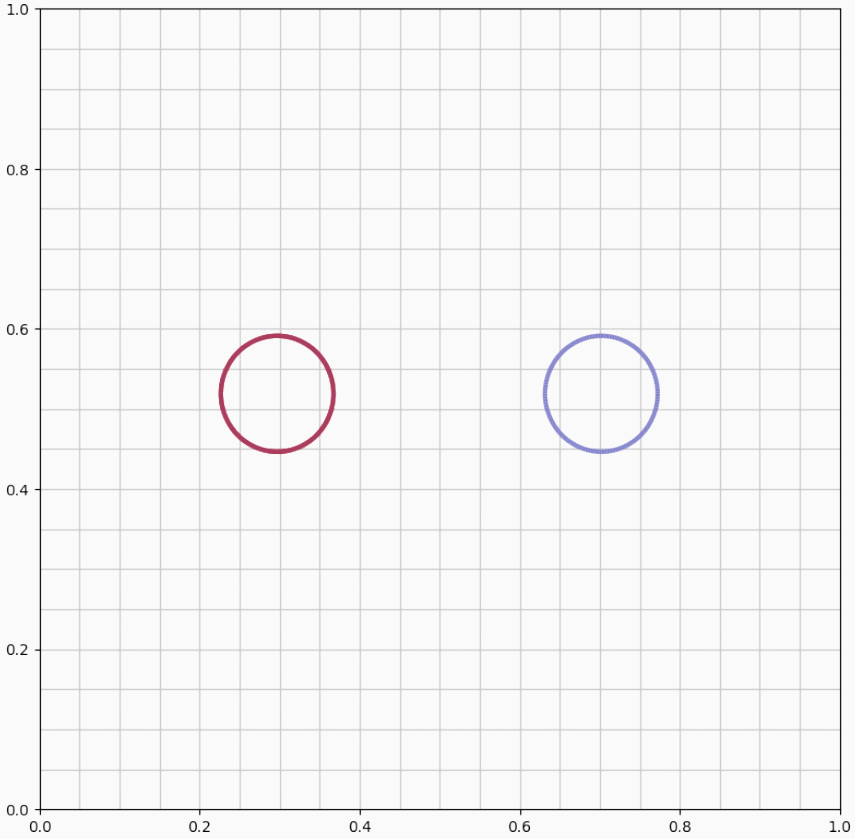
$$= \quad 1 + 2\cdot c \ \text{ on } (1) \qquad 1 - c \ \text{ on } (1)^{\perp}$$

$$g_q = \tfrac{1}{1+2\cdot c} \ \text{ on } (1) \qquad \tfrac{1}{1-c} \ \text{ on } (1)^{\perp}$$

$$v = v_{\text{mean}} + v_{\text{var}} \qquad \langle v, g_q v \rangle = \tfrac{1}{1+2\cdot c}\|v_{\text{mean}}\|_2^2 + \tfrac{1}{1-c}\|v_{\text{var}}\|_2^2$$

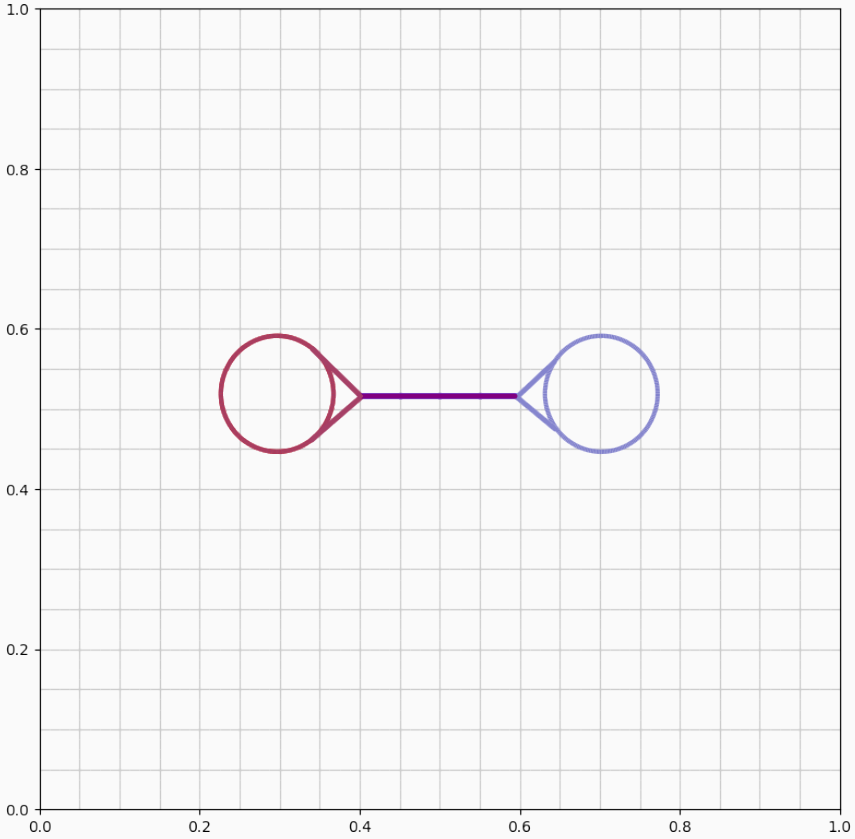
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



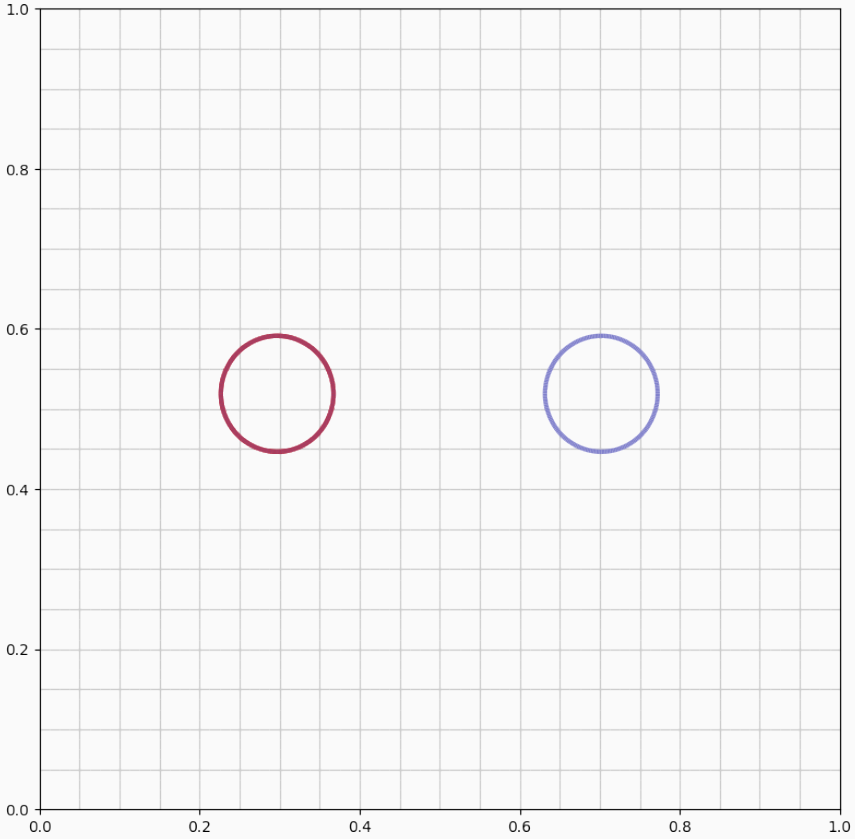
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



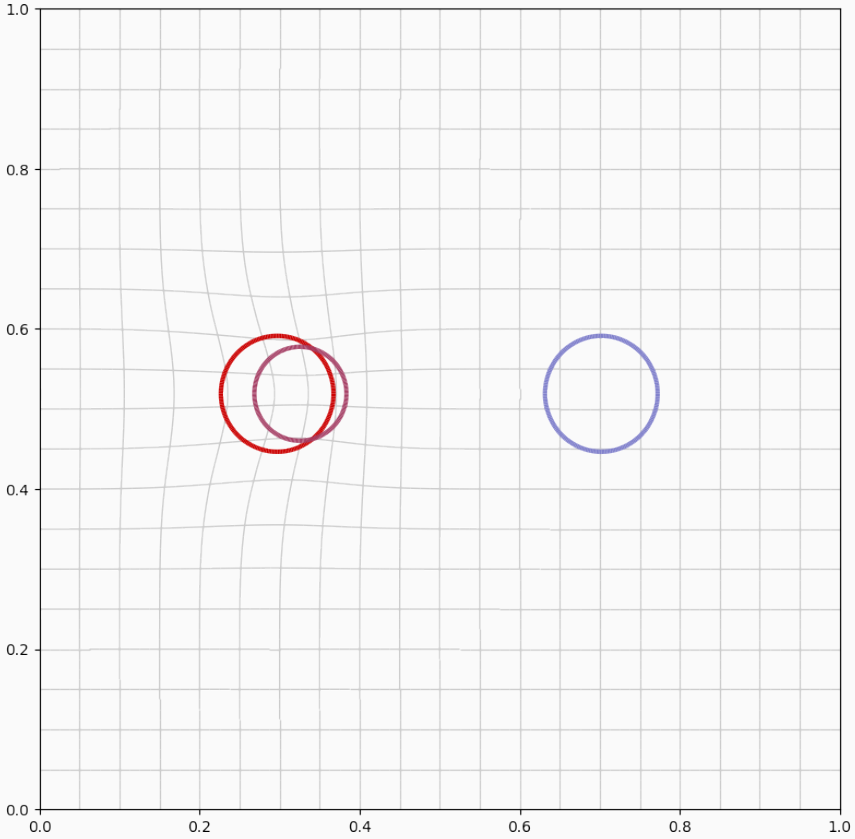
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



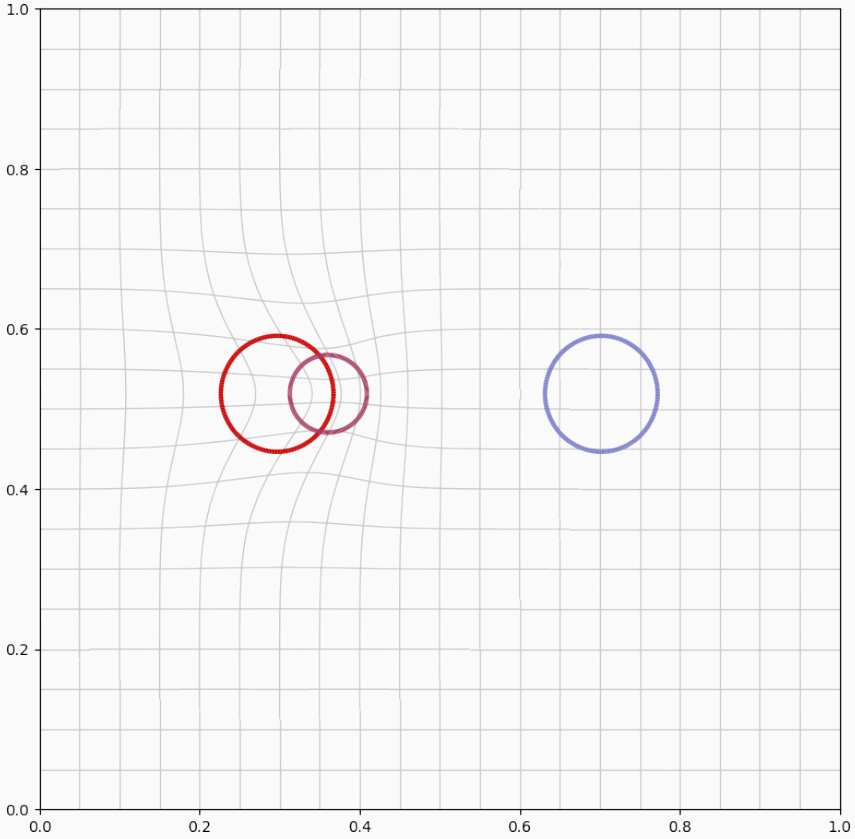
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



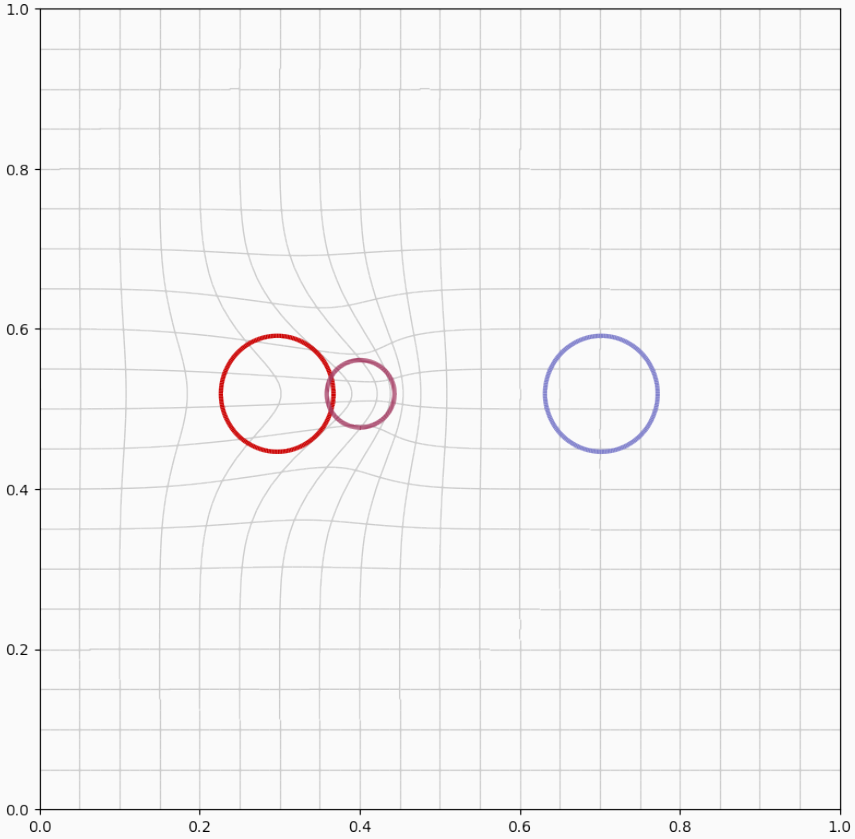
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



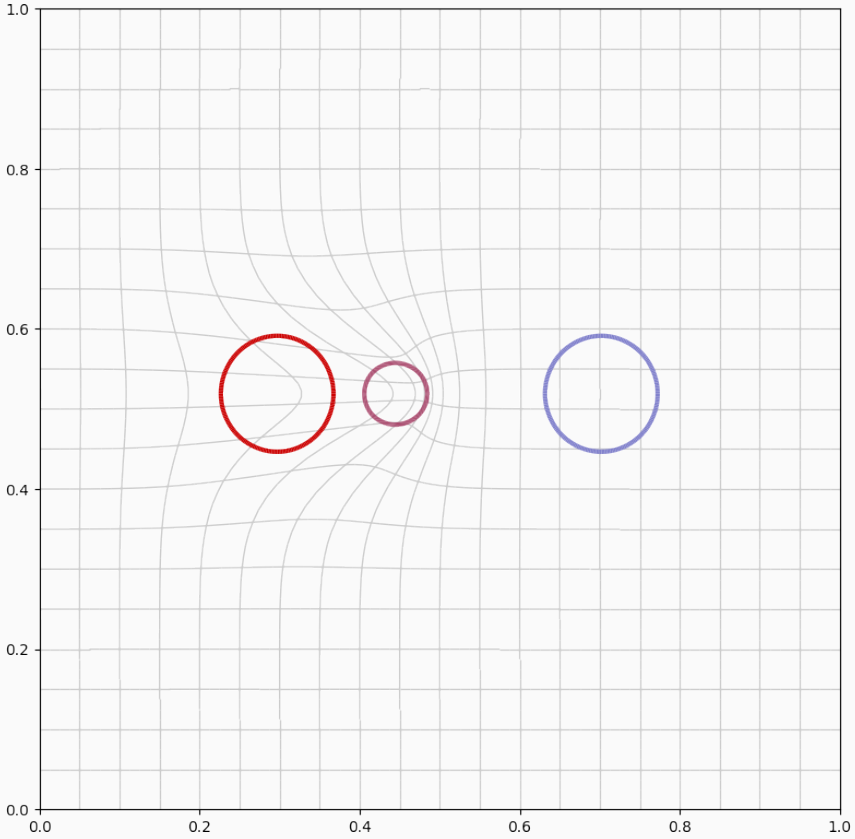
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



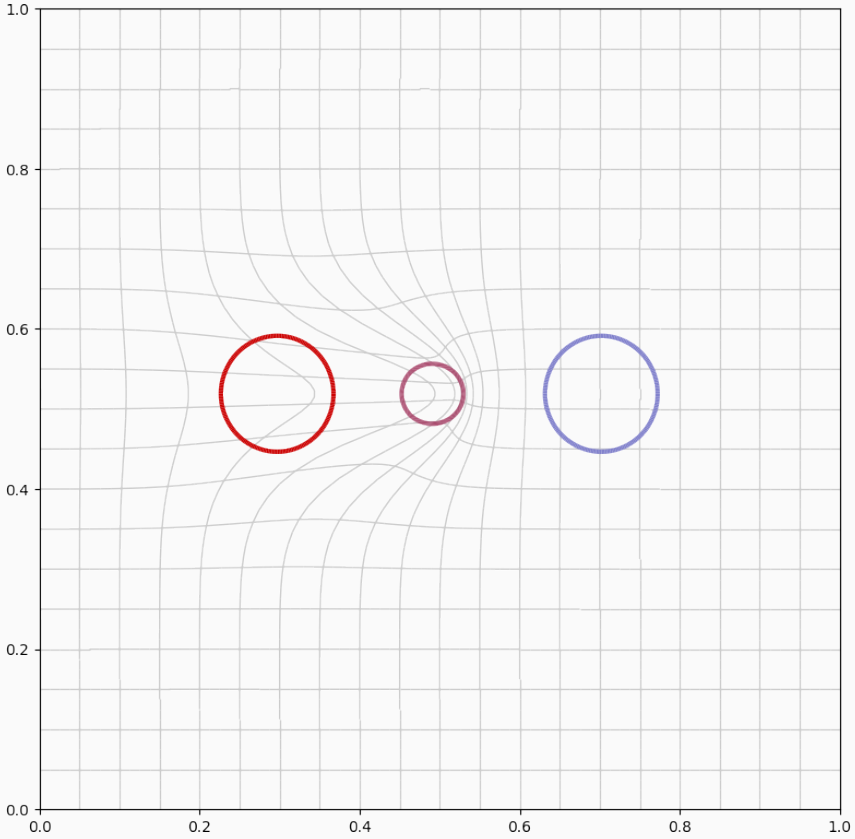
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



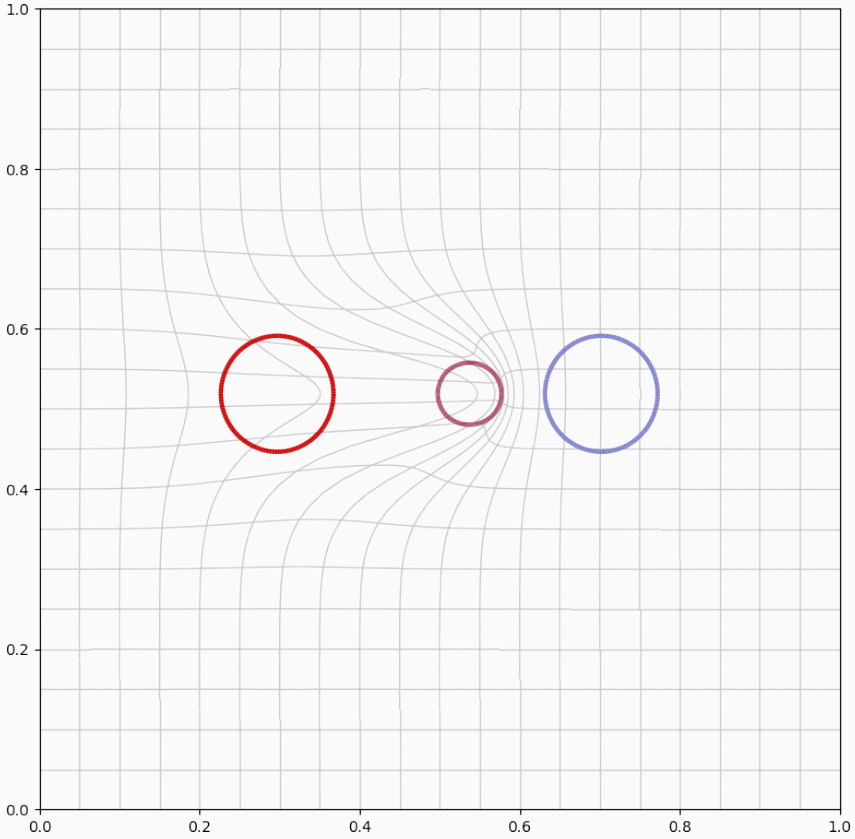
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



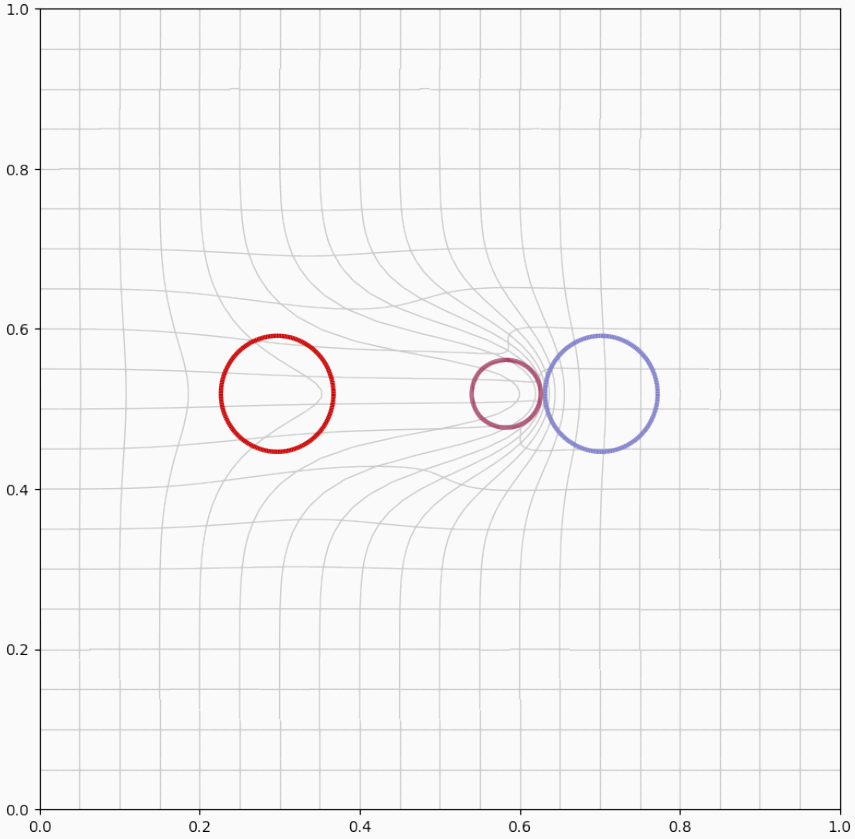
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2        }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



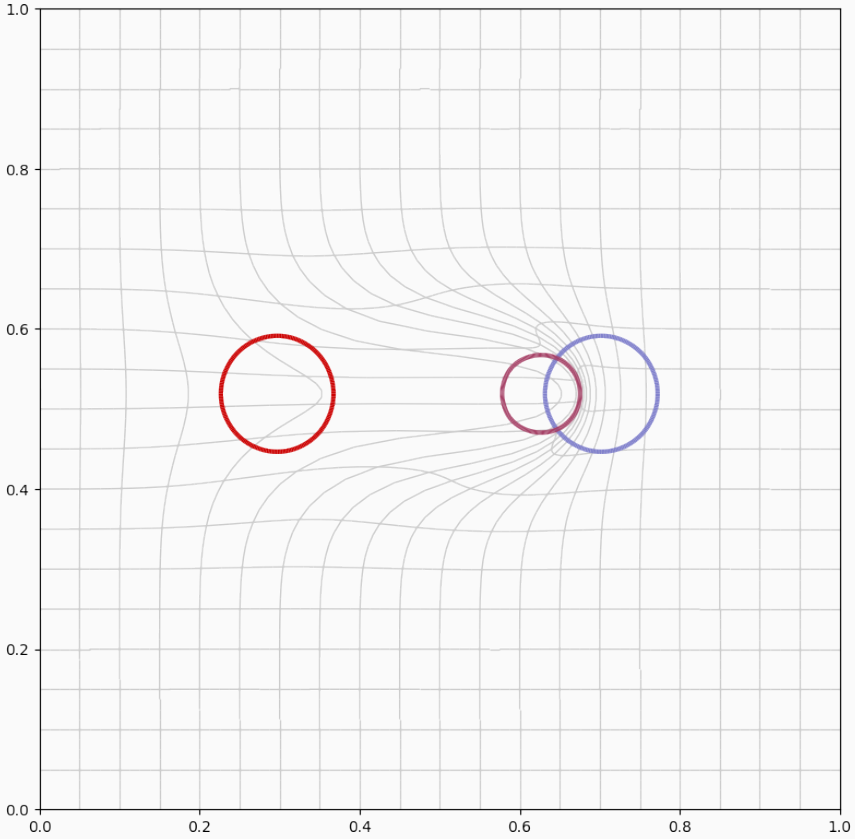
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



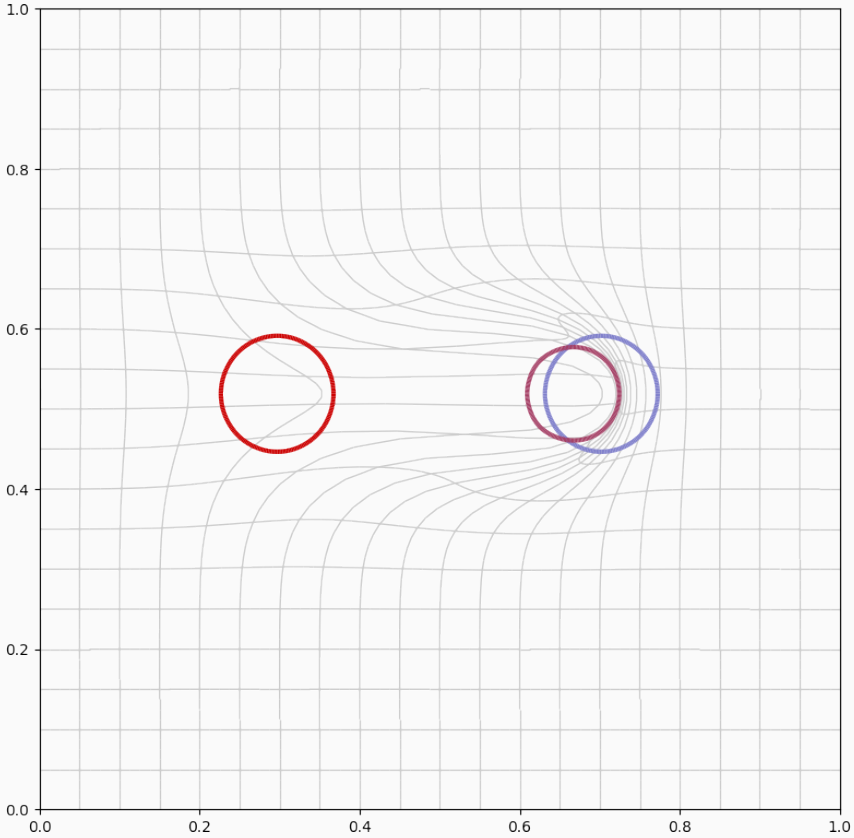
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



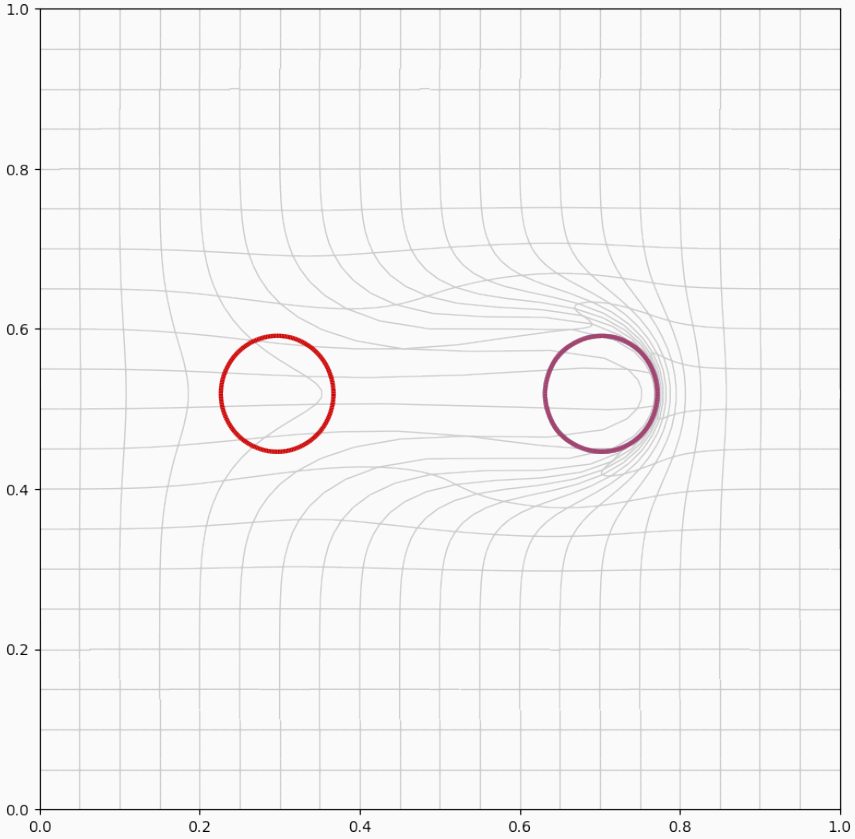
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



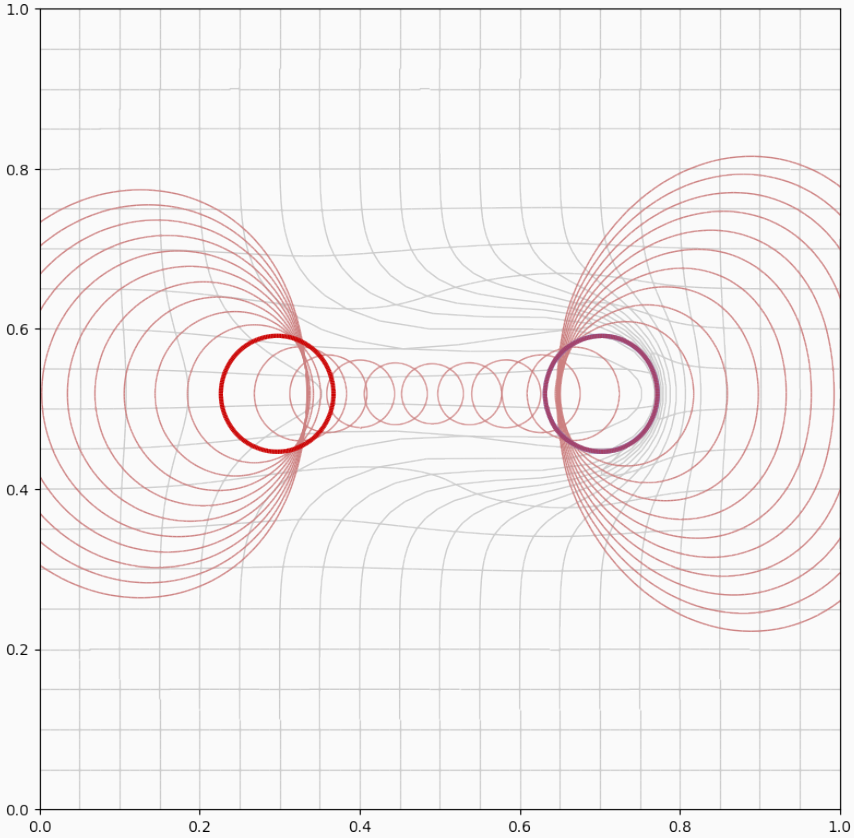
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



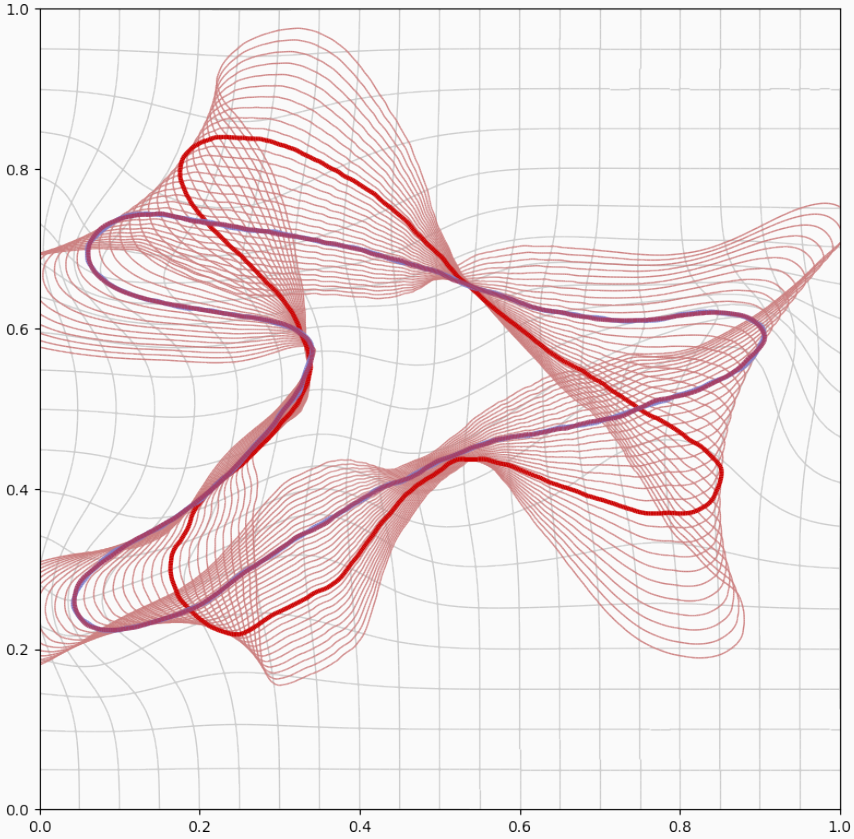
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



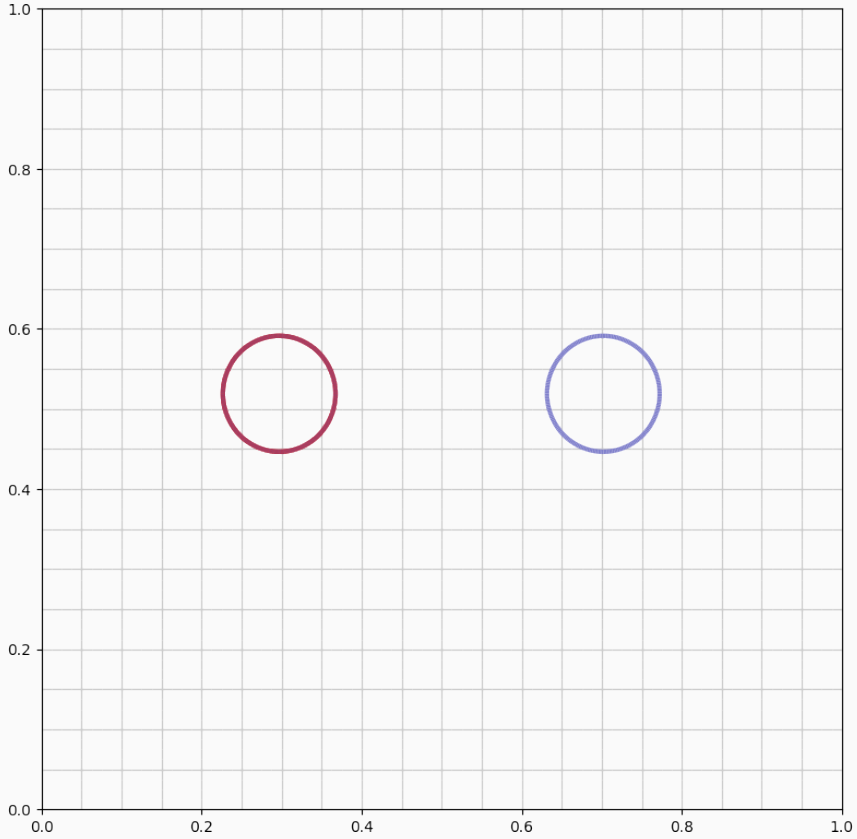
Kernels cometrics are not translation-friendly [MMM12]

```
k = { "id"    : Kernel("gaussian(x,y)") ,
      "gamma" :              1/.1**2       }
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



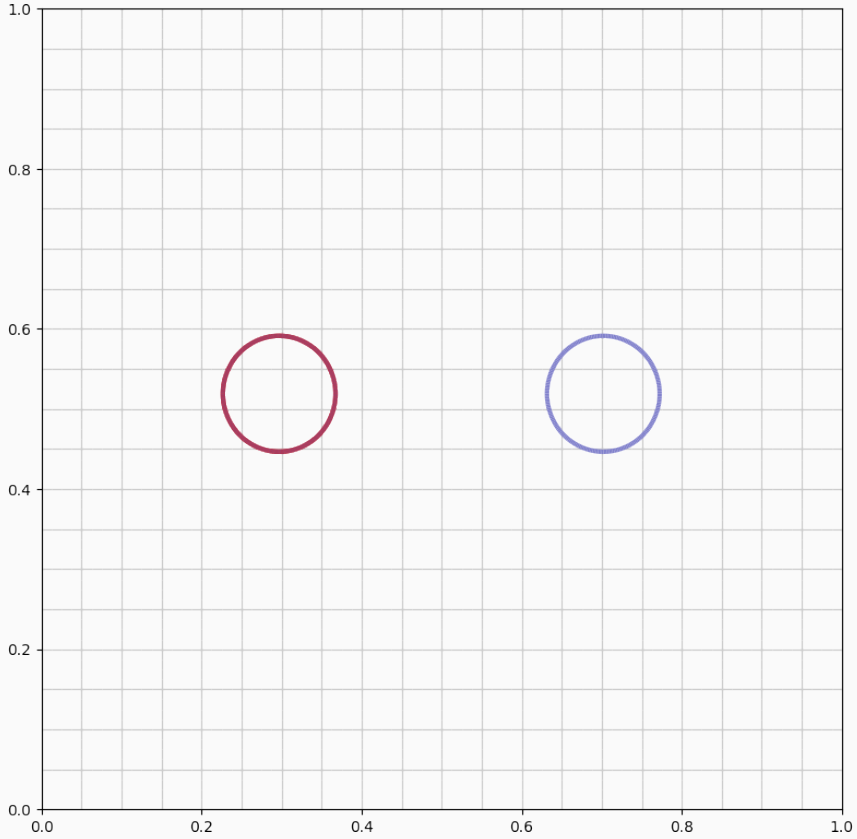
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



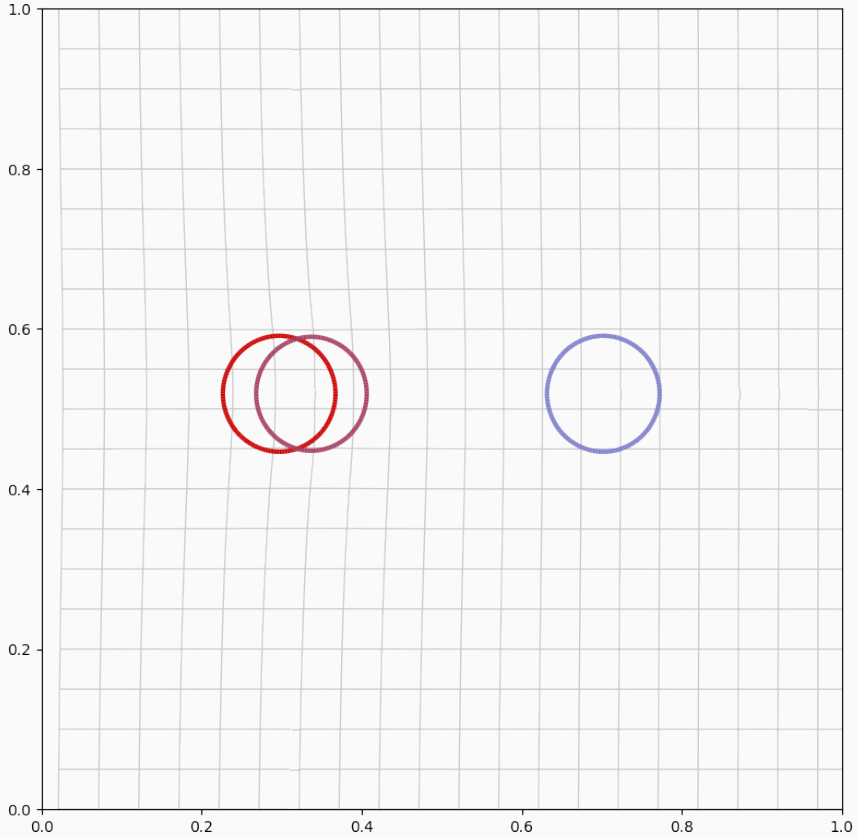
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



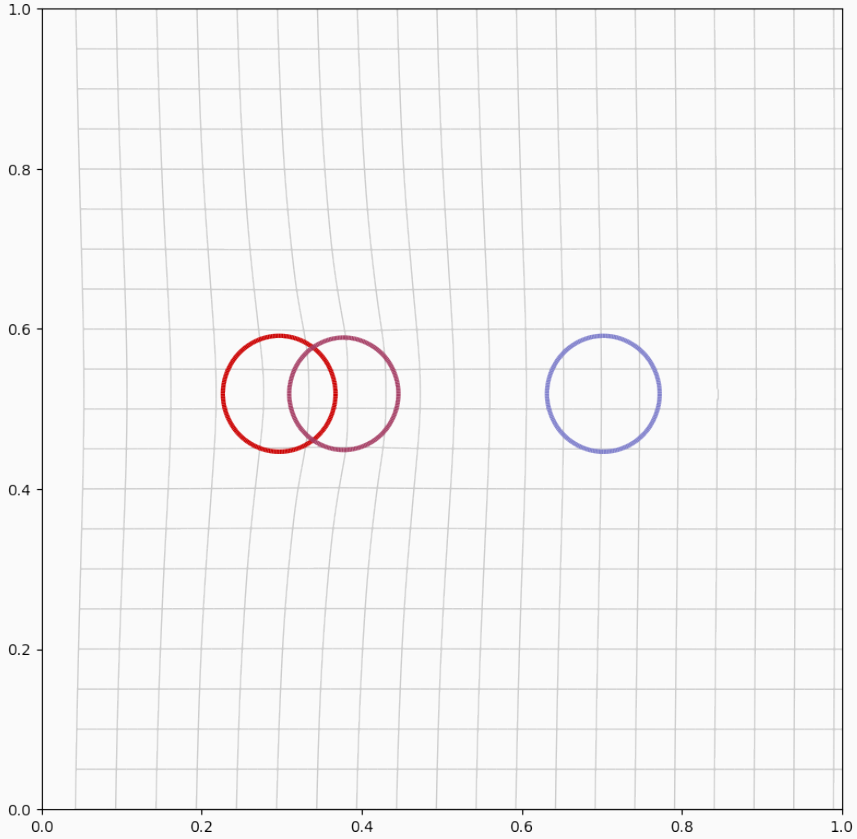
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



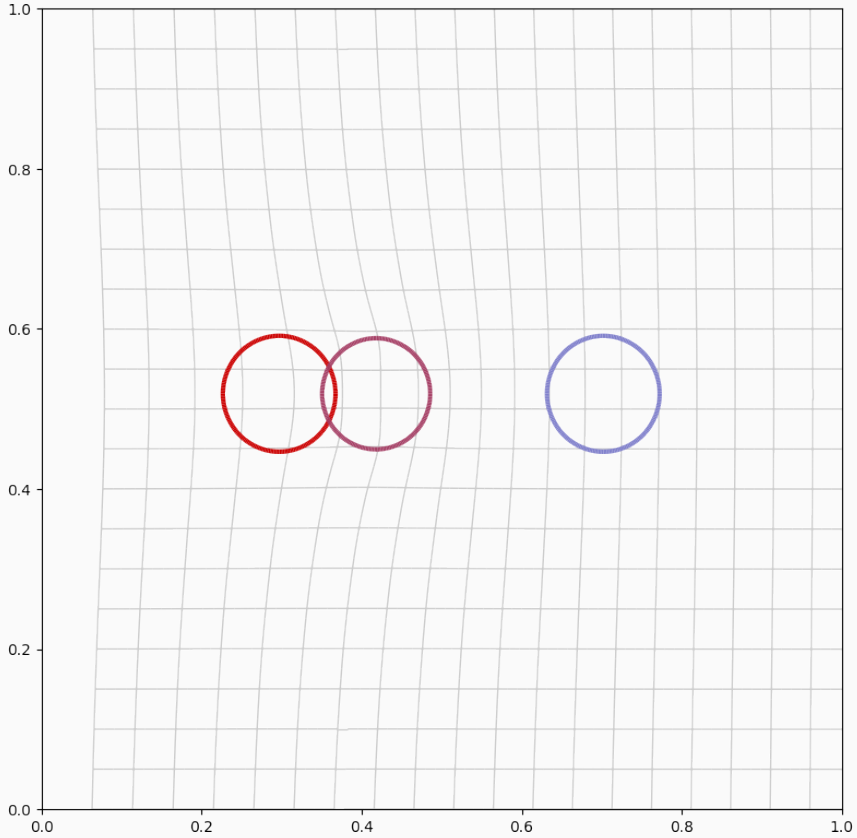
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



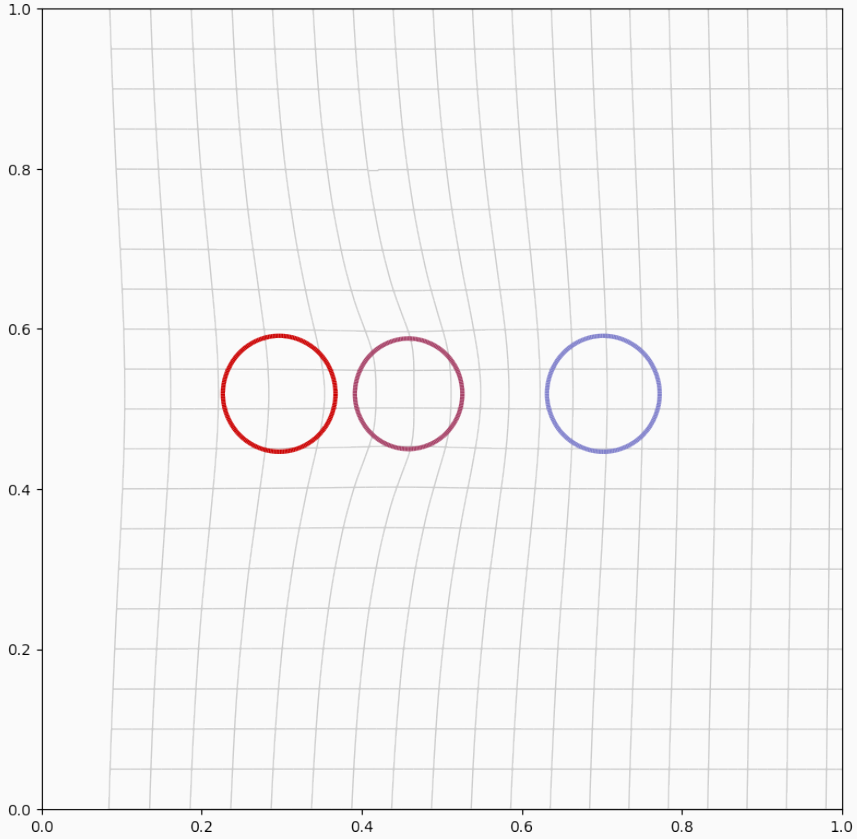
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 , 1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



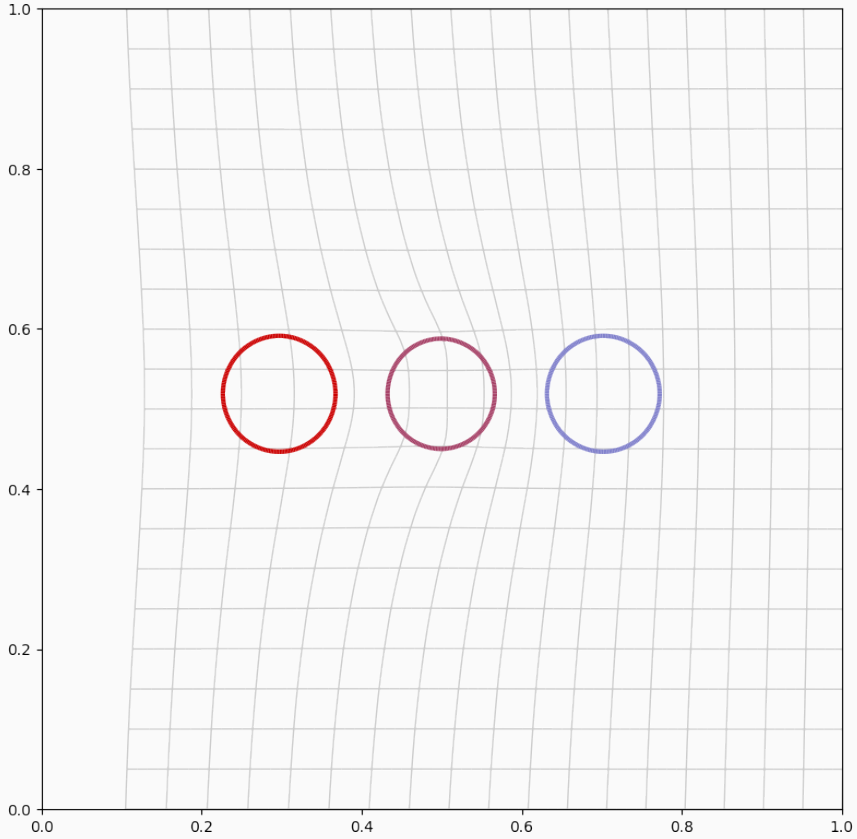
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



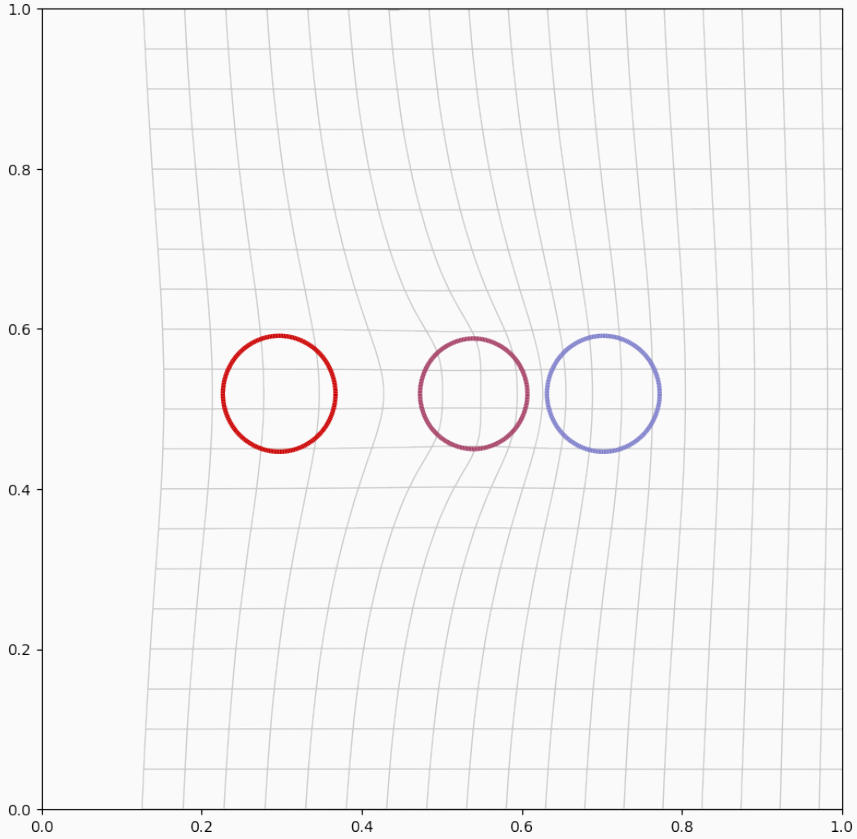
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



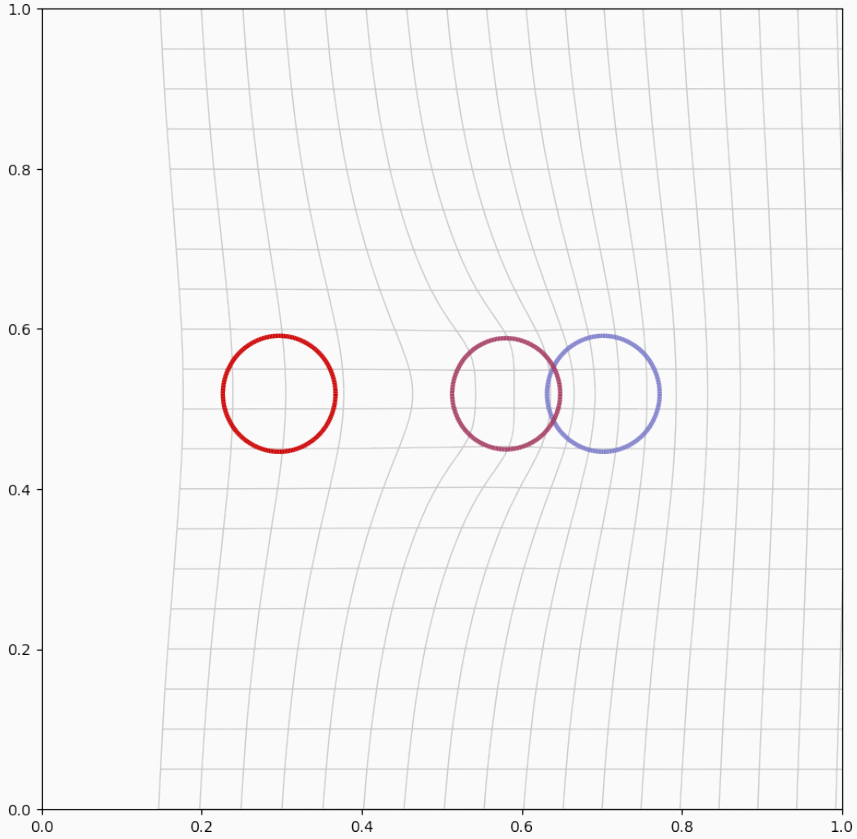
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



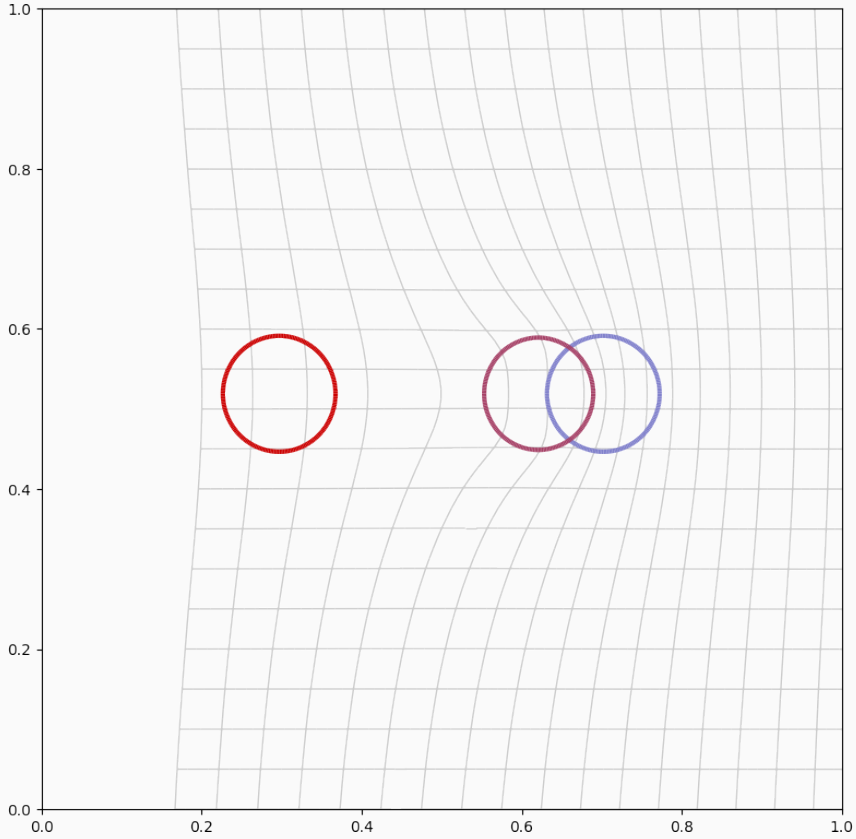
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



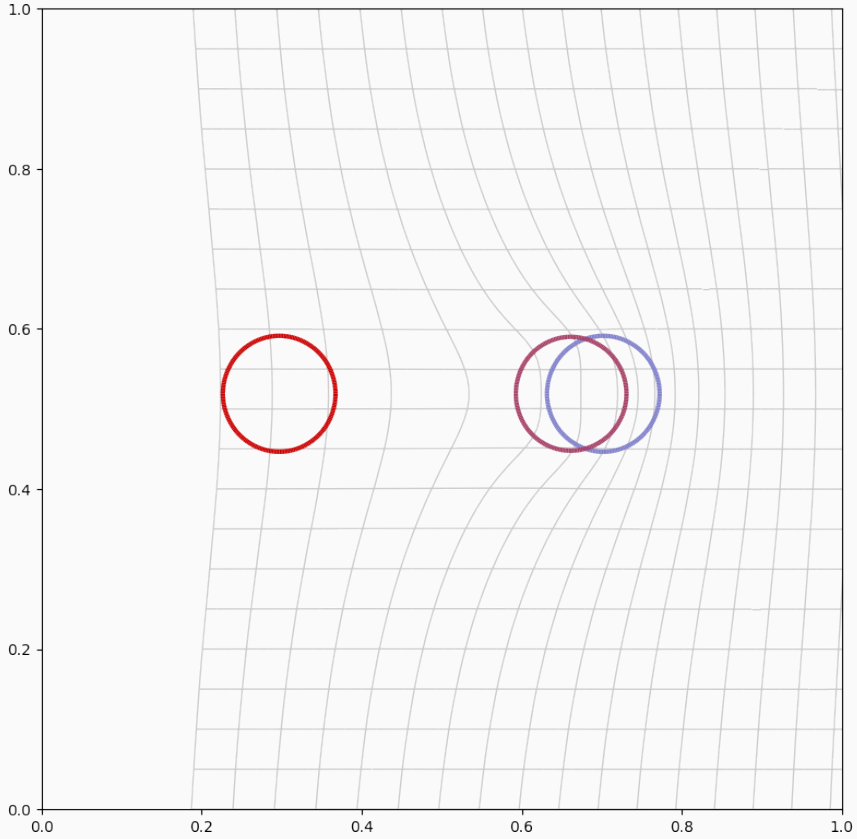
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



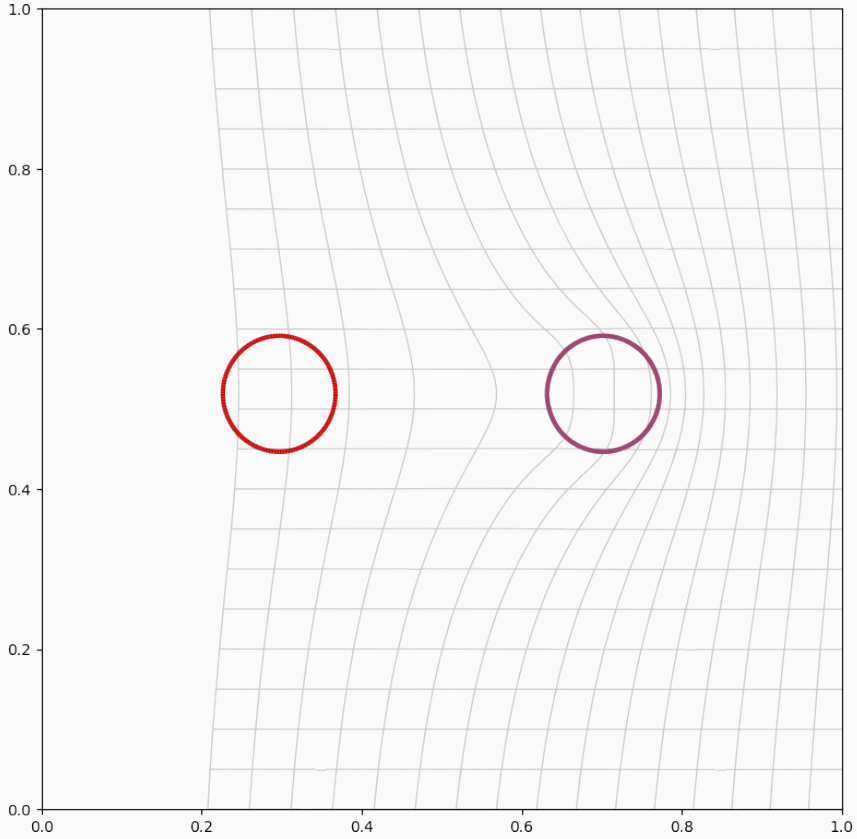
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



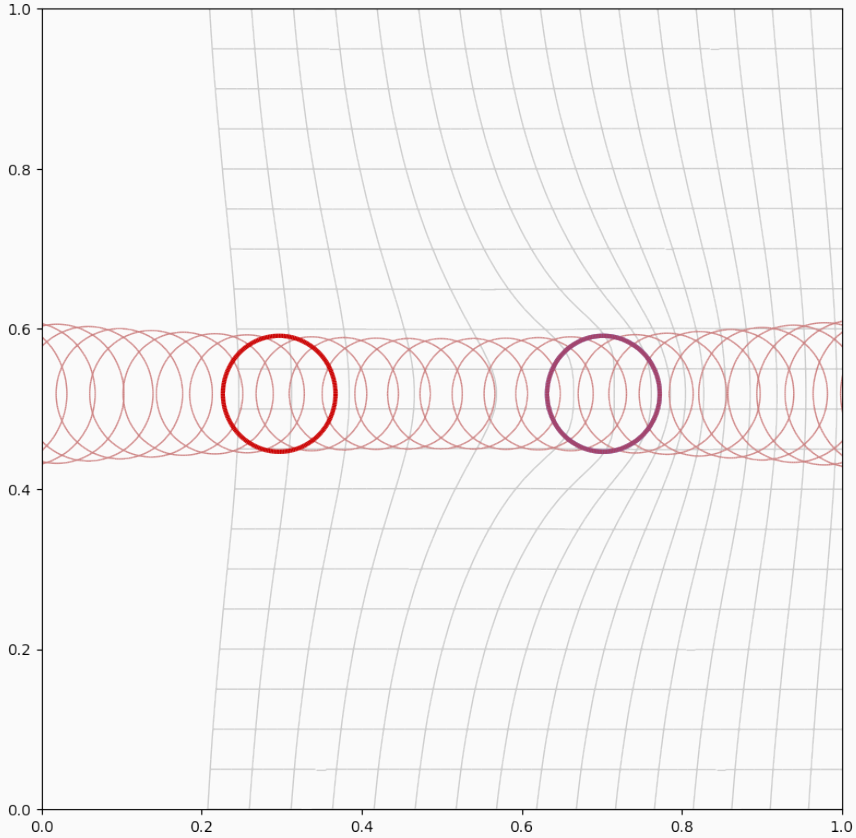
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



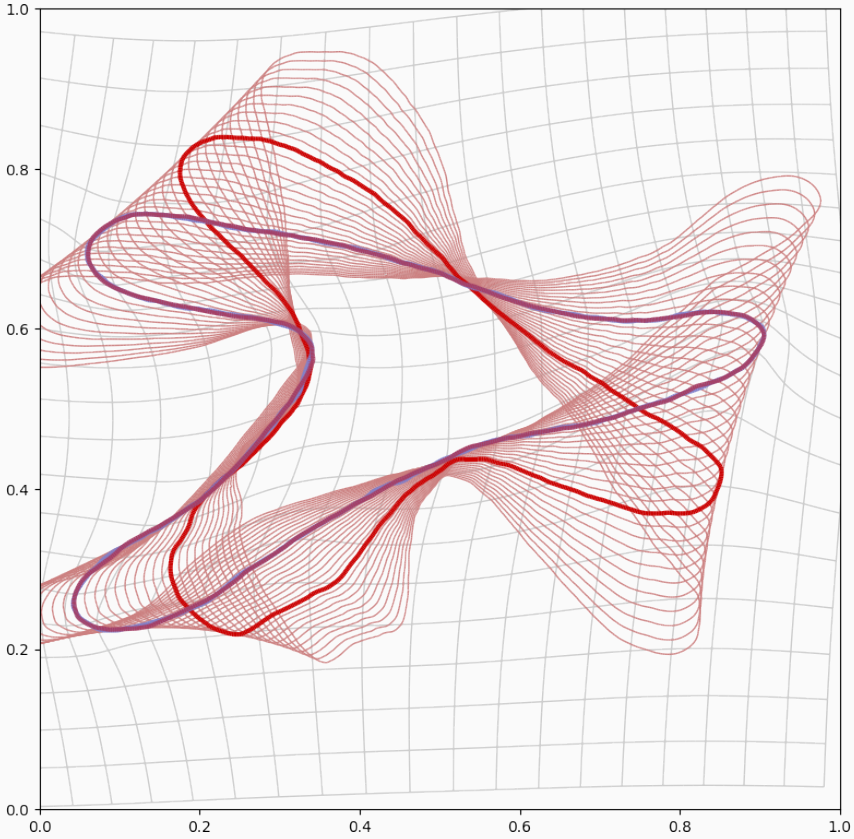
This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



This can be mitigated through the use of heavy-tail kernels

```
k = { "id"    : Kernel("1 + cauchy(x,y) + cauchy(x,y)"),
      "gamma" :                ( 1/.25**2 ,  1/.05**2 )}
def H(q,p) :
  return .5* dot( p, kernel_product( k, q, q, p) )
```



LDDMM is the cheapest Riemannian theory of shapes

- PyTorch has allowed us to **untangle** the shape space's geometry from the ODE solver.

LDDMM is the cheapest Riemannian theory of shapes

- PyTorch has allowed us to **untangle** the shape space's geometry from the ODE solver.
- LDDMM's kernel cometrics are the **cheapest**.

LDDMM is the cheapest Riemannian theory of shapes

- PyTorch has allowed us to **untangle** the shape space's geometry from the ODE solver.
- LDDMM's kernel cometrics are the **cheapest**.
- They are best understood through **convolution** operators $k \star \cdot$:
  If $\pi = \sum_i p_i \, \delta_{q_i}$,

$$\langle p \, , K_q \, p \rangle = \sum_{i,j} k(q_i - q_j) \, \langle p_i, p_j \rangle_2 = \langle \pi, \, k \star \pi \rangle = \langle \pi, \pi \rangle_k \, .$$

LDDMM is the cheapest Riemannian theory of shapes

- PyTorch has allowed us to **untangle** the shape space's geometry from the ODE solver.
- LDDMM's kernel cometrics are the **cheapest**.
- They are best understood through **convolution** operators $k \star \cdot$:
  If $\pi = \sum_i p_i \, \delta_{q_i}$,

$$\langle p \, , K_q \, p \rangle = \sum_{i,j} k(q_i - q_j) \, \langle p_i, p_j \rangle_2 = \langle \pi, \, k \star \pi \rangle = \langle \pi, \pi \rangle_k \, .$$

LDDMM is the cheapest Riemannian theory of shapes

- PyTorch has allowed us to **untangle** the shape space's geometry from the ODE solver.
- LDDMM's kernel cometrics are the **cheapest**.
- They are best understood through **convolution** operators $k \star \cdot$:
  If $\pi = \sum_i p_i \, \delta_{q_i}$,

$$\langle p, K_q p \rangle = \sum_{i,j} k(q_i - q_j) \langle p_i, p_j \rangle_2 = \langle \pi, k \star \pi \rangle = \langle \pi, \pi \rangle_k .$$

How do we go further?

A small step beyond LDDMM

What if we used:

```
k = ... # Your favorite kernel

def H(q,p) :
  l = Lambda(q) # an arbitrary N-vector
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```

A small step beyond LDDMM

What if we used:

```
k = ... # Your favorite kernel

def H(q,p) :
  l = Lambda(q) # an arbitrary N-vector
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```

The right theoretical setting:

Transport of a **measure** $\mu = \sum_i \mu_i \delta_{q_i}$, scalar field $\lambda_\mu : \mathbb{R}^D \to \mathbb{R}_{>0}$.

A small step beyond LDDMM

What if we used:

```
k = ... # Your favorite kernel

def H(q,p) :
  l = Lambda(q) # an arbitrary N-vector
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```

The right theoretical setting:

Transport of a **measure** $\mu = \sum_i \mu_i \delta_{q_i}$, scalar field $\lambda_\mu : \mathbb{R}^D \to \mathbb{R}_{>0}$.

$$\langle \pi, \pi \rangle_{k,\mu} = \langle \pi, K_\mu \, \pi \rangle$$

A small step beyond LDDMM

What if we used:

```
k = ... # Your favorite kernel

def H(q,p) :
  l = Lambda(q) # an arbitrary N-vector
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```

The right theoretical setting:

Transport of a **measure** $\mu = \sum_i \mu_i \delta_{q_i}$, scalar field $\lambda_\mu : \mathbb{R}^D \to \mathbb{R}_{>0}$.

$$\langle \pi, \pi \rangle_{k,\mu} = \langle \pi, K_\mu \, \pi \rangle = \big\langle \lambda_\mu \pi, k \star (\lambda_\mu \pi) \big\rangle$$

A small step beyond LDDMM

What if we used:

```
k = ... # Your favorite kernel

def H(q,p) :
  l = Lambda(q) # an arbitrary N-vector
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```

The right theoretical setting:

Transport of a **measure** $\mu = \sum_i \mu_i \delta_{q_i}$, scalar field $\lambda_\mu : \mathbb{R}^D \to \mathbb{R}_{>0}$.

$$\langle \pi, \pi \rangle_{k,\mu} = \langle \pi, K_\mu \, \pi \rangle = \big\langle \lambda_\mu \pi, k \star (\lambda_\mu \pi) \big\rangle = \sum_{i,j} k(q_i - q_j) \, \big\langle \ell_i p_i, \ell_j p_j \big\rangle_2$$

A small step beyond LDDMM

What if we used:

```
k = ... # Your favorite kernel

def H(q,p) :
  l = Lambda(q) # an arbitrary N-vector
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```

The right theoretical setting:

Transport of a **measure** $\mu = \sum_i \mu_i \delta_{q_i}$, scalar field $\lambda_\mu : \mathbb{R}^D \to \mathbb{R}_{>0}$.

$$\langle \pi, \pi \rangle_{k,\mu} = \langle \pi, K_\mu \, \pi \rangle = \big\langle \lambda_\mu \pi, k \star (\lambda_\mu \pi) \big\rangle = \sum_{i,j} k(q_i - q_j) \, \big\langle \ell_i p_i, \ell_j p_j \big\rangle_2$$

$$\ell_i = \lambda_\mu(q_i), \qquad K_\mu = \Delta(\ell) \, K_q \, \Delta(\ell)$$

Normalizing kernels dynamically

The new `H` and `Lambda` routines allow us to **scale the kernel matrix.**

Normalizing kernels dynamically

The new `H` and `Lambda` routines allow us to **scale the kernel matrix.**

Let's use this degree of freedom to **normalize** it:

Normalizing kernels dynamically

The new H and Lambda routines allow us to **scale the kernel matrix.**

Let's use this degree of freedom to **normalize** it:

```
def Lambda( q, mu ) :
  l = torch.ones(N)
  for i in range(nits) : # nits=2 is ok
    l = ( l / kernel_product(k, q, q, l*mu) ).sqrt()
  return l
```

Normalizing kernels dynamically

The new H and Lambda routines allow us to **scale the kernel matrix.**
Let's use this degree of freedom to **normalize** it:

```
def Lambda( q, mu ) :
  l = torch.ones(N)
  for i in range(nits) : # nits=2 is ok
    l = ( l / kernel_product(k, q, q, l*mu) ).sqrt()
  return l
```

Symmetric Sinkhorn Theorem [KRU14]:
These iterations converge towards the unique field λ_μ such that

$$\lambda_\mu \cdot k \star (\lambda_\mu \mu) \;=\; 1, \quad \text{i.e.} \quad K_\mu \mu \;=\; \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Properties

- The ambient vector field is given by

$$v(x) = K_\mu \pi = \lambda_\mu \cdot k \star (\lambda_\mu \pi)$$

Properties

- The ambient vector field is given by

$$v(x) = K_\mu \pi = \lambda_\mu \cdot k \star (\lambda_\mu \pi)$$
$$= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)}$$

Properties

- The ambient vector field is given by

$$v(x) = K_\mu \pi = \lambda_\mu \cdot k \star (\lambda_\mu \pi)$$

$$= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)} = \frac{k \star (\lambda_\mu \mu \frac{\mathrm{d}\pi}{\mathrm{d}\mu})}{k \star (\lambda_\mu \mu)}$$

Properties

- The ambient vector field is given by

$$\begin{aligned} v(x) &= K_\mu \pi = \lambda_\mu \cdot k \star (\lambda_\mu \pi) \\ &= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)} = \frac{k \star (\lambda_\mu \mu \frac{\mathrm{d}\pi}{\mathrm{d}\mu})}{k \star (\lambda_\mu \mu)} \\ &= \text{Weighted } \textbf{barycenter} \text{ of the } \frac{p_i}{\mu_i} \end{aligned}$$

Properties

- The ambient vector field is given by

$$\begin{aligned} v(x) &= K_\mu \pi = \lambda_\mu \cdot k \star (\lambda_\mu \pi) \\ &= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)} = \frac{k \star (\lambda_\mu \mu \frac{\mathrm{d}\pi}{\mathrm{d}\mu})}{k \star (\lambda_\mu \mu)} \\ &= \text{Weighted } \textbf{barycenter} \text{ of the } \frac{p_i}{\mu_i} \end{aligned}$$

- **Translating** μ by a vector v is a **geodesic** trajectory associated to $\pi = \mu v$,

Properties

- The ambient vector field is given by

$$\begin{aligned} v(x) \;=\; K_\mu \pi \;&=\; \lambda_\mu \cdot k \star (\lambda_\mu \pi) \\ &= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)} \;=\; \frac{k \star (\lambda_\mu \mu \frac{\mathrm{d}\pi}{\mathrm{d}\mu})}{k \star (\lambda_\mu \mu)} \\ &= \text{Weighted } \textbf{barycenter} \text{ of the } \frac{p_i}{\mu_i} \end{aligned}$$

- **Translating** μ by a vector v is a **geodesic** trajectory associated to $\pi = \mu v$, with cost

$$\langle \pi, \pi \rangle_{k,\mu} \;=\; \langle \pi, K_\mu \, \pi \rangle$$

Properties

- The ambient vector field is given by

$$\begin{aligned} v(x) \;=\; K_\mu \pi \;&=\; \lambda_\mu \cdot k \star (\lambda_\mu \pi) \\ &= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)} \;=\; \frac{k \star (\lambda_\mu \mu \frac{\mathrm{d}\pi}{\mathrm{d}\mu})}{k \star (\lambda_\mu \mu)} \\ &= \text{Weighted } \textbf{barycenter} \text{ of the } \frac{p_i}{\mu_i} \end{aligned}$$

- **Translating** μ by a vector v is a **geodesic** trajectory associated to $\pi = \mu v$, with cost

$$\langle \pi, \pi \rangle_{k,\mu} \;=\; \langle \pi, K_\mu \, \pi \rangle \;=\; \langle \mu v, \underbrace{K_\mu \mu}_{1} \, v \rangle \;=\; \langle \mu v, v \rangle$$

Properties

- The ambient vector field is given by

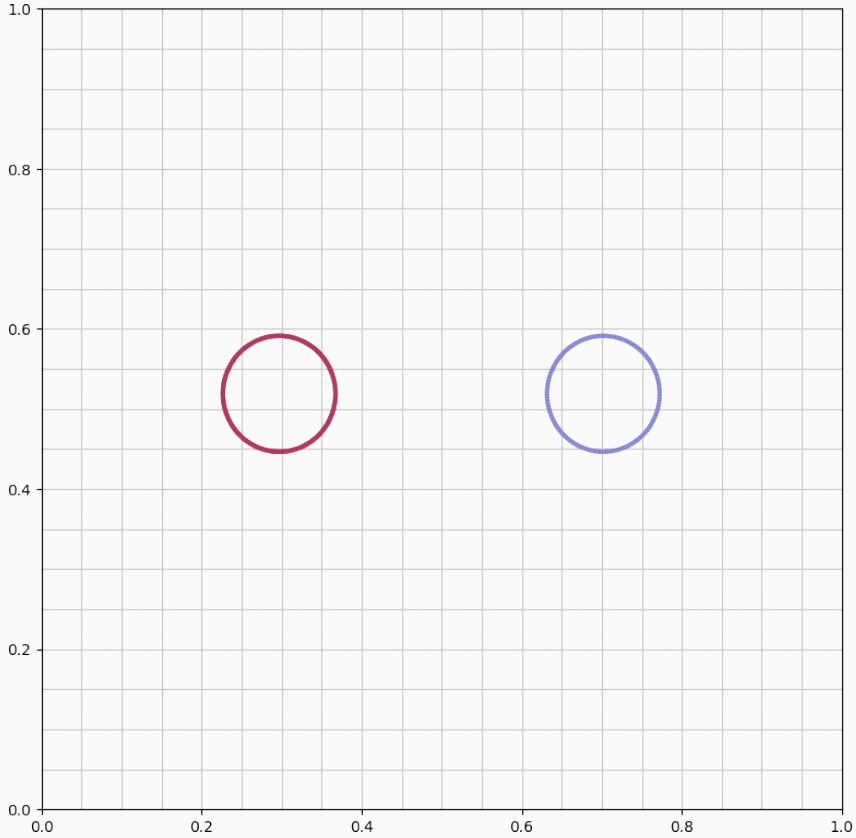
$$v(x) = K_\mu \pi = \lambda_\mu \cdot k \star (\lambda_\mu \pi)$$
$$= \frac{k \star (\lambda_\mu \pi)}{k \star (\lambda_\mu \mu)} = \frac{k \star (\lambda_\mu \mu \frac{\mathrm{d}\pi}{\mathrm{d}\mu})}{k \star (\lambda_\mu \mu)}$$
$$= \text{Weighted } \textbf{barycenter} \text{ of the } \frac{p_i}{\mu_i}$$

- **Translating** μ by a vector v is a **geodesic** trajectory associated to $\pi = \mu v$, with cost

$$\langle \pi, \pi \rangle_{k,\mu} = \langle \pi, K_\mu \pi \rangle = \langle \mu v, \underbrace{K_\mu \mu}_{1} v \rangle = \langle \mu v, v \rangle = |\mu| \cdot \|v\|_2^2 .$$

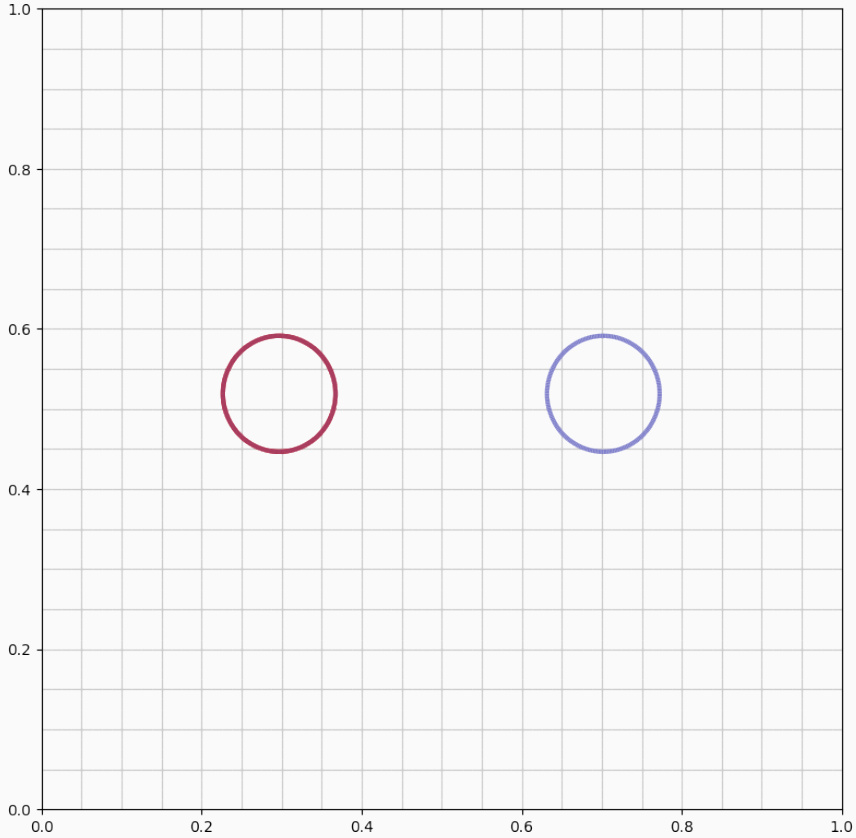
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



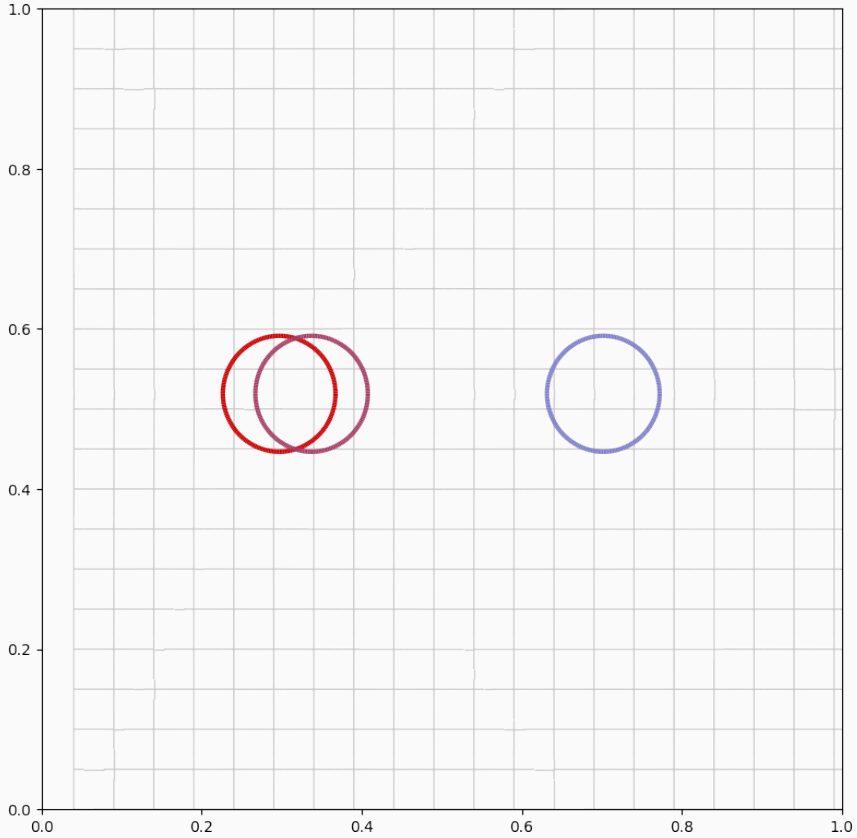
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



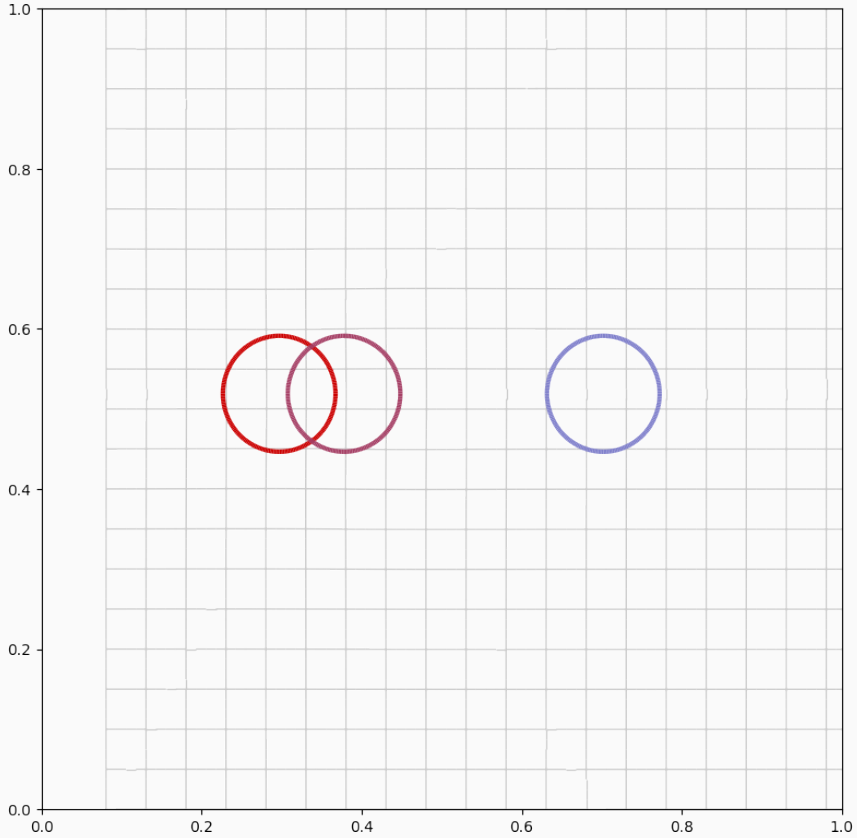
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



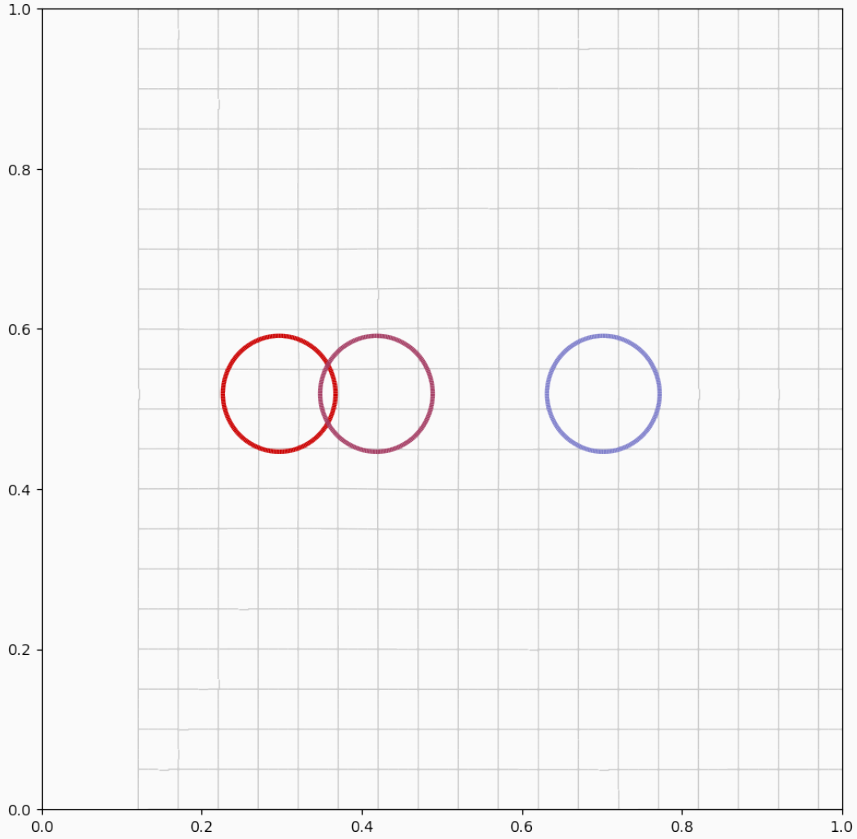
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



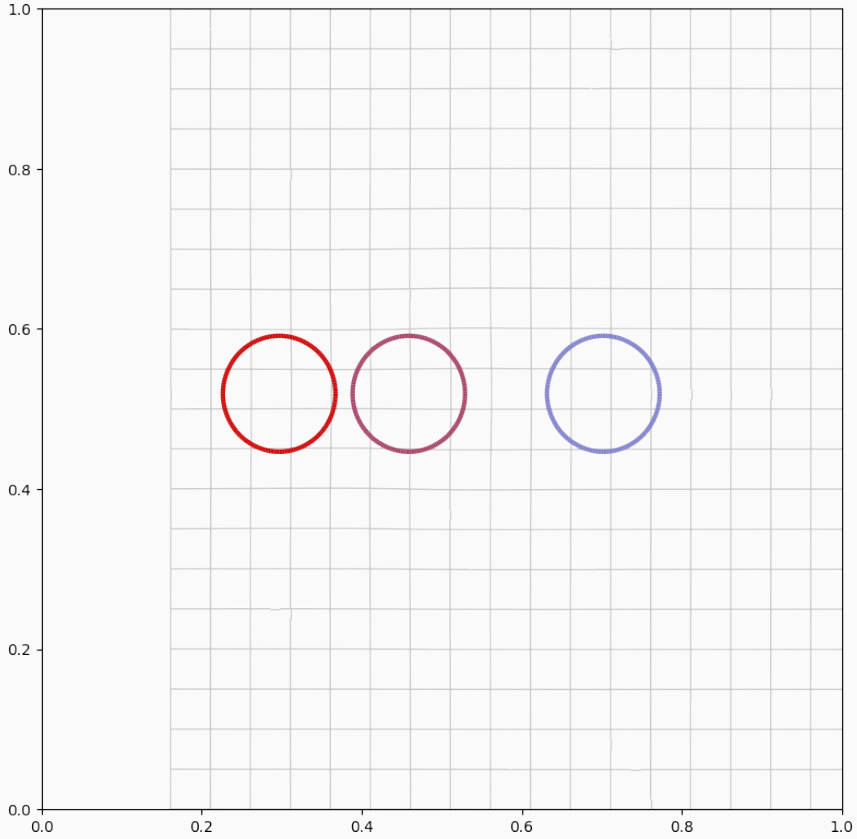
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



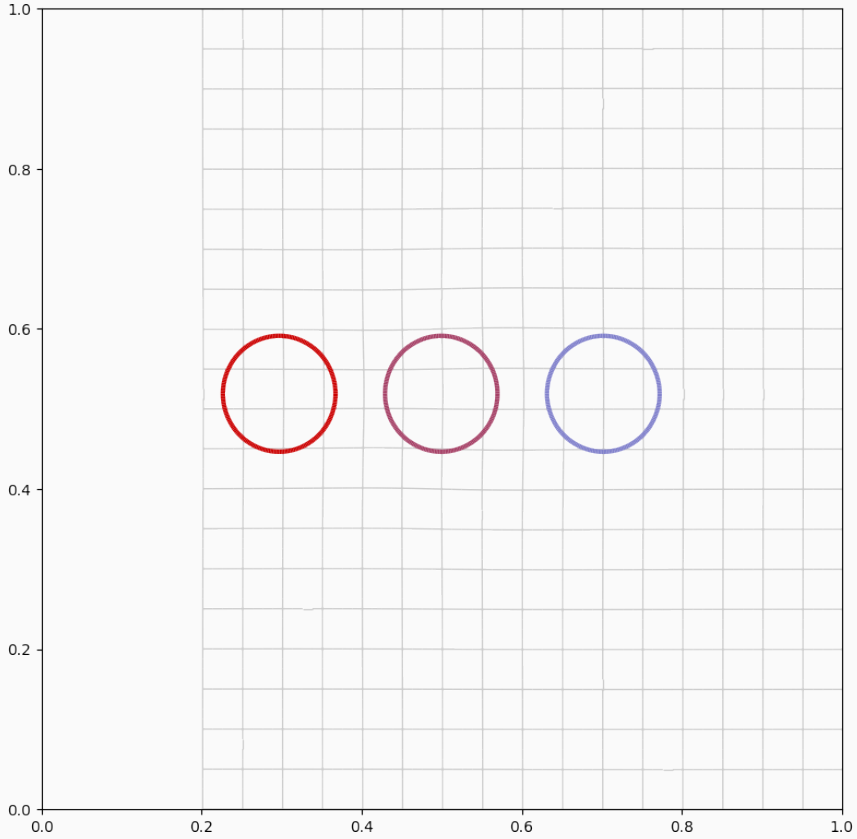
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



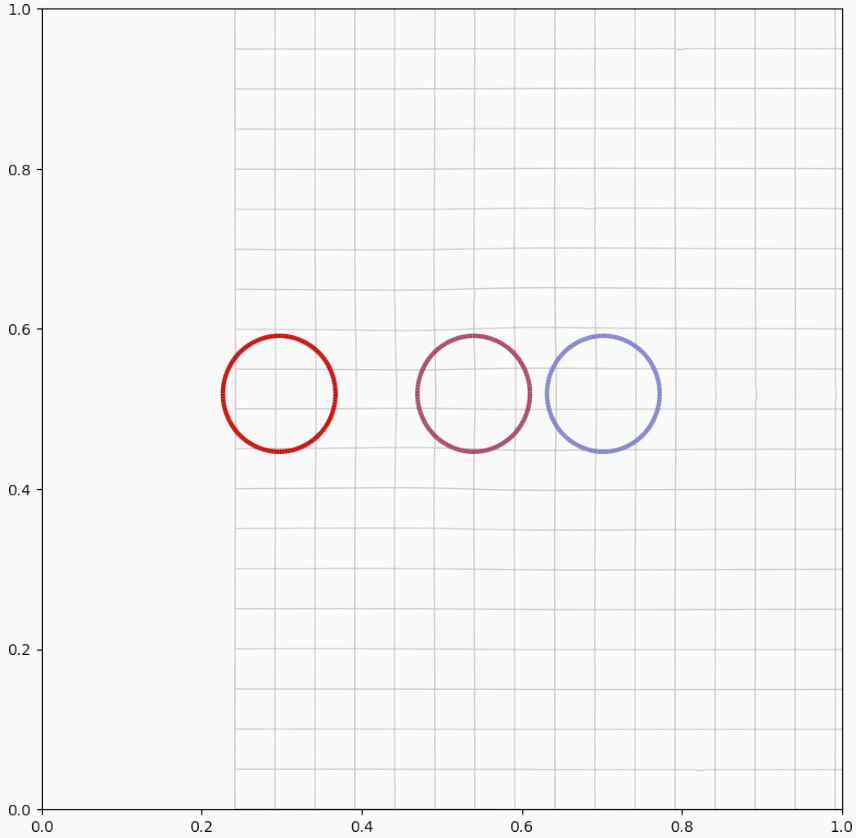
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



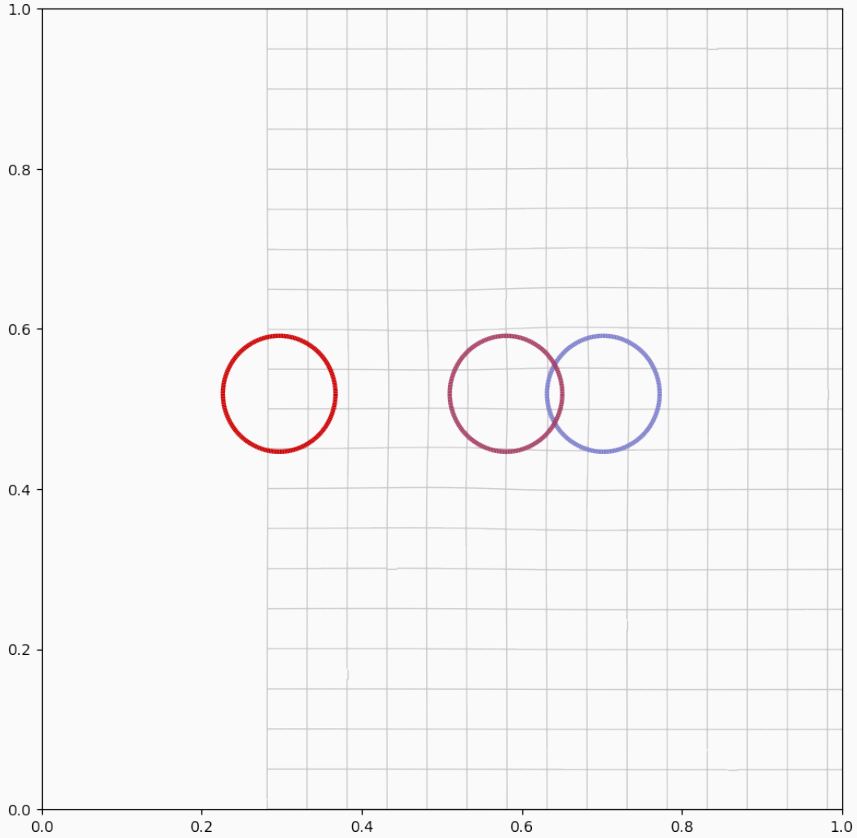
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



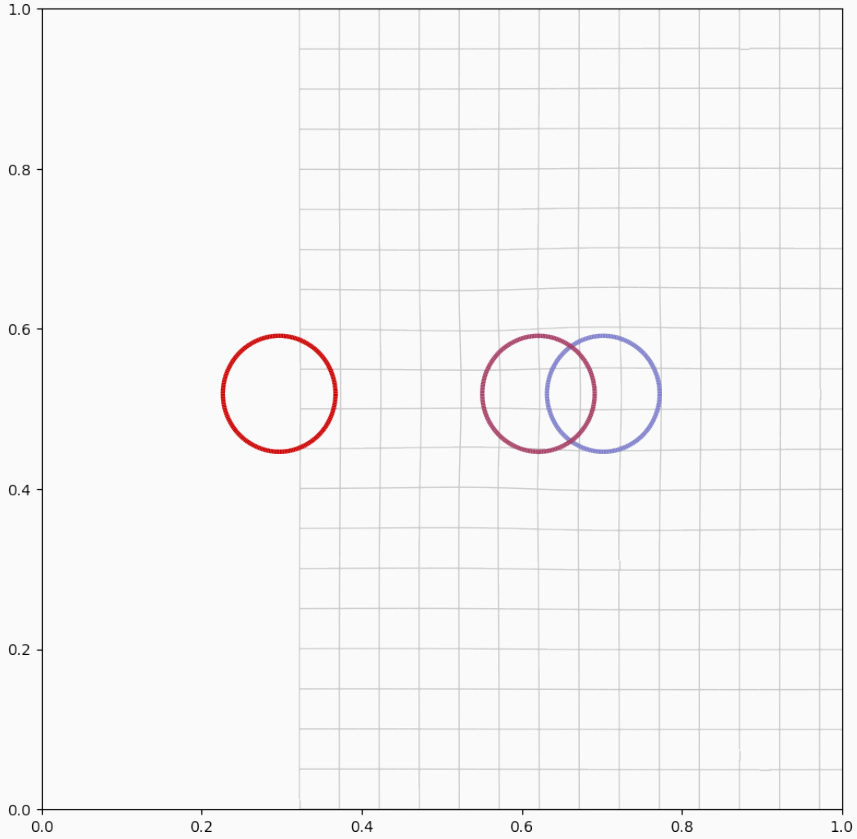
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



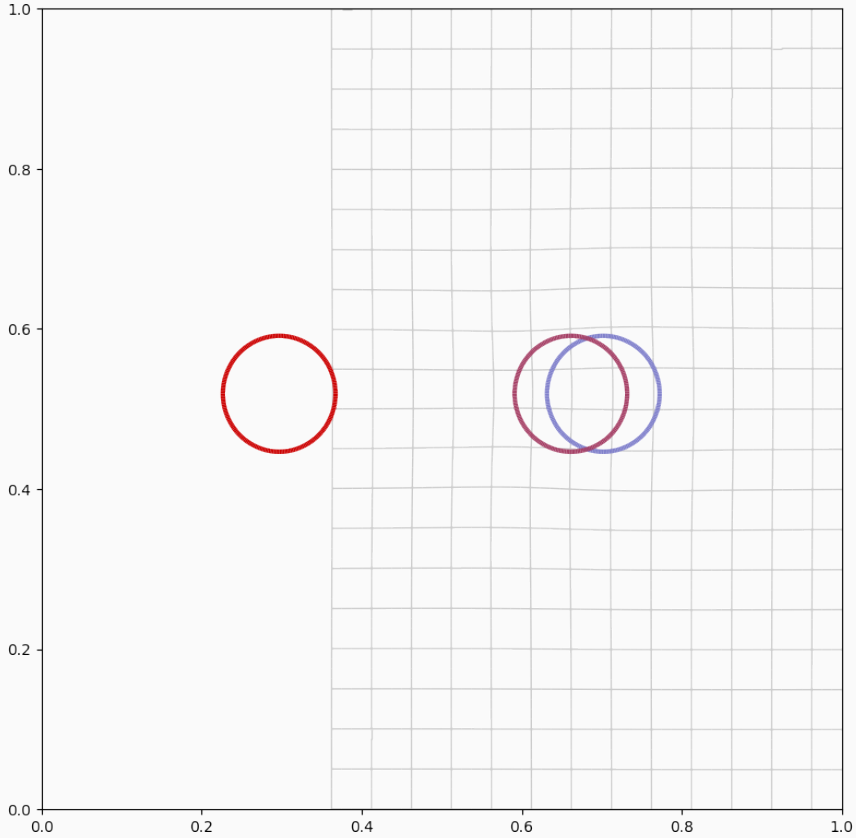
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



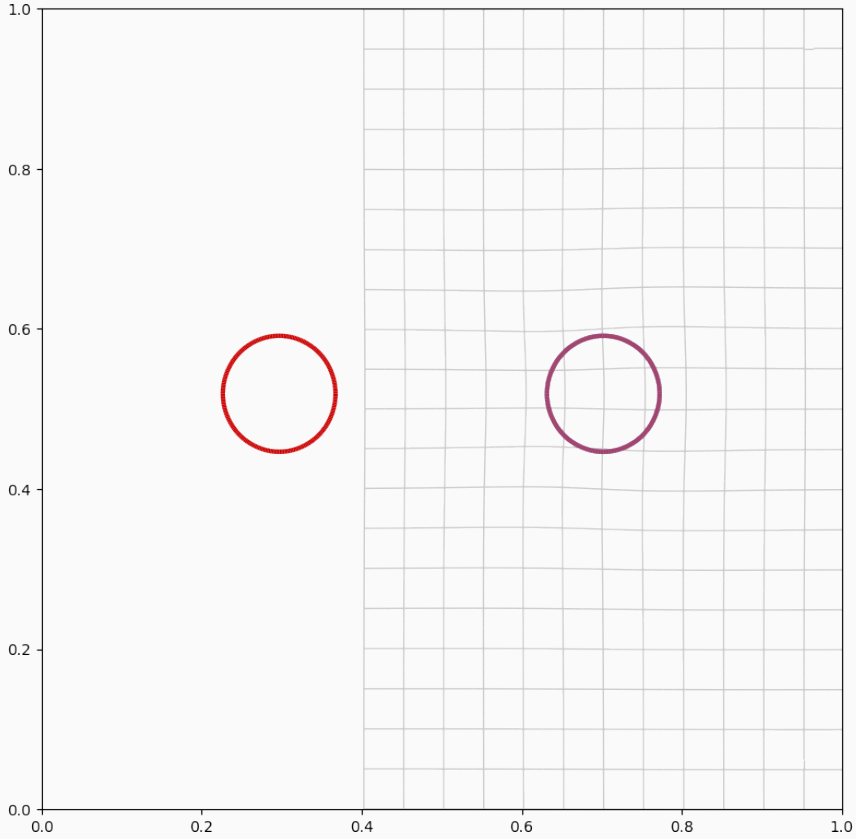
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



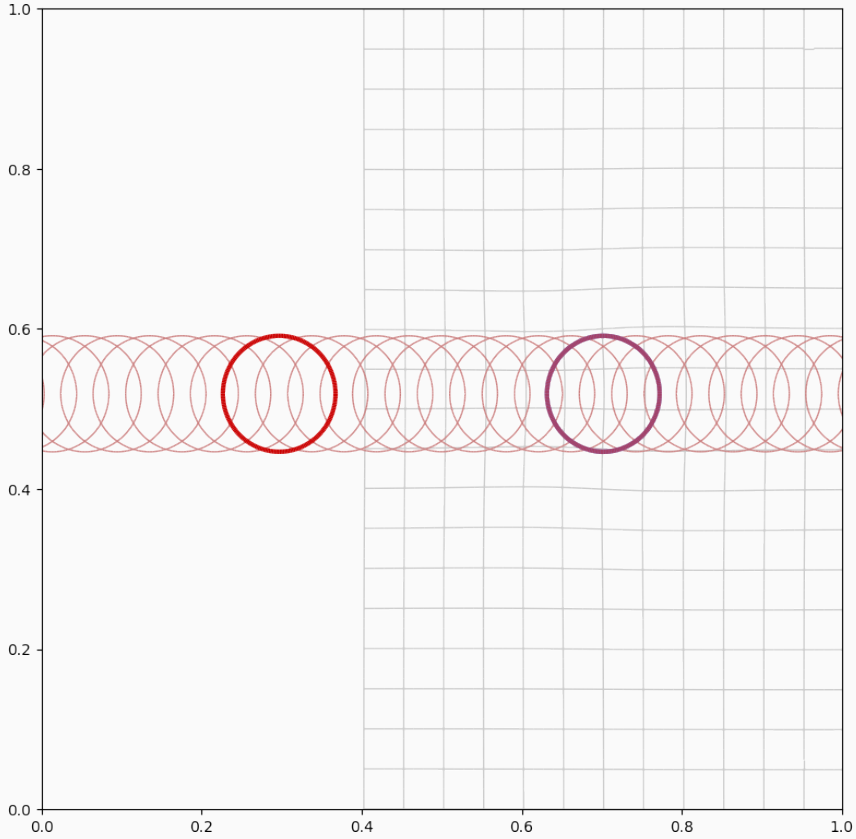
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



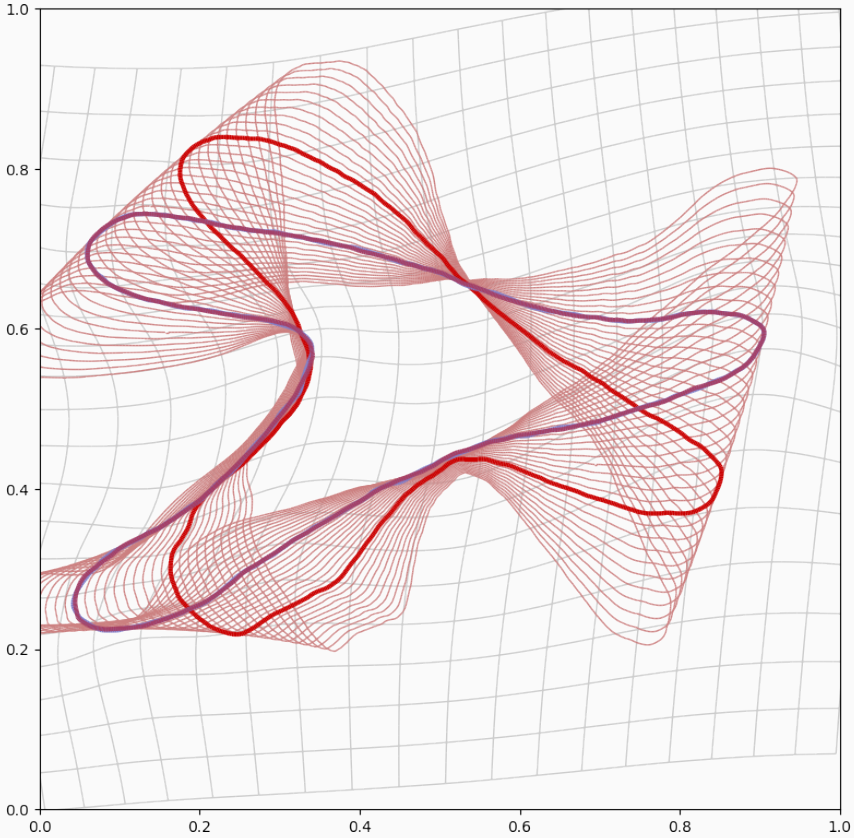
It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



It works!

```
k = { "id" : Kernel("cauchy(x,y)"), "gamma" : 1/.1**2 }
def H(q,p) :
  l = Lambda(q, mu) # nits=2...
  return .5* dot( l*p, kernel_product( k, q, q, l*p) )
```



Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;

Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;

Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?

Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?
- is an idiomatic way of **flagging influential parts** of the shape, through a **mask of weights** μ_i.

Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?
- is an idiomatic way of **flagging influential parts** of the shape, through a **mask of weights** μ_i.

Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?
- is an idiomatic way of **flagging influential parts** of the shape, through a **mask of weights** μ_i.

Branched OT $\longleftarrow$ LDDMM $\longrightarrow$ Translations, $\|v\|^2$

Automatic differentiation: develop models at will

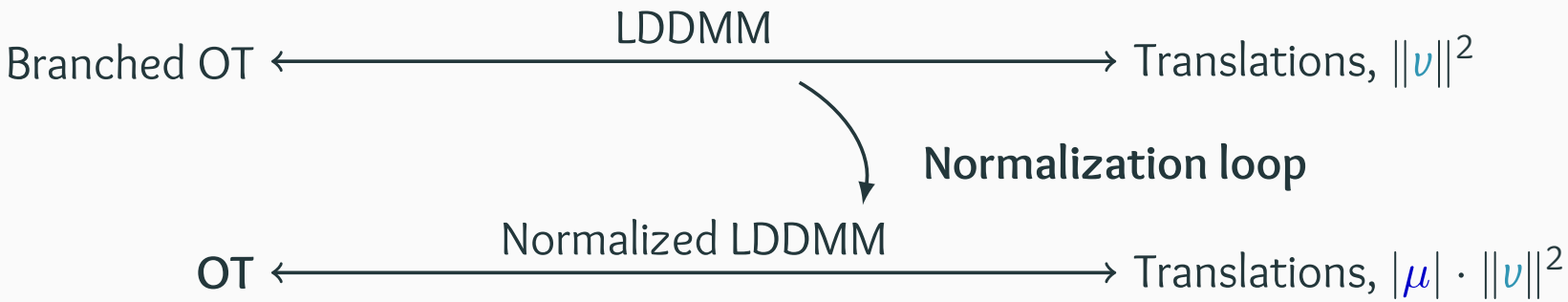
Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?
- is an idiomatic way of **flagging influential parts** of the shape, through a **mask of weights** μ_i.

Automatic differentiation: develop models at will

Using this normalized cometric:

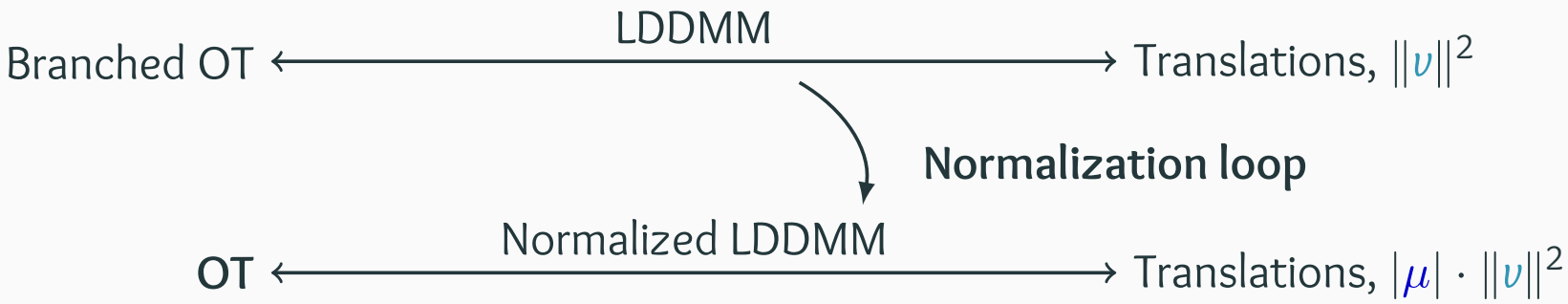
- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?
- is an idiomatic way of **flagging influential parts** of the shape, through a **mask of weights** μ_i.



Automatic differentiation: develop models at will

Using this normalized cometric:

- is tractable thanks to the **PyTorch+KeOps** combo;
- is about **4 times** as expensive as LDDMM;
- could be implemented on **images** – using `mermaid`?
- is an idiomatic way of **flagging influential parts** of the shape, through a **mask of weights** μ_i.



$\Longrightarrow$ What about **learned** cometrics?

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

What I still have to show you:

- **Matlab**, R, numpy bindings.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

What I still have to show you:

- **Matlab**, R, numpy bindings.
- x2-x4 **performance** improvements come September.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

What I still have to show you:

- **Matlab**, R, numpy bindings.
- x2-x4 **performance** improvements come September.
- **LogSumExp**, Max, Min reductions.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

What I still have to show you:

- **Matlab**, R, numpy bindings.
- x2-x4 **performance** improvements come September.
- **LogSumExp**, Max, Min reductions.
- Cheap fidelities that **do not saturate** at long range.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

What I still have to show you:

- **Matlab**, R, numpy bindings.
- x2-x4 **performance** improvements come September.
- **LogSumExp**, Max, Min reductions.
- Cheap fidelities that **do not saturate** at long range.

Conclusion

- PyTorch $\longrightarrow$ Focus on your **models**.
- KeOps $\longrightarrow$ **Scale up** to 3D datasets.

What I still have to show you:

- **Matlab**, R, numpy bindings.
- x2-x4 **performance** improvements come September.
- **LogSumExp**, Max, Min reductions.
- Cheap fidelities that **do not saturate** at long range.

$\Longrightarrow$ See you soon?

Thank you!

Any questions?

References i

Line Kühnel, Alexis Arnaudon, and Stefan Sommer.
Differential geometry and stochastic dynamics with deep learning numerics.
arXiv preprint arXiv:1712.08364, 2017.

Philip A Knight, Daniel Ruiz, and Bora Uçar.
A symmetry preserving algorithm for matrix scaling.
SIAM journal on Matrix Analysis and Applications, 35(3):931–955, 2014.

Mario Micheli, Peter W Michor, and David Mumford.
Sectional curvature in terms of the cometric, with applications to the riemannian manifolds of landmarks.
SIAM Journal on Imaging Sciences, 5(1):394–433, 2012.

References ii


Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer.
Automatic differentiation in pytorch.
2017.