Seamless Kernel Operations on GPU, with auto-differentiation and without memory overflows

B. Charlier (IMAG - Univ Montpellier)
J. Feydy (Imperial College, London)
J. Glaunès (MAP5 - Univ Paris Descartes)
F.D. Collin (CNRS - IMAG - Univ Montpellier)
G. Durif (CNRS - IMAG - Univ Montpellier)

January 7th, 2020 – SciDoLyse meeting, Lyon

http://www.kernel-operations.io/
1. Introduction

2. Matrix reduction and kernel operations

3. Computation on GPU

4. Implementation

5. Using KeOps

6. Autodiff 101

7. Inverse of SPD matrix with pyKeOps
Introduction
What KeOps can do?

• Compute **generic reductions** of very large arrays/matrices

\[ \sum_{i=1}^{M} a_{ij} \quad \text{or} \quad \sum_{j=1}^{N} a_{ij} \]

for some large matrix \( A = [a_{ij}] \in \mathbb{R}^{M \times N} \)

• Compute **kernel** dot products and the associated gradients

\[ \sum_{i=1}^{M} K(x_i, y_j) \quad \text{or} \quad \sum_{j=1}^{N} K(x_i, y_j) \]

for a kernel function \( K \) and some vectors \( x_i, y_j \in \mathbb{R}^{D} \)
What KeOps can do?

- Compute generic reductions of very large arrays/matrices
- Compute kernel dot products and the associated gradients
- Use cases involve large $M$ and $N$ (typically from $\approx 10^3$ to $10^7$). Two pitfalls:
  - computation times (complexity is $O(MN)$)
  - memory overflow (storage should be $O(M)$ or $O(M)$)
Kernel spaces in statistics and Learning

- Kernel density estimation:
- Classification/Regression: SVM, K-NN, etc...
- Kernel embeddings to compare distribution:
- Interpolation and Kriging
- Optimal Transport
Motivations

• GPU user-friendly computing: development effort oriented for deep learning
  → PyTorch or TensorFlow provide GPU implementation of common operations, together with automatic differentiation.

• GPU computing can be used for general purpose computations, not only neural networks
  → Generic codes to use GPU computing require low-level tools (CUDA, OpenCL)

• Needs: provide an effortless tool for GPU computing (application: statistics, machine learning and more)
Matrix reduction and kernel operations
Matrix reduction

- Simple row or column-wise matrix reduction

\[
\begin{bmatrix}
\sum_{i=1}^{M} a_{ij}
\end{bmatrix} \in \mathbb{R}^N \quad \text{or} \quad \begin{bmatrix}
\sum_{j=1}^{N} a_{ij}
\end{bmatrix} \in \mathbb{R}^M
\]

for a matrix \( A = [a_{ij}] \in \mathbb{R}^{M \times N} \)

- Vector/matrix or matrix/matrix product

\[
\begin{bmatrix}
\sum_{i=1}^{M} a_{ij} \beta_j
\end{bmatrix} \in \mathbb{R}^N \quad \text{or} \quad \begin{bmatrix}
\sum_{j=1}^{N} a_{ij} \beta_j
\end{bmatrix} \in \mathbb{R}^M
\]

for a matrix \( A = [a_{ij}] \in \mathbb{R}^{M \times N} \) and a vector \( \beta = [\beta_j] \in \mathbb{R}^N \)
Matrix reduction

\[ \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix}_{M \times N} \]

\[ \downarrow \]

\[ \begin{bmatrix} \cdots & \cdots & \sum_{i=1}^{M} a_{ij} & \cdots & \cdots \end{bmatrix}_{1 \times N} \]
Matrix reduction

\[
\begin{bmatrix}
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\end{bmatrix}_{M \times N} \rightarrow
\begin{bmatrix}
\vdots \\
\vdots \\
\sum_{j=1}^{N} a_{ij} \\
\vdots \\
\vdots \\
\end{bmatrix}_{M \times 1}
\]
Kernel operator

Considering some data vector $\mathbf{x}_i$ and $\mathbf{y}_j$ in $\mathbb{R}^D$

- (Intuitively) a **kernel** function is an application $K : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}$

$$\ (\mathbf{x}_i, \mathbf{y}_i) \mapsto K(\mathbf{x}_i, \mathbf{y}_i)$$

...corresponding to a **scalar product** between $\mathbf{x}_i$ and $\mathbf{y}_j$ in a different space

- Example

  - **Linear kernel:**
    $$K(\mathbf{x}_i, \mathbf{y}_j) = \langle \mathbf{x}_i, \mathbf{y}_j \rangle = \mathbf{x}_i^T \mathbf{y}_j = \sum_{k=1}^{D} x_{ik} y_{jk}$$

  - **Gaussian kernel:**
    $$K(\mathbf{x}_i, \mathbf{y}_j) = \exp \left( -\frac{1}{2\sigma^2} \| \mathbf{x}_i - \mathbf{y}_j \|_2^2 \right)$$
Kernel reduction

• Convolution-like operation

\[
\sum_{i=1}^{M} K(x_i, y_j) \beta_j \quad \in \mathbb{R}^N \quad \text{or} \quad \sum_{j=1}^{N} K(x_i, y_j) \beta_j \quad \in \mathbb{R}^M
\]

for some \( D \)-vectors \((x_i)_{i=1,\ldots,M} \in \mathbb{R}^{M \times D}, (y_j)_{j=1,\ldots,N} \in \mathbb{R}^{N \times D}\) and \(\beta = [\beta_j] \in \mathbb{R}^N\)

\(\rightarrow\) Row-wise or column-wise reduction on the matrix \(K = \begin{bmatrix} K(x_i, y_j) \end{bmatrix} \in \mathbb{R}^{M \times N}\)
Kernel reduction

\[
\begin{bmatrix}
    \times & \times & \times & \times & \times & \times \\
    \times & \times & \times & \times & \times & \times \\
    \times & \times & \times & \times & \times & \times \\
    \times & \times & \times & \times & \times & \times \\
    \times & \times & \times & \times & \times & \times \\
    \times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\]

\[
\sum_{i=1}^{M} K(x_i, y_j)
\]

\[
\begin{bmatrix}
    \ldots & \ldots & \sum_{i=1}^{M} K(x_i, y_j) & \ldots & \ldots \\
\end{bmatrix}
\]

\[
M \times N \quad 1 \times N
\]
Kernel reduction

\[
\begin{bmatrix}
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\]

\[\sum_{j=1}^{N} K(x_i, y_j)\]
Kernel reduction

• More complex operation

\[ \sum_{i=1}^{M} K_1(x_i, y_j) K_2(u_i, v_j) \langle \alpha_i ; \beta_j \rangle \quad \text{or} \quad \sum_{j=1}^{N} K_1(x_i, y_j) K_2(u_i, v_j) \langle \alpha_i ; \beta_j \rangle \]

for some kernel \( K_1 \) and \( K_2 \), and some \( D \)-vectors \((x_i)_i, (u_i)_i, (\alpha_i)_i \in \mathbb{R}^{M \times D}\) and \((y_j)_j, (v_j)_j, (\beta_j)_j \in \mathbb{R}^{N \times D}\).
Generic reduction in KeOps

\[ 1 \leq i \leq N \text{ et } 1 \leq j \leq M \text{ avec } N,M \approx 10^4 \text{ ou } 10^6 \]

- A generic case:
  \[
  \sum_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \in \mathbb{R}^M
  \]
  \[i=1,\ldots,M\]

- ...an even more generic case:
  \[
  \bigstar_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \in \mathbb{R}^M
  \]
  \[i=1,\ldots,M\]

where \(\bigstar\) can be any reduction (sum, max, min, etc.) over a dimension
Why GPU computing

• Matrix/kernel reduction = combination of generic matrix operations

• GPU are good for matrix computations

• **Problem:** the matrix $K = \left[ K(x_i, y_j) \right] \in \mathbb{R}^{M \times N}$ is very large ($M, N \approx 10^4$ ou $10^6$)

  → how to store it in memory

  → how to iterate through rows/columns to do computations
Computation on GPU
Target:

- Gamers: 1000 euros
- Scientific computing: 3000 – 9000 euros

Under the hood: similar chipsets with few enhancements (ECC, float64,...)
GPU = massively parallel architecture

A GPU architecture

→ scalable array of multithreaded Streaming Multiprocessors (SMs)

→ each single processor (called a thread) is able to execute an independent set of instructions.

1000’s of cores inside a single GPU

(multi-CPU architecture = at best 10’s – 100’s of cores)
MatMult: A first naive implementation

\[ A \in \mathbb{R}^{M \times N} \text{ and } B \in \mathbb{R}^{N \times D} \]

A matrix multiplication

\[ AB = \left[ \sum_k a_{ik} b_{kj} \right]_{M \times D} \]

→ a set of \( N \times D \) scalar products

Parallel computing

• each thread computes \( D \) scalar products, i.e. \( \langle a_{i.}, b_{.j} \rangle \) for all \( j \)
MatMult: A first naive implementation

A ∈ \( \mathbb{R}^{M \times N} \) and B ∈ \( \mathbb{R}^{N \times D} \)

Thread \( i \) needs to access

- row \( a_i \in \mathbb{R}^N \)
- all columns \( (b_j)_{j=1,...,D} \) i.e. the full matrix B

→ potential memory overflow

→ no mutual memory access between threads
MatMult: A first naive implementation

\[ \mathbf{A} \in \mathbb{R}^{M \times N} \text{ and } \mathbf{B} \in \mathbb{R}^{N \times D} \]

Assign a block of rows \( i \) to a thread

- share the memory access to each \( b_{j} \) to compute all rows \( i \) in the block

\[ \langle a_{i}, b_{j} \rangle \]

→ each thread still requires to access the full matrix \( \mathbf{B} \) to finish the computations for a row \( i \)
Memory management on GPU

- Data initially stored on the host (in RAM) should be transferred to the device (GPU) to be treated (bottleneck).

- Different kinds of memory: global vs shared memory.

Smart use of the shared memory: key to provide an efficient code in term of computational time.

→ less transfer between global and thread memory (shared mem and/or register).
MatMult: Tiled implementation (decomposition with block sub-matrix product)

- Tasks (scanning rows $a_i$) divided into tiles
- All thread use the shared memory within a block
  → a single memory transferred of each tile in $B$ for all threads
MatMult: Tiled implementation (decomposition with block sub-matrix product)

- Tasks (scanning rows $a_i$) divided into tiles
- All thread use the shared memory within a block
  - → a single memory transferred of each tile in $B$ for all threads
MatMult: Tiled implementation (decomposition with block sub-matrix product)

- Block $a_i$
- Tile $b_{j}$

$\langle a_i, b_{j} \rangle$ load in shared mem

- Accumulation (addition of the intermediate results) when scanning tiles across $A$
Benchmark I

Runtimes for Gaussian Matrix-Vector products in dimension 3

Seconds

Number of samples

Memory overflow!
Implementation
Coding generic formulas with KeOps

- **Mathematical formula** with two vectors \( x, y \in \mathbb{R}^D \):
  \[
  (x, y) \mapsto \exp \left( \langle x, y \rangle \right)
  \]
- A formula \( F \) in KeOps is first encoded as a string using combinations of elementary operations
  
  "Exp(Scalprod(x,y))"

- Then it is expanded internally in the C++ code using templates:
  
  \[
  F=\text{Exp}\langle\text{Scalprod}\langle X, Y \rangle\rangle
  \]

\[\rightarrow\] A formula is an instantiation of a variadic recursively defined templated class

\[\rightarrow\] KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)
Coding generic formulas with KeOps

- Mathematical formula with two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$:
  \[(\mathbf{x}, \mathbf{y}) \mapsto \exp (\langle \mathbf{x}, \mathbf{y} \rangle)\]

- A formula $F$ in KeOps is first **encoded as a string** using combinations of elementary operations

  "Exp(Scalprod(x,y))"

- Then it is expanded internally in the C++ code using templates:

  $F=\text{Exp}<\text{Scalprod}<X,Y>>$

→ A formula is an instantiation of a variadic recursively defined templated class
→ KeOps is able to generate shared objects that compute on a GPU
  (compilation on the fly)
Coding generic formulas with KeOps

- Mathematical formula with two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$:
  \[(\mathbf{x}, \mathbf{y}) \mapsto \exp(\langle \mathbf{x}, \mathbf{y} \rangle)\]

- A formula $F$ in KeOps is first encoded as a string using combinations of elementary operations
  
  "Exp(Scalprod(x,y))"

- Then it is expanded internally in the C++ code using templates:
  
  $F=\text{Exp<Scalprod<X,Y>>}$

→ A formula is an instantiation of a variadic recursively defined templated class

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)
Coding generic formulas with KeOps

• Mathematical formula with two vectors $x, y \in \mathbb{R}^D$:
  $$(x, y) \mapsto \exp \left( \langle x, y \rangle \right)$$

• A formula $F$ in KeOps is first encoded as a string using combinations of elementary operations
  
  
  "Exp(Scalprod(x,y))"

• Then it is expanded internally in the C++ code using templates:
  
  $$F=\text{Exp}<\text{Scalprod}<X,Y>>$$

→ A formula is an instantiation of a variadic recursively defined templated class

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)
Coding generic formulas with KeOps

- Mathematical formula with two vectors $x, y \in \mathbb{R}^D$:
  \[(x, y) \mapsto \exp\left(\langle x, y \rangle\right)\]

- A formula $F$ in KeOps is first encoded as a string using combinations of elementary operations
  
  "Exp(Scalprod(x,y))"

- Then it is expanded internally in the C++ code using templates:
  
  $F = \text{Exp} < \text{Scalprod} < X, Y >>$

→ A formula is an instantiation of a variadic recursively defined templated class

→ KeOps is able to generate shared objects that compute on a GPU
  (compilation on the fly)
Under the hood: C++ encoding of maths operations in KeOps I

A typical KeOps (unary) operation is a struct that looks like:

```cpp
template<class F>
struct Exp : UnaryOp<Exp, F> {
  // dimension of the output
  static const int DIM = F::DIM;
}
```

Implementation of the operation:

```cpp
/// inlined function in the final cuda code
static DEVICE INLINE void Operation(TYPE *out, TYPE *in) {
  #pragma unroll
  for (int k = 0; k < DIM; k++) { out[k] = exp(in[k]); }
}
```
Gradient computation:

```
/////////////// Autodiff!
    template<class V, class GRADIN>
    using DiffT = typename F::template DiffT<V, Mult<Exp<F>, GRADIN>>;
```

String encoding:

```
/////////// Macro providing high level syntax
#define Exp(f) Exp<decltype(f)>()
```
A typical KeOps (binary) operation is a `struct` that looks like:

```cpp
template < class FA, class FB >
struct Add : BinaryOp< Add, FA, FB > {
    // dimension of the output ... and (compile time) checks
    static const int DIM = FA::DIM; // Output dim = FA::DIM = FB::DIM
    static_assert(DIM == FB::DIM, "Dimensions must be the same for Add");
}
```

Implementation of the operation:

```cpp
static DEVICE INLINE void Operation(TYPE *out, TYPE *inA, TYPE *inB) {
    for (int k = 0; k < DIM; k++) {out[k] = inA[k] + inB[k];}
}
```
Gradient computation:

```cpp
//////////////// Autodiff!

template < class V, class GRADIN >
using DiffT = Add< typename FA::template DiffT< V, GRADIN >,
                  typename FB::template DiffT< V, GRADIN > >;
```

Simplification rule:

```cpp
//////////////// Simplification rules: e.g.

template < class F >
struct Add_Alias0< F, F > { using type = Scal< IntConstant< 2 >, F >; };
```
KeOps proposes a wide range of elementary operations

- **Vector operations**: scalar product, norm, distance, normalization, vector/vector element-wise operation ($+, -, *, /$), tensors reduction, etc...
- **Elementary $\mathbb{R} \rightarrow \mathbb{R}$ functions**: exp, log, inverse, abs, pow, sqrt, sin, cos, etc...
- **Simple matrix operations**: matrix product, tensor product (in Python), etc...
- **Matrix reduction**: sum, min, max, argmin, argmax, etc...

→ a formula = a combination of these operations
Using KeOps
Kernel Operations on the GPU, with autodiff, without memory overflows

The KeOps library lets you compute generic reductions of large 2D arrays whose entries are given by a mathematical formula. It combines a tiled reduction scheme with an automatic differentiation engine, and can be used through Matlab, NumPy or PyTorch backends. It is perfectly suited to the computation of Kernel dot products and the associated gradients, even when the full kernel matrix does not fit into the GPU memory.

Using the PyTorch backend, a typical sample of code looks like:

```python
import torch
from pykeops.torch import KeOps

# A kernel density estimator between point clouds a R^3
my_kernel = KeOps.Exp([-PyOp(\sqrt{a})^{2}]/2)

# Work with respect to '1', result divided by '1'

# Apply it to 2D arrays x and y with 3 columns and a (huge) number of rows
x = torch.randn(100000, 3, requires_grad=True).cuda()
y = torch.randn(200000, 2, requires_grad=True).cuda()
x_i = x.unsqueeze(1) # Add one batch dimension
y_j = y.unsqueeze(0) # Repeat the other batch
x_i = x_i.expand(y_j.size(0), x_i.size(1), x_i.size(2)) # Broadcast 3 dimensions
y_j = y_j.expand(x_i.size(0), y_j.size(1), y_j.size(2)) # Broadcast 3 dimensions

# KeOps supports autodiff!
```

KeOps allows you to leverage your GPU without compromising on usability. It provides:
- Linear (instead of quadratic) memory footprint for Kernel operations.
- Support for a wide range of mathematical formulas.
KeOps stack

- Dependencies: **Cmake** (≥3.10), **C++ compiler**\(^1\) (g++ ≥ 7 or clang) or **cuda** compiler (nvcc ≥10) and CUDA libs (for GPU computing)

- Open source (MIT licence): [github.com/getkeops/keops](https://github.com/getkeops/keops)

- Continuous integration (tested on linux distros and MacOs): Jenkins at [ci.inria.fr](http://ci.inria.fr)


\(^1\)for CPU computing
KeOps user interface

- PyKeOps: **Python** *(numpy and pytorch)*
- KeOpsLab: **Matlab**
- RKeOps: **R**
- **C++** API
Example in Python: single Gaussian convolution

We want to compute

$$\gamma_i = \sum_{j=1}^{N} \exp \left( -s \|x_i - y_j\|_2^2 \right) b_j$$

with $s \in \mathbb{R}$, $[x_i]_{i=1,...,M} \in \mathbb{R}^{M \times 3}$, $[y_j]_{j=1,...,N} \in \mathbb{R}^{N \times 3}$ and $[b_j]_{j=1,...,N} \in \mathbb{R}^{N \times 6}$
Example in Python: single Gaussian convolution

From Python using Numpy (similar in R or Matlab)

```python
from pykeops.numpy import Genred

## compilation on the fly (user-friendly syntax)
my_conv = Genred(
    formula='Sum_Reduction(Exp(-s * SqNorm2(x - y)) * b, 0)',
    aliases=['s = Pm(1)',  # parameter (scalar)
             'x = Vi(3)',   # vector indexed by i (of dim 3)
             'y = Vj(3)',   # vector indexed by j (of dim 3)
             'b = Vj(6)'],  # vector indexed by j (of dim 6)
    dtype='float32')

# assuming s, x, y and b are Numpy arrays (data and parameter values)

## compute directly on the GPU
gamma = my_conv(s, x, y, b)
```

# assuming s, x, y and b are Numpy arrays (data and parameter values)

## compute directly on the GPU
gamma = my_conv(s, x, y, b)
Mathematical formula (standard Gaussian kernel)

\[ \gamma_i = \sum_{j=1}^{N} \exp \left( \| x_i - y_j \|_2^2 \right) \]

with \([x_i]_{i=1,...,M} \in \mathbb{R}^{M \times 3}, [y_j]_{j=1,...,N} \in \mathbb{R}^{N \times 3}\)
Create two arrays with 3 columns and a (huge) number of lines, on the GPU

```python
import torch
x = torch.randn(1000000, 3, requires_grad=True).cuda()
y = torch.randn(2000000, 3).cuda()
```

Use a decorator to turn tensors into KeOps symbolic variables:

```python
from pykeops.torch import LazyTensor
x_i = LazyTensor(x[:, None, :])  # x_i.shape = (1e6, 1, 3)
y_j = LazyTensor(y[None, :, :])  # y_j.shape = (1, 2e6, 3)
```
Example in Python (LazyTensor)

Perform symbolic large-scale computations

```python
# Symbolic (1e6,2e6,1) matrix of squared distances
D_ij = ((x_i - y_j)**2).sum(dim=2)

# Symbolic (1e6,2e6,1) Gaussian kernel matrix
K_ij = (- D_ij).exp()
```

Get the result (computations on GPU are done here)

```python
# Genuine torch.cuda.FloatTensor
a_i = K_ij.sum(dim=1)  # a_i.shape = (1e6, 1)

## KeOps supports autograd!
g_x = torch.autograd.grad((a_i ** 2).sum(), [x])
```
More features

- **PyKeOps (Numpy), KeOpsLab, Rkeops**: formula gradient computation

- **PyKeOps (PyTorch)**: automatic differentiation engine (compatible with PyTorch autograd)

- possible to add new generic operations upon request (**responsive user support via Github issues**)

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

$$
\nabla F(x_0) = \begin{pmatrix}
\frac{\partial}{\partial x_1} F(x_0) \\
\frac{\partial}{\partial x_2} F(x_0) \\
\vdots \\
\frac{\partial}{\partial x_n} F(x_0)
\end{pmatrix} \approx \frac{1}{\delta t} \begin{pmatrix}
F(x_0 + \delta t \cdot (1, 0, \ldots, 0)) - F(x_0) \\
F(x_0 + \delta t \cdot (0, 1, \ldots, 0)) - F(x_0) \\
\vdots \\
F(x_0 + \delta t \cdot (0, 0, \ldots, 1)) - F(x_0)
\end{pmatrix}.
$$
Let $F : \mathbb{R}^n \to \mathbb{R}$ be a smooth function. Then:

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

$\implies$ costs $(N+1)$ evaluations of $F$, which is poor.
Reverse AD

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

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

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

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

- the adjoint of the differential is $(d_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$. 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,

$$M_{\partial_x F(x_0)} = \begin{pmatrix} \partial_{x_1} F(x_0) \\ \partial_{x_2} F(x_0) \\ \vdots \\ \partial_{x_n} F(x_0) \end{pmatrix} = \nabla_x F(x_0)$$
Reverse AD = backpropagating = chain rules

Backpropagating through a computational graph requires:

$$F_i : E_{i-1} \rightarrow E_i$$

$$x \mapsto F_i(x)$$

encoded as computer programs.
Reverse AD = backpropagating = chain rules

\[ \frac{\partial F(x_0)}{\partial x} \cdot 1 \]

\[ x^* \]

\[ m \]

\[ \mathbb{R}^n \]

\[ E_0 \]

\[ x_0 \]

\[ F_1 \]

\[ x_1 \]

\[ E_1 \]

\[ \cdots \]

\[ \cdots \]

\[ \cdots \]

\[ E_p \]

\[ x_p \]

\[ \mathbb{R}^m \]

\[ \text{Backpropagating} \] through a computational graph requires:

\[ F_i : E_{i-1} \rightarrow E_i \]

\[ \partial_x F_i : E_{i-1} \times E_i \rightarrow E_{i-1} \]

\[ x \leftrightarrow F_i(x) \]

\[ (x_{i-1}, x_i^*) \leftrightarrow \partial_x F_i(x_{i-1}) \cdot x_i^* \]

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, $x_p = F(x_0) \in \mathbb{R}$. 

[Diagram of reverse AD process]
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

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

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

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

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

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

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

where

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

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

\[
\begin{align*}
x & = \text{torch.rand}(4, 1, \text{requires_grad=True}) \\
y & = \text{torch.rand}(3, 1, \text{requires_grad=True}) \\
\text{aliases} & = ["x=Vi(1)", "y=Vj(1)"] \\
\text{formula} & = "\text{Exp}(x+y)" \\
F & = \text{pykeops.torch.Genred(formula, aliases, axis=1)(x,y)} \\
\text{torch.allclose}\left(\text{torch.exp}(x + y.t()).\text{sum}(1), F.\text{view}(-1))\right) & \text{# True}
\end{align*}
\]
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:

```python
e = torch.rand_like(x)

aliases_grad = ['x=Vi(1)', 'y=Vj(1)', 'e=Vi(1)']

formula_grad = "Grad(Exp(x+y), y, e)"

F_grad = pykeops.torch.Genred(formula_grad, aliases_grad, axis=0)(x, y, e)

torch.allclose(torch.autograd.grad(F, [y], e)[0], F_grad)
# True

torch.allclose(torch.exp(x + y.t()).t() @ e, F_grad)
# True
```
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)
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)
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 $\nu = (\nu_t)_{t \in [0, 1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let
  $\phi : [0, 1] \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$:

  $$
  \begin{cases}
  \dot{\phi}_t(x) = \nu_t(\phi_t(x)) \\
  \phi_0(x) = x.
  \end{cases}
  $$

  $t \in [0, 1]$ and $x \in \mathbb{R}^3$
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 \( \phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3 \):

\[
\begin{cases}
  \dot{\phi}_t(x) = v_t(\phi_t(x)) \\
  \phi_0(x) = x.
\end{cases}
\]

\( t \in [0,1] \) and \( x \in \mathbb{R}^3 \)
Motivation: LDDMM

Deformation = flow of time varying smooth vector field

- **Flow:** let $\nu = (\nu_t)_{t \in [0,1]} \in \mathcal{V}$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0, 1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

\[
\begin{aligned}
\dot{\phi}_t(x) &= \nu_t(\phi_t(x)) \\
\phi_0(x) &= x.
\end{aligned}
\]

$t \in [0, 1]$ and $x \in \mathbb{R}^3$
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 \( \phi : [0, 1] \times \mathbb{R}^3 \to \mathbb{R}^3 \):

\[
\begin{align*}
    \dot{\phi}_t(x) &= v_t(\phi_t(x)) \\
    \phi_0(x) &= x.
\end{align*}
\]

\( t \in [0, 1] \) 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 \( \phi : [0,1] \times \mathbb{R}^3 \rightarrow \mathbb{R}^3 \):

\[
\begin{aligned}
\dot{\phi}_t(x) &= v_t(\phi_t(x)) \\
\phi_0(x) &= x.
\end{aligned}
\]

\( t \in [0,1] \) and \( x \in \mathbb{R}^3 \)
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 $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases}
\phi_t(x) = v_t(\phi_t(x)) \\
\phi_0(x) = x.
\end{cases}$$

$t \in [0,1]$ and $x \in \mathbb{R}^3$
Soit $p, q \in \mathbb{R}^{N \times D}$, on cherche à calculer

$$H(q, p) = p^t K_{q,q} p = \sum_i \sum_j p_i^t K_{s}(q_i, q_j) p_j$$

exp($-||q_i - q_j||^2 / s$)

peut être interprété comme une énergie cinétique ou une norme d’un RKHS.
Computing an Hamiltonian

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

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

peut être interprété comme une énergie cinétique ou une norme d’un RKHS.

```python
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
```

```
K_qq = torch.exp(-sqd / (s**2)) # Gaussian kernel

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

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

```
# Automatic differentiation is straightforward
[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)
```
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  # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H = 0.5 * torch.dot(p.view(-1), v.view(-1))

# Automatic differentiation is straightforward
[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)
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  # matrix 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  # matrix 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  # matrix 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.)
# 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 # matrix mult. (N,N)@(N,D) = (N,D)

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

# Automatic differentiation is straightforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)

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

```python
def compute_hamiltonian(q, p, s):
    N, D = q.shape
    K_qq = torch.exp(-torch.sum((q - q.unsqueeze(0)) ** 2, 2) / (s ** 2))  # Gaussian kernel
    v = K_qq @ p  # matrix mult.
    H = .5 * torch.dot(p.view(-1), v.view(-1))
    return H
```

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

# 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.)
import pykeops

# Compute the kernel convolution with keops
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")

# Then, compute the Hamiltonian $H(q,p)$: $0.5 \times \langle p, v \rangle$
H = 0.5 * torch.dot( p.view(-1), v.view(-1) )

# Automatic differentiation works
[dq, dp] = torch.autograd.grad( H, [q, p], 1.)
Inverse of SPD matrix with pyKeOps
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

$$K a = b$$

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

\[
(K + \alpha \text{Id})a = b \iff a = (K + \alpha \text{Id})^{-1} b.
\]

...which is not a reduction.
Let $K_\alpha = K + \alpha I$. We may consider the quadratic problem

$$\arg\min_a \frac{1}{2} a^t K_\alpha a - a^t b$$

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 $K_\alpha$
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

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

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

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

```python
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)
```
Let $V$ be a RKHS with radial Gaussian Kernel. We want to solve

$$\arg\min_{\mathbf{v} \in \mathbf{V}} \|\mathbf{v}\|_{\mathbf{V}}^2 + \frac{1}{\alpha} \|\mathbf{v}(x_i) - b_i\|_2^2$$

```
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)
x = 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)
Using Scipy LinAlg routines

Create a toy dataset:

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

Turn it into a KeOps `LazyTensor`:

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

Using a `sum` reduction we may compute a Gaussian convolution...
...instead, $K_{xx}$ can be directly understood as a `LinearOperator`:

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

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

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

![Graph showing runtimes for inverse radial kernel matrix in dimension 3](image)

- **backend = "NumPy"**
- **backend = "PyTorch"**
- **backend = "NumPy + KeOps"**
- **backend = "PyTorch + KeOps"**
- **backend = "Scipy + KeOps"**

Legend:
- Too slow
- Memory overflow

**x-axis:** Number of samples
**y-axis:** Seconds
Conclusion
KeOps:
Seamless Kernel Operations...
  → write formulas with simple matrix 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/keops/index.html

https://github.com/getkeops/keops