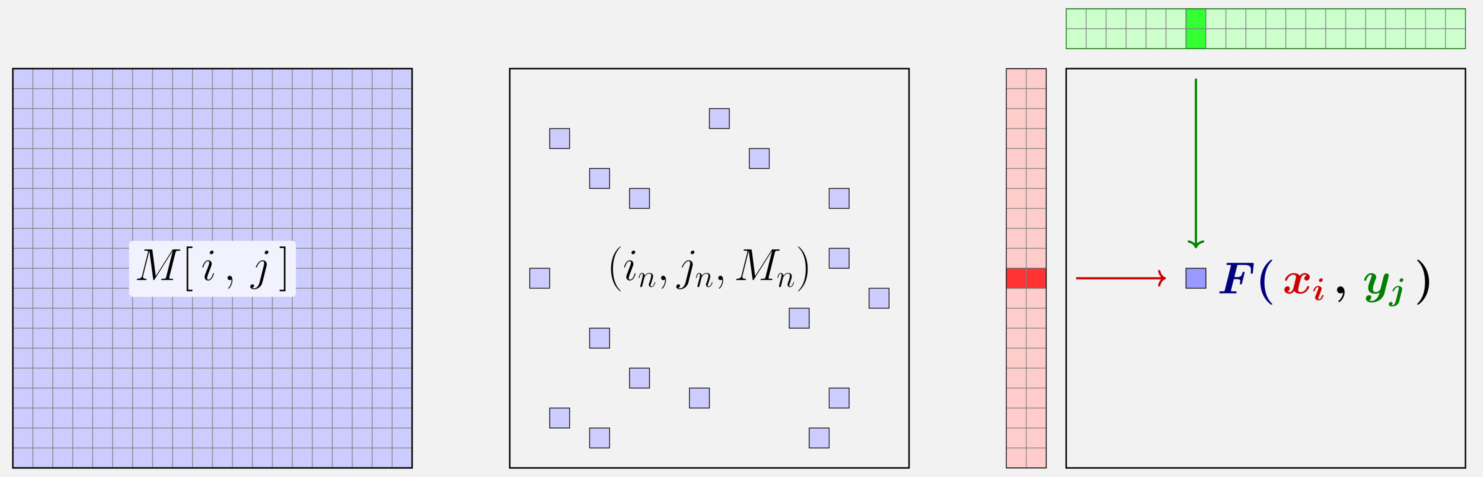
Fast geometric learning with symbolic matrices – www.kernel-operations.io

Jean Feydy[1,*] Joan A. Glaunès[2,*] Benjamin Charlier[3,*] Michael M. Bronstein[1,4]

[1]Imperial College London [2]Université de Paris [3]Université de Montpellier [4]Twitter [*]Equal contribution

1 Symbolic matrices



(a) **Dense matrix** Coefficients only (b) **Sparse matrix** Coordinates + coeffs (c) **Symbolic matrix** Formula + data

We can represent tensors as:

(a) **Dense** matrices – large, contiguous **arrays** of numbers:
- \+ This is a **convenient** and well supported format.
- – It puts a heavy load on the **memories** of our GPUs, with **time-consuming transfers** between layers of CUDA **registers**.

(b) **Sparse** matrices – if they have **few non-zero entries**:
- \+ We encode **large tensors** with a small memory footprint.
- – Outside of **graph** processing, few objects are **sparse enough** to really benefit from this representation.

(c) **Symbolic** matrices – if their coefficients $M_{i,j}$ are given by a **formula** F that is evaluated on vectors "x_i" and "y_j". Think of **distance** and **kernel** matrices, **point** convolutions, **attention** layers, etc.:
- \+ **Linear** memory usage: no more **memory** overflows.
- \+ We can optimize the use of registers for a $\times$**10** - $\times$**100 speed-up** vs. a standard PyTorch GPU baseline.

Our **KeOps library** provides support for **symbolic matrices** on CPUs and GPUs. Under the hood, it combines an optimized **C++** engine with high-level binders for **PyTorch**, **NumPy**, Matlab and R – thanks to Ghislain Durif. We welcome **contributors** for JAX, Julia and other frameworks!

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

2 An extension for PyTorch, NumPy, etc.

Our KeOps library comes with all the perks of a deep learning toolbox:
- \+ A transparent **array-like** interface.
- \+ Full support for automatic **differentiation**.
- \+ A comprehensive collection of **tutorials**, available online.

We support arbitrary **formulas** and **variables** with a wide range of:
- **Reductions:** sum, log-sum-exp, K-min, matrix-vector product, etc.
- **Operations:** $+$, $\times$, sqrt, exp, neural networks, etc.
- **Advanced schemes:** batch processing, block sparsity, etc.

Here is how to perform a fast **nearest neighbor search**:

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

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

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

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

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

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

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

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

The line above is **just as fast** as the bruteforce ("Flat") CUDA scheme of the **FAISS** library... And can be used with **any metric**!

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

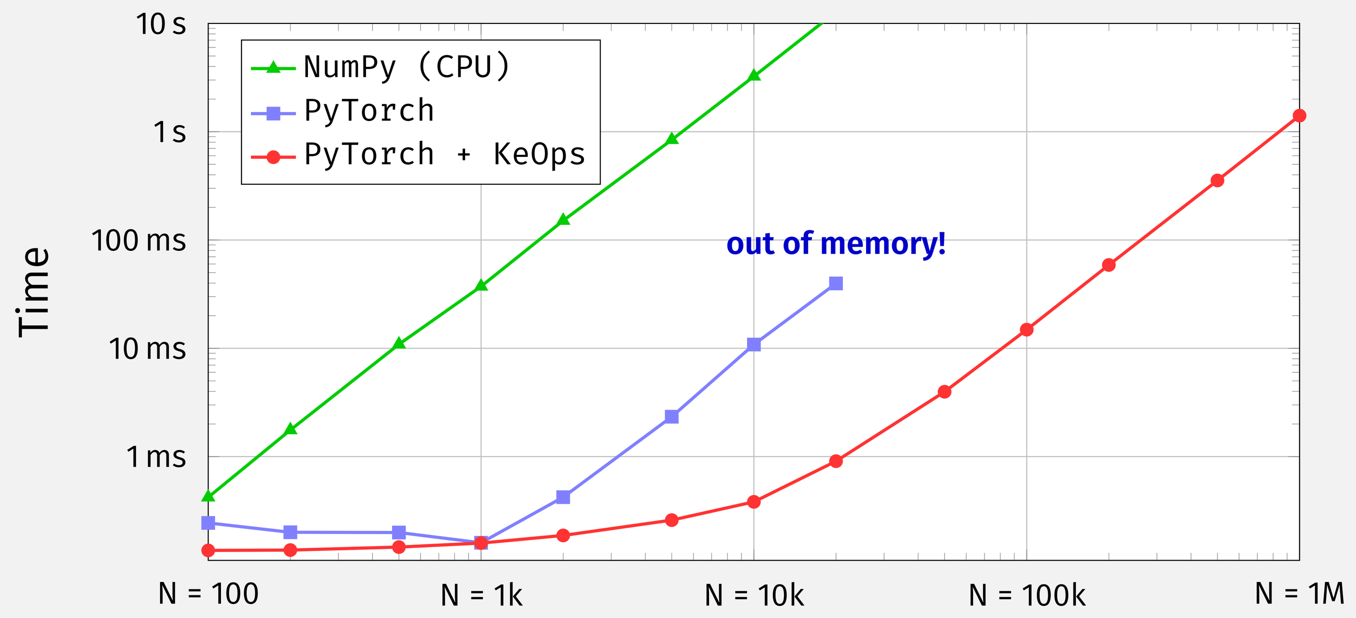
More generally: use symbolic tensors **any way you like**!

```
K_ij = (- D_ij).exp() # (N,M) symbolic Gaussian kernel matrix
a = K_ij @ torch.rand(M,5)  # (N,M) sym.*(M,5) dense = (N,5) dense
g_x, = autograd.grad((a ** 2).sum(), [x])  # Seamless backprop.
```

3 Applications

Symbolic matrices are to **geometric** ML what **sparse** matrices are to **graph** processing.

To illustrate this, we **benchmark** a matrix-vector product with a **N-by-N Gaussian kernel matrix** between 3D point clouds on a RTX 2080 Ti GPU:



For geometric applications in dimension 1 to 100, KeOps **symbolic** tensors:
- \+ Have a negligible **memory** footprint.
- \+ Provide a sizeable **speed-up** for geometric computations.
- – Always rely on **bruteforce** computations.
- – Are less interesting when the formula $F(x_i, y_j)$ is **too large**.

Our top priority for **early 2021** is to mitigate these weaknesses: we will add support for **Tensor cores** and standard **approximation strategies**.

Overall, we believe that **KeOps** will stimulate research on:
- \+ **clustering** algorithms and **UMAP**-like methods,
- \+ **Kernel** methods and **Gaussian** processes,
- \+ **optimal** transport theory,
- \+ **geometric** deep learning and **shape** analysis,
- \+ and even, possibly, natural **language** processing?

We'll be happy to **discuss** these questions with you!

You're welcome to check our **paper**, visit: www.kernel-operations.io and the in-depth tutorial **"Geometric data analysis, beyond convolutions"**: www.jeanfeydy.com/geometric_data_analysis.pdf