An efficient kernel product for autodiff libraries

With applications to measure transport

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

November 13, 2017; GFSW03, Isaac Newton Institute
**What** is PyTorch?  
Deep Learning only $\rightarrow$ Memory overflows
Overview

**What** is PyTorch?
Deep Learning only → Memory overflows

**How** do we fix it?
libkp provides efficient CUDA routines, wrapped in a KernelProduct operator.

Facebook

+ B. Charlier, J. Glaunès
Overview

What is PyTorch?
Deep Learning only → Memory overflows

How do we fix it?
libkp provides efficient CUDA routines, wrapped in a KernelProduct operator.

Where can this bring us?
Normalized Hamiltonian setting.

Facebook
+ B. Charlier, J. Glaunès
+ A. Trouvé
Algorithms typically rely on:

\[ H(q, p) = \frac{1}{2} \langle p, Kq \rangle^2 = \frac{1}{2} \sum_{i,j,k} (q_i, q_j) \langle p_i, p_j \rangle^2 \]

\[ \nabla_q H, \nabla_p H \]
Algorithms typically rely on:

- \( H(q, p) = \frac{1}{2} \langle p, K_q p \rangle_2 = \frac{1}{2} \sum_{i,j} k(q_i, q_j) \langle p_i, p_j \rangle_2 \)
Algorithms typically rely on:

- \( H(q, p) = \frac{1}{2} \langle p, K_q p \rangle_2 = \frac{1}{2} \sum_{i,j} k(q_i, q_j) \langle p_i, p_j \rangle_2 \)
- \( \nabla_q H, \nabla_p H \)
How do we compute a gradient?

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

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

$$
\nabla F(x_0) = \begin{pmatrix}
\partial_{x_1} F(x_0) \\
\partial_{x_2} F(x_0) \\
\vdots \\
\partial_{x_n} F(x_0)
\end{pmatrix} \simeq \frac{1}{\delta t} \begin{pmatrix}
F(x_0 + \delta t \cdot (1, 0, \ldots, 0)) - F(x_0) \\
F(x_0 + \delta t \cdot (0, 1, \ldots, 0)) - F(x_0) \\
\vdots \\
F(x_0 + \delta t \cdot (0, 0, \ldots, 1)) - F(x_0)
\end{pmatrix}.
$$

$$
\Rightarrow \text{costs (N+1) evaluations of } F, \text{ which is poor.}
$$
How do we compute a gradient?

Let \((X, \langle \cdot, \cdot \rangle_X)\) and \((Y, \langle \cdot, \cdot \rangle_Y)\) be two Hilbert spaces. Let \(F : X \to Y\) be a smooth map. Then, we say that:

\[
(d_xF)^*(x_0) : \alpha \in Y^* \to \beta \in X^* \text{ is the adjoint of the differential.}
\]
How do we compute a gradient?

Let \((X, \langle \cdot, \cdot \rangle_X)\) and \((Y, \langle \cdot, \cdot \rangle_Y)\) be two Hilbert spaces. Let \(F : X \to Y\) be a smooth map. Then, we say that:

\[
(d_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^* \text{ is the adjoint of the differential.}
\]

\[
\partial_x F (x_0) : a \in Y \to b \in X \text{ is the gradient.}
\]
How do we compute a gradient?

Let \((X, \langle \cdot, \cdot \rangle_X)\) and \((Y, \langle \cdot, \cdot \rangle_Y)\) be two Hilbert spaces. Let \(F : X \to Y\) be a smooth map. Then, we say that:

\[(d_xF)^*(x_0) : \alpha \in Y^* \to \beta \in X^* \text{ is the adjoint of the differential.}\]

\[\partial_xF (x_0) : a \in Y \to b \in X \text{ is the gradient.}\]

If \(X = \mathbb{R}^n, Y = \mathbb{R}\) endowed with the Euclidean metric,

\[\partial_xF(x_0) = (d_xF(x_0))^T = \begin{pmatrix} \partial_{x_1}F(x_0) \\ \partial_{x_2}F(x_0) \\ \vdots \\ \partial_{x_n}F(x_0) \end{pmatrix}\]
Backpropagating through a computational graph requires:

\[ F_i : E_{i-1} \rightarrow E_i \]
\[ x \mapsto F_i(x) \quad (1) \]

and

\[ \partial_x F_i : E_{i-1} \times E_i \rightarrow E_{i-1} \]
\[ (x_0, a) \mapsto \partial_x F_i(x_0) \cdot a \quad (2) \]

encoded as computer programs.
Backpropagating through a computational graph requires:

\[ F_i : E_{i-1} \rightarrow E_i \]
\[ x \mapsto F_i(x) \quad (1) \]

and

\[ \partial_x F_i : E_{i-1} \times E_i \rightarrow E_{i-1} \]
\[ (x_0, a) \mapsto \partial_x F_i(x_0) \cdot a \quad (2) \]

encoded as computer programs.

This is what PyTorch is all about.
Computing the Hamiltonian

```python
import torch  # GPU + autodiff library
# With PyTorch, using the GPU is that simple:
use_gpu = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_gpu else torch.FloatTensor
```

N = 1000; D = 3 ; # Clouds of 1,000 points in 3D
# Generate arbitrary arrays on the CPU or GPU:
q = torch.from_numpy(...).type(dtype).view(N,D)
p = torch.from_numpy(...).type(dtype).view(N,D)
s = torch.Tensor([2.5]).type(dtype)

# Wrap them into “autodiff” graph nodes. In this demo,
# we won’t try to fine tune the deformation model, so
# we do not need any derivative with respect to s:
q = torch.autograd.Variable(q, requires_grad=True)
p = torch.autograd.Variable(p, requires_grad=True)
s = torch.autograd.Variable(s, requires_grad=False)
```
import torch  # GPU + autodiff library
# With PyTorch, using the GPU is that simple:
use_gpu = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_gpu else torch.FloatTensor

# N = 1000; D = 3 ; # Clouds of 1,000 points in 3D
# Generate arbitrary arrays on the CPU or GPU:
q = torch.from_numpy(...).type(dtype).view(N,D)
p = torch.from_numpy(...).type(dtype).view(N,D)
s = torch.Tensor([2.5]).type(dtype)
import torch  # GPU + autodiff library
# With PyTorch, using the GPU is that simple:
use_gpu = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_gpu \
    else torch.FloatTensor

#
N = 1000; D = 3 ; # Clouds of 1,000 points in 3D
# Generate arbitrary arrays on the CPU or GPU:
q = torch.from_numpy( ... ).type(dtype).view(N,D)
p = torch.from_numpy( ... ).type(dtype).view(N,D)
s = torch.Tensor( [2.5] ).type(dtype)

# Wrap them into "autodiff" graph nodes. In this demo, # we won't try to fine tune the deformation model, so # we do not need any derivative with respect to s:
q = torch.autograd.Variable( q, requires_grad = True )
p = torch.autograd.Variable( p, requires_grad = True )
s = torch.autograd.Variable( s, requires_grad = False)
# Actual computations.

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

```python
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)
```

# Finally, compute the Hamiltonian \( H(q,p) \): \[ \frac{1}{2} \langle p, v \rangle \]

```python
H = 0.5 * torch.dot(p .view(-1), v .view(-1))
```

# Automatic differentiation is straightforward
```python
[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)
# Actual computations.

\[ q_i = q.\text{unsqueeze}(1) \quad \text{# shape (N,D) \rightarrow (N,1,D)} \]
\[ q_j = q.\text{unsqueeze}(0) \quad \text{# shape (N,D) \rightarrow (1,N,D)} \]
\[ \text{sqd} = \text{torch.sum}( (q_i - q_j)**2 , 2 ) \quad \text{# } |q_i-q_j|^2 \]

\[ K_{qq} = \exp\left( - \frac{\text{sqd}}{s^2} \right) \quad \text{# Gaussian kernel} \]

\[ v = K_{qq} \]

\[ \mathbf{p} \quad \text{# matrix mult. (N,N)@(N,D) = (N,D)} \]

# Finally, compute the Hamiltonian \( H(q,p) \):

\[ H = \frac{1}{2} \cdot \langle \mathbf{p}, v \rangle \]

# Automatic differentiation is straightforward

\[ [\mathbf{dq}, \mathbf{dp}] = \text{torch.autograd.grad}( H, [q,p], 1.) \]

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

# Display -- see next figure.

\[ \text{make dot}(H, \{ \text{'q':q, 'p':p, 's':s} \}).\text{render}(\text{view=True}) \]
# Actual computations.

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

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

# Finally, compute the Hamiltonian \( H(q,p) \):

```python
H = 0.5 * torch.dot(p.view(-1), v.view(-1))
```

Automatic differentiation is straightforward:

```python
[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.

```python
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)  # |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

```python
# 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) # |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))
```

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)  # |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.)
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) # |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.)

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

# Actual computations.
q_i = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd = torch.sum((q_i - q_j)**2, 2) # |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)
\[
\begin{align*}
\partial q & \quad (1000, 3) \\
\partial \text{Unsqueeze} & \quad \partial \text{Unsqueeze} \\
\partial \text{Sub} & \\
\partial \text{PowConstant} & \quad q_i - q_j & \quad (1000, 1000, 3) \\
\partial \text{Sum} & \\
\partial \text{PowConstant} & \quad s^2 & \quad (1) \\
\partial \text{Div} & \quad -\|q_i - q_j\|^2 / s^2 & \quad (1000, 1000) \\
\partial \text{Addmm} & \quad p & \quad (1000, 3) \\
\partial \text{Exp} & \quad p & \quad (1000, 1000) \\
\partial \text{Addmm} & \quad K_{q,q} & \quad (1000, 1000) \\
\partial \text{Dot} & \quad p & \quad (3000) \\
\partial \text{Dot} & \quad K_{q,q} p & \quad (3000) \\
\partial \text{MulConstant} & 
\end{align*}
\]
Our contribution

# Compute the kernel convolution
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) )
How does one compute

\[ g_i = \sum_j k(x_i - y_j) b_j \]

on the GPU?
Memory management in CUDA

Leonhard Euler: the perfect XVIIIth century CPU.
Memory management in CUDA

1884: a new age of parallel computing.
Memory management in CUDA

1884: a new age of parallel computing.
1884: inside a computing block.
Memory management in CUDA

1884: inside a computing block.
KernelProd CUDA program executed by a Block

| Input    | in GM: x, y, b  
|          | in TM: BlockId, ThreadId |
| Parameter| $k: x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc. |
| Output   | $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$ |
KernelProd CUDA program executed by a Block

Input:
in GM: x, y, b
in TM: BlockId, ThreadId

Parameter:
k: x^2 \mapsto \exp(-\|x\|^2/\sigma^2), etc.

Output:
\((g_i) = \sum_j k(x_i - y_j) \cdot b_j\)

1: \(i = \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId}\)
KernelProd CUDA program executed by a Block

Input : in GM: x, y, b
         in TM: BlockId, ThreadId

Parameter: k : $x^2 \mapsto \exp\left(-\frac{||x||^2}{\sigma^2}\right)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

1: $i = \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId}$
2: $g[i] = [\theta, \ldots, \theta]$; load $x[i]$ in TM
### KernelProd CUDA program executed by a Block

**Input**: in \( \text{GM}: x, y, b \)

\[ \text{in TM}: \text{BlockId}, \text{ThreadId} \]

**Parameter**: \( k : x^2 \mapsto \exp\left(-\|x\|^2/\sigma^2\right), \text{etc.} \)

**Output**: \((g_i) = \sum_j k(x_i - y_j) \cdot b_j\)

1. \( i = \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId} \)
2. \( g[i] = [0, \ldots, 0]; \text{load } x[i] \text{ in TM} \)
3. \( \text{for } (J=0; J<M; J+=\text{BlockSize}) \text{ do} \)
KernelProd CUDA program executed by a Block

Input : in GM: x, y, b
        in TM: BlockId, ThreadId
Parameter : k: \( x^2 \mapsto \exp(-\|x\|^2/\sigma^2) \), etc.
Output : \( (g_i) = \sum_j k(x_i - y_j) \cdot b_j \)

1: \( i = \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId} \)
2: \( g[i] = [0, \ldots, 0] \); load \( x[i] \) in TM
3: for (J=0; J<M; J+=\text{BlockSize}) do
4: Load in parallel \( (j \in [J, J+\text{BlockSize}[]) \) in SM: \( y[j], b[j] \)
KernelProd CUDA program executed by a Block

Input : in GM: x, y, b
        in TM: BlockId, ThreadId

Parameter: \( k : x^2 \mapsto \exp(-||x||^2/\sigma^2) \), etc.

Output : \((g_i) = \sum_j k(x_i - y_j) \cdot b_j\)

1: \( i = \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId} \)
2: \( g[i] = [\theta, \ldots, \theta] \); load \( x[i] \) in TM
3: for (J=0; J<M; J+=\text{BlockSize}) do
4:   Load in parallel \((j \in [J, J+\text{BlockSize}] \) in SM: y[j], b[j]
5:   for (j=J; j<J+\text{BlockSize}; j++) do
KernelProd CUDA program executed by a Block

Input : in GM: \(x, y, b\)
in TM: BlockId, ThreadId

Parameter : \(k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2), \text{etc.}\)

Output : \((g_i) = \sum_j k(x_i - y_j) \cdot b_j\)

1: \(i = \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId}\)
2: \(g[i] = [0, \ldots, 0]; \text{load } x[i] \text{ in TM}\)
3: for (\(J=0; J<M; J+=\text{BlockSize}\)) do
4: Load in parallel (\(j \in [J,J+\text{BlockSize}]\)) in SM: \(y[j], b[j]\)
5: for (\(j=J; j<J+\text{BlockSize}; j++\)) do
6: \(r2 = \text{sum}((x[i] - y[j])**2)\)
7: \(g[i] += k(r2) \cdot b[j]\)
KernelProd CUDA program executed by a Block

**Input**: in GM: x, y, b

in TM: BlockId, ThreadId

**Parameter**: k : x^2 → exp(-∥x∥^2/σ^2), etc.

**Output**: (g_i) = \sum_j k(x_i - y_j) \cdot b_j

1: i = BlockId \cdot BlockSize + ThreadId
2: g[i] = [0,...,0]; load x[i] in TM
3: for (J=0; J<M; J+=BlockSize) do
4: Load in parallel (j ∈ [J,J+BlockSize[) in SM: y[j], b[j]
5: for (j=J; j<J+BlockSize; j++) do
6:  
7: Push g[i] back in the GM
A flexible and scalable development framework

PyTorch + libkp:

- No need to write backwards anymore
- No more memory overflows
PyTorch + libkp:

- No need to write backwards anymore
- No more memory overflows

⇒ Try out your ideas within a couple of hours!
Normalizing Hamiltonians to get mass awareness
In the computational sense, it is the **cheapest** way to build regularizing metrics on point clouds:

- Hamilton’s theorem \((g_q \rightarrow K_q)\)
- The current availability of GPUs (parallelism)
Is LDDMM the missing link between Monge and Procustes?

If $k$ is a smooth enough kernel function, it defines a RKHS norm

$$
\|v\|_k^2 = \langle v, k(-1) \ast v \rangle = \int_{\mathbb{R}^d} \frac{1}{\hat{k}(\omega)} |\hat{v}(\omega)|^2 \, d\omega, \quad (3)
$$

$$
\|p\|_k^* = \langle p, k \ast p \rangle. \quad (4)
$$
Is LDDMM the missing link between Monge and Procustes?

If \( k \) is a smooth enough kernel function, it defines a RKHS norm

\[
\|v\|_k^2 = \langle v, k^{(-1)} * v \rangle = \int_{\mathbb{R}^d} \frac{1}{\hat{k}(\omega)} |\hat{v}(\omega)|^2 \, d\omega, \quad (3)
\]

\[
\|p\|_k^{*2} = \langle p, k * p \rangle. \quad (4)
\]

The **Reduction Principle:**

\[
(q_t, p_t) \leftrightarrow \varphi_t \text{ where } \varphi_t \text{ is } k\text{-smooth} \quad (5)
\]
If $k$ is a smooth enough kernel function, it defines a RKHS norm

$$\|v\|_k^2 = \langle v, k^{(-1)} \ast v \rangle = \int_{\mathbb{R}^d} \frac{1}{\hat{k}(\omega)} |\hat{v}(\omega)|^2 \, d\omega,$$

(3)

$$\|p\|_k^*^2 = \langle p, k \ast p \rangle.$$  

(4)

The Reduction Principle:

$$(q_t, p_t) \longleftrightarrow \varphi_t \quad \text{where } \varphi_t \text{ is } k\text{-smooth}$$

(5)

On landmarks, one could be tempted to believe that:

Wasserstein ($\sigma = 0$) $\xrightarrow{\sigma^{++}} \| \cdot \|_k$ $\xrightarrow{\sigma^{++}}$ ($\sigma = \infty$) Translations
Contributions:

- Flexible and scalable development tools.
Recap of today’s presentation

Contributions:

• Flexible and scalable development tools.
• Implement easily metrics which are not right-invariant.
Recap of today’s presentation

Contributions:

- Flexible and scalable development tools.
- Implement easily metrics which are not right-invariant.

Schedule:

Today: Detailed PDF report + Git (Numpy, PyTorch, Matlab and R bindings), see
www.math.ens.fr/~feydy/research.html

1st of Dec.: Full report on Arxiv.
1st of Jan.: libkp completed: Currents, Varifolds, etc.
1st of Apr.? Full Normalized Hamiltonians paper.
Thank you for your attention.