



# Debugging PyTorch code

**Dmytro Mishkin, FEE, CTU in Prague**

**Before we start**

**Coding is not a sprint, it is a marathon**

# Coding is not a sprint, it is a marathon

- You should minimize your suffering

# Coding is not a sprint, it is a marathon

- You should minimize your suffering
  - Even better if you have fun.

# Coding is not a sprint, it is a marathon

- You should minimize your suffering
  - Even better if you have fun.
  - Spend some time on learning tools - matplotlib, pdb, jupyter notebooks.

# Coding is not a sprint, it is a marathon

- You should minimize your suffering
  - Even better if you have fun.
- Spend some time on learning tools - matplotlib, pdb, jupyter notebooks.
  - Also good to make yourself familiar with the main libraries you use: numpy, pytorch.

# Coding is not a sprint, it is a marathon

- You should minimize your suffering
  - Even better if you have fun.
- Spend some time on learning tools - matplotlib, pdb, jupyter notebooks.
  - Also good to make yourself familiar with the main libraries you use: numpy, pytorch.
  - Usually there is already a function, which implements what you want



# Coding is not a sprint, it is a marathon

- You should minimize your suffering
  - Even better if you have fun.
  - Spend some time on learning tools - matplotlib, pdb, jupyter notebooks.
    - Also good to make yourself familiar with the main libraries you use: numpy, pytorch.
    - Usually there is already a function, which implements what you want
- And have enough sleep.

# Two kinds of bugs

# Two kinds of bugs

- It throws an error, then read the error message.

# Two kinds of bugs

- It throws an error, then read the error message.
  - pdb is your friend. StackOverflow is your friend. ChatGPT is your friend. Error is your friend.

# Two kinds of bugs

- It throws an error, then read the error message.
  - pdb is your friend. StackOverflow is your friend. ChatGPT is your friend. Error is your friend.
- It does not crash, but doesn't work as expected. That's harder, usually.

# Debug checklist: general

# Debug checklist: general

- Garbage in, garbage out. Therefore **check your inputs before anything else.**

# Debug checklist: general

- Garbage in, garbage out. Therefore **check your inputs before anything else.**
- Debugging the system is hard. Always try to **isolate the problem**, and **work with a single function**



# Debug checklist: general

- Garbage in, garbage out. Therefore **check your inputs before anything else.**
- Debugging the system is hard. Always try to **isolate the problem**, and **work with a single function**
  - **Write down toy-input and expected output.**

# Debug checklist: general

- Garbage in, garbage out. Therefore **check your inputs before anything else.**
- Debugging the system is hard. Always try to **isolate the problem**, and **work with a single function**
  - **Write down toy-input and expected output.**
- Print/log everything. Input, outputs, types, counters. Everything.

# Debugging. Specific advices

## Data type

- Check the data type.

# Debugging. Specific advices

## Data type

- Check the data type.

```
>>> import numpy as np
>>> a=[1,2]
>>> b=[3,4]
>>> a+b
[1, 2, 3, 4]
```

# Debugging. Specific advices

## Data type

- Check the data type.

```
>>> import numpy as np
>>> a=[1,2]
>>> b=[3,4]
>>> a+b
[1, 2, 3, 4]

>>> np.array(a) + np.array(b)
array([4, 6])
```

**Specific example. What would be  $1+1$ ?**

# Specific example. What would be 1+1?

```
>>> import torch
>>> a = torch.tensor([1,1])
>>> b = torch.ones(2)
>>> c = torch.zeros(2) + 1
```

# Specific example. What would be 1+1?

```
>>> import torch
>>> a = torch.tensor([1,1])
>>> b = torch.ones(2)
>>> c = torch.zeros(2) + 1

>>> print (a.dtype, b.dtype, c.dtype)
torch.int64 torch.float32 torch.float32
```



# Specific example. What would be 1+1?

```
>>> import torch
>>> a = torch.tensor([1,1])
>>> b = torch.ones(2)
>>> c = torch.zeros(2) + 1

>>> print (a.dtype, b.dtype, c.dtype)
torch.int64 torch.float32 torch.float32

[>>> c[a]
tensor([1., 1.]
```

# Specific example. What would be 1+1?

```
>>> import torch
>>> a = torch.tensor([1,1])
>>> b = torch.ones(2)
>>> c = torch.zeros(2) + 1

>>> print (a.dtype, b.dtype, c.dtype)
torch.int64 torch.float32 torch.float32
```

```
[>>> c[a]
tensor([1., 1.])
```

```
[>>> a[c]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tensors used as indices must be long, int, byte or bool tensors
```

# Specific example. What would be 1+1?

```
>>> import torch
>>> a = torch.tensor([1,1])
>>> b = torch.ones(2)
>>> c = torch.zeros(2) + 1
```

```
>>> print (a.dtype, b.dtype, c.dtype)
torch.int64 torch.float32 torch.float32
```

```
[>>> c[a]
tensor([1., 1.])
```

```
[>>> a[c]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tensors used as indices must be long, int, byte or bool tensors
```

```
>>> a+b
tensor([2., 2.])
>>> b+a
tensor([2., 2.])
```

**Some operations silently change data type,  
Others do not**

**Some operations silently change data type,  
Others do not**

```
[>>> a+1  
tensor([2, 2])  
_
```

# Some operations silently change data type,

Others do not

```
[>>> a+1  
tensor([2, 2])  
_
```

```
[>>> a*2  
tensor([2, 2])
```



# Some operations silently change data type, Others do not

```
[>>> a+1  
tensor([2, 2])
```

```
[>>> a*2  
tensor([2, 2])
```

```
[>>> a/1  
tensor([1., 1.])
```

# It is not always you

- Sometimes libraries have bugs too.
- Double check before blaming them, though.
- When you find a bug in an open source library - raise issue on GitHub.



# Shape and broadcasting

- Some operations depend on shape.

# Shape and broadcasting

- Some operations depend on shape.

```
a = torch.tensor([1,2,3]).float()  
b = torch.tensor([1,2,3]).float()
```

# Shape and broadcasting

- Some operations depend on shape.

```
a = torch.tensor([1,2,3]).float()  
b = torch.tensor([1,2,3]).float()
```

```
def mul_with_print(a, b):  
    c = a * b  
    print (f'a.shape = {a.shape}, b.shape={b.shape}, c.shape={c.shape}')  
    print (f'c={c}')
```

# Shape and broadcasting

- Some operations depend on shape.

```
a = torch.tensor([1,2,3]).float()  
b = torch.tensor([1,2,3]).float()
```

```
def mul_with_print(a, b):  
    c = a * b  
    print (f'a.shape = {a.shape}, b.shape={b.shape}, c.shape={c.shape}')  
    print (f'c={c}')
```

```
[In [5]: mul_with_print(a, b)  
a.shape = torch.Size([3]), b.shape=torch.Size([3]), c.shape=torch.Size([3])  
c=tensor([1., 4., 9.])
```

# Shape and broadcasting

```
[In [9]: mul_with_print(a.reshape(3,1), b)
a.shape = torch.Size([3, 1]), b.shape=torch.Size([3]), c.shape=torch.Size([3, 3])
c=tensor([[1., 2., 3.],
          [2., 4., 6.],
          [3., 6., 9.]])
```

```
[In [10]: mul_with_print(a.reshape(3,1, 1), b)
a.shape = torch.Size([3, 1, 1]), b.shape=torch.Size([3]), c.shape=torch.Size([3, 1, 3])
c=tensor([[[1., 2., 3.]],
          [[2., 4., 6.]],
          [[3., 6., 9.]])
```

```
[In [11]: mul_with_print(a.reshape(3,1, 1, 1), b)
a.shape = torch.Size([3, 1, 1, 1]), b.shape=torch.Size([3]), c.shape=torch.Size([3, 1, 1, 3])
c=tensor([[[[1., 2., 3.]]],
          [[[2., 4., 6.]]],
          [[[3., 6., 9.]]]])
```

# Shape and broadcasting



# Shape and broadcasting

```
[In [12]: mul_with_print(a.reshape(3,1), b.reshape(3,1))  
a.shape = torch.Size([3, 1]), b.shape=torch.Size([3, 1]), c.shape=torch.Size([3, 1])  
c=tensor([[1.],  
          [4.],  
          [9.]])
```

# Shape and broadcasting

```
[In [12]: mul_with_print(a.reshape(3,1), b.reshape(3,1))
a.shape = torch.Size([3, 1]), b.shape=torch.Size([3, 1]), c.shape=torch.Size([3, 1])
c=tensor([[1.],
         [4.],
         [9.]])
```

```
[In [13]: mul_with_print(a.reshape(3,1), b.reshape(1,3))
a.shape = torch.Size([3, 1]), b.shape=torch.Size([1, 3]), c.shape=torch.Size([3, 3])
c=tensor([[1., 2., 3.],
         [2., 4., 6.],
         [3., 6., 9.]])
```



# Shape and broadcasting

# Shape and broadcasting

- Solution 1: understand broadcasting

# Shape and broadcasting

- Solution 1: understand broadcasting
  - <https://numpy.org/doc/stable/user/basics.broadcasting.html>
  - <https://pytorch.org/docs/stable/notes/broadcasting.html>

# Shape and broadcasting

- Solution 1: understand broadcasting
  - <https://numpy.org/doc/stable/user/basics.broadcasting.html>
  - <https://pytorch.org/docs/stable/notes/broadcasting.html>
- Solution 2: check the shape in the input, throw error if not expected

# Shape and broadcasting

- Solution 1: understand broadcasting
  - <https://numpy.org/doc/stable/user/basics.broadcasting.html>
  - <https://pytorch.org/docs/stable/notes/broadcasting.html>
- Solution 2: check the shape in the input, throw error if not expected

# Shape and broadcasting

- Solution 1: understand broadcasting
  - <https://numpy.org/doc/stable/user/basics.broadcasting.html>
  - <https://pytorch.org/docs/stable/tensors.html>
- Solution 2: check the shape if not expected

```
def find_fundamental(
    points1: torch.Tensor, points2: torch.Tensor, weights: Optional[torch.Tensor] = None
) -> torch.Tensor:
    """Compute the fundamental matrix using the DLT formulation.

    The linear system is solved by using the Weighted Least Squares Solution for the 8 Point Problem.

    Args:
        points1: A set of points in the first image with a tensor shape :math:`(B, N, 2)`,
        points2: A set of points in the second image with a tensor shape :math:`(B, N, 2)`,
        weights: Tensor containing the weights per point correspondence with a shape of :math:`(B, N)`

    Returns:
        the computed fundamental matrix with shape :math:`(B, 3, 3)`.
    """
    if points1.shape != points2.shape:
        raise AssertionError(points1.shape, points2.shape)
    if points1.shape[1] < 8:
        raise AssertionError(points1.shape)
    if not (weights is None):
        if not (len(weights.shape) == 2 and weights.shape[1] == points1.shape[1]):
            raise AssertionError(weights.shape)
```

# Memory sharing

# Memory sharing

- Many python objects share memory, e.g. lists, np.arrays, dicts



# Memory sharing

- Many python objects share memory, e.g. lists, np.arrays, dicts

```
[In [1]: a=[1, 2]
```

```
[In [2]: b = a
```

```
[In [3]: b[1]+=1
```

```
[In [4]: print (a, b)  
[1, 3] [1, 3]
```

```
In [5]: █
```

# Memory sharing

- Many python objects share memory, e.g. lists, np.arrays, dicts
- your friend is:

```
[In [1]: a=[1, 2]
```

```
[In [2]: b = a
```

```
[In [3]: b[1]+=1
```

```
[In [4]: print (a, b)  
[1, 3] [1, 3]
```

```
In [5]: █
```

# Memory sharing

- Many python objects share memory, e.g. lists, np.arrays, dicts
- your friend is:
  - from copy import deepcopy

```
[In [1]: a=[1, 2]
```

```
[In [2]: b = a
```

```
[In [3]: b[1]+=1
```

```
[In [4]: print (a, b)  
[1, 3] [1, 3]
```

```
In [5]: █
```

# Memory sharing

- Many python objects share memory, e.g. lists, np.arrays, dicts
- your friend is:
  - from copy import deepcopy

```
[In [10]: c=deepcopy(a)
[In [11]: c[1]+=1
[In [12]: print (a, c)
[1, 3] [1, 4]
```

```
[In [1]: a=[1, 2]
[In [2]: b = a
[In [3]: b[1]+=1
[In [4]: print (a, b)
[1, 3] [1, 3]
In [5]: █
```

# Always check xy order



**Michał Tyszkiewicz**

@jatentak

...

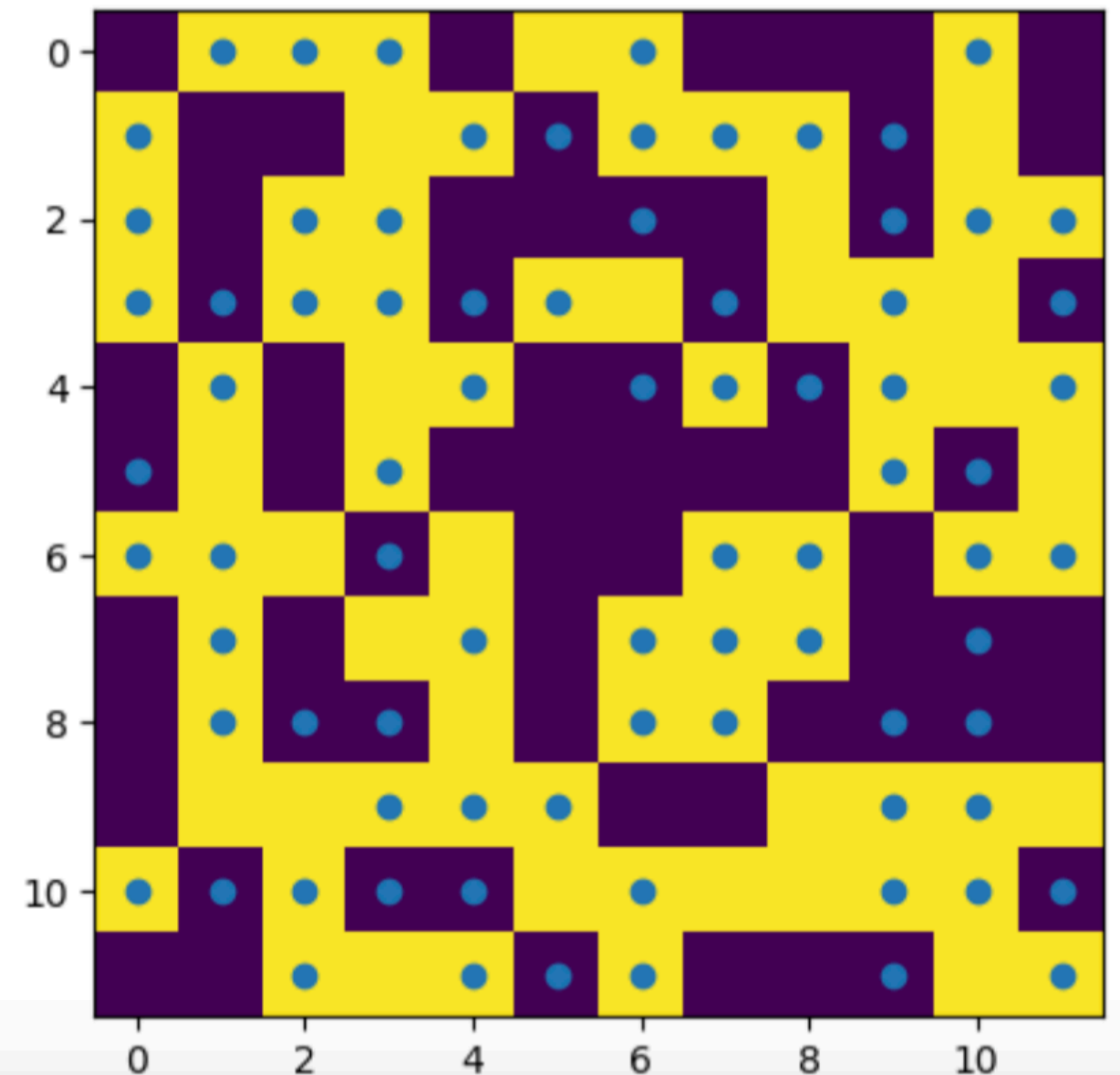
Odpověď uživatelům [@ducha\\_aiki](#) a [@kornia\\_foss](#)

A common bug: confusing height and width (x and y) in some reshape operation. A classic followup is trying to debug it with matplotlib and getting further confused by the difference between scatter and imshow conventions.

[Přeložit Tweet](#)

```
import matplotlib.pyplot as plt
import torch

image = torch.randn(12, 12) > 0
x, y = torch.where(image)
plt.imshow(image)
_ = plt.scatter(x, y)
```



# Checklist

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?



# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function

```
[In [6]: import kornia as K

[In [7]: K = torch.eye(4)

[In [8]: K.tensor_to_image(K)
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-8-93a4af1f8c92> in <module>
----> 1 K.tensor_to_image(K)

AttributeError: 'Tensor' object has no attribute 'tensor_to_image'
```

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?
  - E.g. old `torch.solve(B, A)`, but `torch.linalg.solve(A, B)`

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?
  - E.g. old `torch.solve(B, A)`, but `torch.linalg.solve(A, B)`
6. Do I have NaN-prone operations?

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?
  - E.g. old `torch.solve(B, A)`, but `torch.linalg.solve(A, B)`
6. Do I have NaN-prone operations?
  - e.g. `log`, `sqrt`, `division`, etc. Use `eps` there or some kind of guards



# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?
  - E.g. old `torch.solve(B, A)`, but `torch.linalg.solve(A, B)`
6. Do I have NaN-prone operations?
  - e.g. `log`, `sqrt`, `division`, etc. Use `eps` there or some kind of guards
7. Do I have some memory sharing?

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?
  - E.g. old `torch.solve(B, A)`, but `torch.linalg.solve(A, B)`
6. Do I have NaN-prone operations?
  - e.g. `log`, `sqrt`, `division`, etc. Use `eps` there or some kind of guards
7. Do I have some memory sharing?
8. Is there anything hardcoded?

# Checklist

1. Did I prepared minimal input and expected output? Math-based, or reliable library based
2. Did I visualize everything?
3. Did I printed shape, data types, and values?
4. Did I checked for a stupid mistakes? Like typos in variable names, naming variables as function
5. Did I checked library versions and updates?
  - E.g. old `torch.solve(B, A)`, but `torch.linalg.solve(A, B)`
6. Do I have NaN-prone operations?
  - e.g. `log`, `sqrt`, `division`, etc. Use `eps` there or some kind of guards
7. Do I have some memory sharing?
8. Is there anything hardcoded?
9. Can the bug in one function be compensated by other bug in other function?