- 用arc diff提交代码的步骤：git add->git commit->arc diff
  git add: vscode中Changes右侧加号。
  git commit: Changes上方勾号，在弹出来的 `Message(commit on 'dev')` 中填写注释。(commit是本地的操作，跟origin/master没关系，commit当然是到dev)
  `arc diff --update Dxxx`，进入GNU nano以后^O esc+M enter X(为control键)
- 拉取origin/master的新版本的话，切到master分支下面git pull就行。vscode里面都可以直接操作：…-拉取。
- python多行代码注释快捷键：选中后，按command+/
- Mac本地没有NVIDIA的卡，没有cuda环境，也就不能使用cupy。
- tensor默认在cpu上。为了tensor在gpu上跑，在新建一个tensor时都要指定device。简单的比如 `x = torch.randn((2,2*3), device="cuda")`
-

```
args = parse() #从命令行，也就是slurm文件中读取参数
torch.cuda.set_device(args.local_rank) #pytorch的分布式机制跟cuda C类似，使用多个进程同时跑同一份代码，但是rank不同。local_rank=0/1/2/3，是指同时启动了4个进程，r
torch.distributed.init_process_group('nccl', init_method = 'env://') #init_method = 'env://'是环境的设置，不用管。
device = torch.device(f'cuda:{args.local_rank}') #f是formatted string，是python比较新的版本提供的一种格式化字符串机制。
if args.local_rank == 0:
    xxxxx
```

- 如何给vscode给python文件头模板？右下角齿轮-User Snippets-python-进入python.json文件-添加如下代码-在.py文件中输入header即可弹出。（用户代码片段：此网页）

```
"header": {
    "prefix": "header",
    "body": [
        "#!/usr/bin/env python3",
        "# -*- coding: utf-8 -*-"
        "\"\"\"",
        "@Time    :   $CURRENT_YEAR/$CURRENT_MONTH/$CURRENT_DATE $CURRENT_HOUR:$CURRENT_MINUTE:$CURRENT_SECOND",
        "@Author  :   Joey Z",
        "\"\"\"",
    ],
    }
```

- .json文件中不允许添加注释（所以python使用的 `%` `#` 都不可以）
- 要让代码能在gpu上运行，首先函数作用的对象要在gpu(device)上，其次函数要是gpu的版本。gpu和cpu的计算架构完全不同，一个函数不可能无缘无故就能作用在gpu上的数据，所以需要重新写代码。有些代码是cupy已经提供了的，所以不用再写。
  所有代码都是cpu来跑的，cpu能做的事情是给gpu提交任务。

reconstruct.py

- `from torch.utils.dlpack import to_dlpack, from_dlpack` , 将dlpack与tensor作转换。
  `torch.utils.dlpack.from_dlpack(dlpack) → Tensor` ：Decodes a DLPack to a tensor.
  `torch.utils.dlpack.to_dlpack(tensor) → PyCapsule` ：Returns a DLPack representing the tensor.
  dlpack – a PyCapsule object with the dltensor.
  PyCapsule: This subtype of PyObject represents an opaque value, useful for C extension modules who need to pass an opaque value (as a void* pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.
- cupy的ndarray，numpy的ndarray，pytorch的tensor两两相互转化：
  cupy的ndarray与numpy的ndarray相互转化：

```
#cupy->numpy
numpy_data = cp.asnumpy(cupy_data)

#numpy->cupy
cupy_data = cp.asarray(numpy_data)
```

pytorch的tensor与numpy的ndarray相互转化：

tensor->ndarray: `torch.tensor.numpy()`

ndarray->tensor: `torch.from_numpy(numpy.ndarray(shape=(1,2)))`

pytorch的tensor与cupy的ndarray相互转化：

```
#tensor->cupy的ndarray
cupy_data = cp.fromDlpack(to_dlpack(tensor_data))

#cupy的ndarray->tensor
tensor_data = cp.from_dlpack(toDlpack(cupy_data))
```

- torch.utils.data.random_split(): Randomly split a dataset into non-overlapping new datasets of given lengths.

```
n_val = int(len(dataset) * val_percent)
n_train = len(dataset) - n_val
train, val = random_split(dataset, [n_train, n_val])
```

- 对tensor的for循环，首先对tensor的第一维度循环，而非其所有元素。

```
import torch
x = torch.randn((2,2*3))
for v in x:
    print(v)

Output:

tensor([-0.2150,  0.5214,  0.7759,  0.6315,  0.5889,  1.1286])
tensor([ 0.8052,  0.7156, -1.0608, -0.1623,  0.6096,  0.8428])
```

- ndarray.reshape()不改变ndarray本身的值
- torch.distributed.reduce(tensor, dst, op=<ReduceOp.SUM: 0>, group=, async_op=False)
  Reduces the tensor data across all machines. Only the process with rank dst is going to receive the final result.把进程里面的变量放在一起做一个操作。

- torch.distributed.broadcast(tensor, src, group=, async_op=False)
  Broadcasts the tensor to the whole group.把某个进程的某个变量广播给其他所有进程。

  每个进程算一部分梯度，最后大家一起求和。求和完了梯度返回给所有进程，保证每个进程梯度是一样的。这就叫并行计算。

  - The package argparse will figure out how to parse those out of sys.argv. There are two other modules that fulfill the same task, namely getopt (an equivalent for getopt() from the C language) and the deprecated optparse. Note also that argparse is based on optparse, and therefore very similar in terms of usage.

```
# 声明一个parser
parser = argparse.ArgumentParser()
# 添加参数
parser.add_argument()
# 读取命令行参数
parser.parse_args()
```

  - sys是Python的一个「标准库」，也就是官方出的「模块」，是「System」的简写，封装了一些系统的信息和接口。「argv」是「argument variable」参数变量的简写形式，一般在命令行调用的时候由系统传递给程序。这个变量其实是一个List列表，argv[0] 一般是

被调用的脚本文件名或全路径，和操作系统有关，argv[1]和以后就是传入的数据了。

-

```
seq = ['one', 'two', 'three']
for i, element in enumerate(seq):
    print(i,element)


0 one
1 two
2 three
```

- function zip():

```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica")

x = zip(a, b)


(('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica'))
```

- f'{}': Formatted string literals，是python比较新的版本提供的一种格式化字符串机制。f是跟它后面的字符串构成一个整体的，与C的printf()不同。
- 多在CRS写实验报告，写在http://124.65.131.150:11111/T351。

test_adam_sortedL1loss.py

- 代码里的gradient_func：输入model,imgs，对imgs平移，projs=model投影，projs卷ctf，(projs-imgs)卷ctf，最后back_project并return。

, so

(T是平移，C是卷ctf，P是投影),

，

对这个换过的2范数取梯度:

back_project.py==

x:model, b:imgs

==imgs平移，projs=model投影，projs卷ctf，(projs-imgs)卷ctf，最后back_project并return呢

trans在读取的时候就加了负号了。

- ctf跟光学成像有关，确切的说是3维的，且每个z-slice都是不同的。但是single particle模型里面不管这些，因为冰层厚度小，忽略z轴的差异。所以可以当一个二维卷积核。它可以由那几个参数计算出来，不需要存储成矩阵。
  实空间卷积一个函数相当于Fourier空间逐点乘这个函数的Fourier变换。这个ctf的Fourier变换在Fourier空间中也是一个中心对称的实值函数（这么说好像不太好，说球对称吧。就是，f(x,y,z)只和 有关。）。就是说，这个函数的函数值是实的，并且，只和频率有关。
- back_project.py== ：读back_project的时候跟project对比一下，你就知道为什么是转置了。
  打开device_back_project.cu, device_project.cu，发现无法理解为什么device_project.cu实现了project了，它甚至没有for循环。答：针对voxel的循环由cuda的并行机制解决了，cuda的并行机制是一次性开启大量的线程，每个线程跑同样的代码，但是拥有不同的id，所以你就开# of voxel这么多个线程，每个线程只负责投影一个voxel，所以就没有for了。
  device_project.cu里干了事的代码只有两行（两个return语句）：
  atomicAdd(image + dev_logic_image_index(i_col, i_row, n_col, n_row), value);#每个线程只负责投影一个voxel，负责单个进程的投影
  *voxel_mdl += wt * pixel_prj[dev_logic_image_index(i_col, i_row, n_col, n_row)];
  为什么要atomic？因为可能存在两个voxel同时要往一个pixel写数据的情况，防止写入冲突。
  wt是啥？看global_back_project.cu里调用dev_back_project的地方。
  *是dereference操作符，用来获取一个指针指向的对象。

本来方程是Ax-b，也就是TCPx-b（T是平移，C是卷ctf，P是投影），我们把它换成CPx-T^{-1}b了，然后对着这个换过的2范数取梯度。

trans在读取的时候就加了负号了。

- torch.distributed.get_world_size(group=