这段代码是一个Python脚本，导入了几个库和模块，包括numpy、matplotlib、mpi4py和numba。脚本还定义了几个函数，并使用导入的库创建了一个动画。

以下是所使用库和模块的简要描述：

- `numpy`：Python的数值计算库，用于定义数组并在其上执行数学运算。
- `matplotlib`：Python的数据可视化库，用于创建动画。
- `mpl_toolkits.mplot3d.axes3d`：matplotlib的一个模块，用于创建3D图形。
- `matplotlib.animation`：matplotlib的一个模块，用于创建动画。
- `math`：Python的数学库，包含一些基本的数学函数。
- `cmath`：Python的复数数学库，包含一些基本的复数数学函数。
- `time`：Python的时间库，包含一些处理时间的函数。
- `mpi4py`：Python的MPI（Message Passing Interface）库，用于并行计算。
- `sys`：Python的系统库，包含一些系统相关的函数。
- `numba`：Python的一个即时编译器，用于加速Python代码。
- `numba.cuda`：numba的一个模块，用于在GPU上执行加速计算。
- `numba.cuda.random`：numba.cuda的一个模块，用于在GPU上生成随机数。

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation
import math
import cmath
import time
from mpl_toolkits.mplot3d import Axes3D
import mpi4py
import sys
from mpi4py import MPI
from numba import cuda
from numba.cuda.random import create_xoroshiro128p_states, xoroshiro128p_normal_fl
from timeit import default_timer as timer
```

这段代码使用了mpi4py库中的MPI模块来创建了一个MPI通信的通信器（communicator）`comm`，并通过该通信器获取了当前进程的排名（rank）。

MPI（Message Passing Interface）是一种在分布式内存并行计算中进行进程间通信的标准。mpi4py是Python对MPI的一个封装，使得Python程序能够使用MPI进行并行计算。在这段代码中，`MPI.COMM_WORLD`创建了一个包含所有进程的通信器，每个进程通过`comm.Get_rank()`获取自己在通信器中的排名（从0开始）。

In [ ]:
```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

In [ ]:
```
N = M = Lx = Lz = 256  # 用于定义一个立方体网格。
mu, sigma = 0, 1  # 分别为正态分布的均值和标准差。这些变量用于生成随机数
Dw = (4, 0)  # 表示了风的方向，第一个数字为东西方向上的风向，第二个数字为南北方向上的风向
frame = 1  # 表示当前帧的编号
Vw = 30  # 表示风的速度
A = 0.003**2 * 0.02  # 表示随机力的振幅，用于描述受到的随机力的大小。
```

In [ ]:
```
epsilon = 0  # 表示流体的黏性系数
g = 9.8  # 表示重力加速度
```

```
x = np.arange(N) - int(N/2)  # 表示立方体网格在x方向上的坐标
z = np.arange(N) - int(N/2)  #表示立方体网格在z方向上的坐标
x_mesh, z_mesh = np.meshgrid(x, z)  # 将x和z两个一维数组转换为网格坐标（二维数组）
yarray = np.zeros((Lx, Lz, frame))
# yarray 表示立方体网格中每个点的高度，Lx和Lz分别表示立方体网格在x方向和z方向上的大小，fr
GPU_total_time = 0  # 记录GPU计算的总时间
```

计算二维平面上一个给定波矢对应的海面高度h_0值，其中包括对随机数的使用以及对风速功率谱的计算。

In [ ]:

```
@cuda.jit(device=True)
def h0_cuda(k, conjugate, random):  # 表示波矢、是否取共轭以及用于生成随机数的状态数组
    epsilon_real = random[cuda.grid(1)]
    epsilon_imag = random[cuda.grid(1) + N]
    k_length = cmath.sqrt(k[0]**2 + k[1]**2)

    if k_length == 0:
        k_unit = (0, 0)
    else:
        k_unit = (k[0]/k_length, k[1]/k_length)

    Dw_length = cmath.sqrt(Dw[0]**2 + Dw[1]**2)
    Dw_unit = (Dw[0]/Dw_length, Dw[1]/Dw_length)
    L = Vw**2/g

    kDw_length = abs(k_unit[0] * Dw_unit[0] + k_unit[1] * Dw_unit[1])
    if k_length == 0:
        Ph = 0
    else:
        Ph = cmath.sqrt(A * cmath.exp(-1/((k_length*L)**2)) / (k_length**4) * (kDw_


    result = 1/cmath.sqrt(2) * float(epsilon_real) * Ph +  1/cmath.sqrt(2) * float
    if conjugate:
        result = 1/cmath.sqrt(2) * float(epsilon_real) * Ph - 1/cmath.sqrt(2) * fl

    return result
```

于计算一个给定时间和波矢对应的海面高度h值，其中包括对随机数的使用以及对初始海面高度h_0值的计算。

In [ ]:

```
@cuda.jit(device=True)
def h_cuda(k, t,  random_0, random_1):
    k_length = cmath.sqrt(k[0]**2 + k[1]**2)
    w = cmath.sqrt(g*k_length)
    e_iwkt = cmath.exp(w * t * 1j)
    e_neg_iwkt = cmath.exp(-w * t * 1j)
    neg_k = (-k[0], -k[1])
    h = h0_cuda(k, False, random_0) * e_iwkt + h0_cuda(neg_k, True, random_1) * e_
    return h
```

计算每个位置对应的海面高度h值，并将结果存储在数组中。函数对数组中每个位置进行迭代，并使用h_cuda函数计算对应位置的h值。

In [ ]:

```
@cuda.jit
def h_kernal(array, t, random_0, random_1):
    pos = cuda.grid(1)
    X_index = pos / N
    Z_index = pos % N
    X_value = X_index - int(N/2)
```

```
    Z_value = Z_index - int(N/2)
    k = (2 * math.pi * X_value / Lx, 2 * math.pi *  Z_value / Lz)
    array[pos] = h_cuda(k, t,random_0, random_1)
```

定义了一个DFT（离散傅里叶变换）函数，输入为一维数组x，输出为x的离散傅里叶变换结果。

In [ ]:
```
def dft(x):
    length = int(x.shape[0])
    n = np.arange(length)
    k = z.reshape((length, 1)) # k and z are the same here, so we just use z
    M = np.exp(-2j * np.pi * k * n / length)
    return np.dot(M, x)
```

将两个长度为 N 的傅里叶变换结果组合成一个长度为 $2N$ 的傅里叶变换结果。

In [ ]:
```
def fft_combine(res):
    length = res.shape[0]
    fft1 = res[:int(length/2)]
    fft2 = res[int(length/2):]

    K = np.exp(-2j * np.pi * np.arange(length) / length)
    K_split1 = K[:int(length/2)]
    K_split2 = K[int(length/2):]
    return np.concatenate([fft1 +  K_split1 * fft2,
                          fft1 +  K_split2 * fft2])
```

In [ ]:
```
def workerID_bin2dec(workerID):
    workerID_dec = 0
    factor = 1
    for i in workerID[::-1]:
        workerID_dec += i * factor
        factor *= 2
    return workerID_dec
```

实现了一个向量形式的快速傅里叶变换（FFT），将输入的一维数组转换为其对应的频域表示形式。

In [ ]:
```
def fft_vector(x):
    length = x.shape[0]
    if np.log2(length) % 1 > 0:
        raise ValueError("Size Error")

    n = np.arange(2)
    k = n[:, None]
    M = np.exp(-2j * np.pi * n * k / 2)
    X = np.dot(M, x.reshape((2, -1)))
    while X.shape[0] < length:
        X_even = X[:, :int(X.shape[1] / 2)]
        X_odd = X[:, int(X.shape[1] / 2):]
        K = np.exp(-1j * np.pi * np.arange(X.shape[0]) / X.shape[0])[:, None]
        X = np.vstack([X_even + K * X_odd, X_even - K * X_odd])
    return X.ravel()
```

基于递归实现的快速傅里叶变换（FFT）算法，输入一个一维复数数组x，返回一个相同大小的数组，代表x的傅里叶变换结果。

In [ ]:
```
def fft_recursion(x):
    x = np.asarray(x, dtype=np.complex128)
```

```python
    length = x.shape[0]
    if np.log2(length) % 1 > 0:
        raise ValueError("Size Error")
    elif length <= 2:
        n = np.arange(length)
        k = n.reshape((length, 1))
        M = np.exp(-2j * np.pi * k * n / length)
        return np.dot(M, x)
    else:
        X_even = fft_recursion(x[::2])
        X_odd = fft_recursion(x[1::2])
        K = np.exp(-2j * np.pi * np.arange(length) / length)
        K_split1 = K[:int(length/2)]
        K_split2 = K[int(length/2):]
        return np.concatenate([X_even + K_split1 * X_odd,
                               X_even + K_split2 * X_odd])
```

用于并行计算FFT（快速傅里叶变换）的函数，通过分解 FFT 的计算任务并在多个进程中同时进行计算，以提高计算效率。使用了递归计算的方法，以及一些数据通信和组合技巧来将计算结果进行合并。

In [ ]:

```python
def parallel_FFT(x, myrank, num_of_workers):
    # it describe what should each work do to perform a parallel_fft

    # test output
    current_stage = 0
    max_stage = np.log2(num_of_workers)
    myrank_bin = "{0:b}".format(myrank)

    workerID = []
    for i in myrank_bin:
        workerID.append(int(i))
    while len(workerID) < max_stage:
        workerID = [0] + workerID

        # init results
    results = x
    for i in workerID:
        results = results[:, i::2]
    results = np.apply_along_axis(fft_recursion, 1, results)

    #figure out what stage it is and what should I do
    while current_stage < max_stage:
        lastbit = workerID[-1] # 0 means odd, 1 means even

        if lastbit == 0:
            workerID_dec = workerID_bin2dec(workerID)
            source = (workerID_dec + 1) * (2 ** current_stage)
            tag = myrank * 1000 + source # tag == dest * 1000 + source
            recv = comm.recv(source=source, tag=tag)
            results = np.concatenate((results,recv),axis=1)
            results = np.apply_along_axis(fft_combine, 1, results)
            workerID = workerID[:-1]
            current_stage += 1

        else:
            workerID_dec = workerID_bin2dec(workerID)
            dest = (workerID_dec - 1) * (2 ** current_stage)
            tag = dest * 1000 + myrank # tag == dest * 1000 + source
            comm.send(results, dest=dest, tag=tag)
            return None
    return results
```

定义H0值的计算方法，其中包括以下步骤：

1. 从正态分布中获取两个随机样本。
2. 计算k的长度并判断是否为0，如果不是，则计算k的单位向量。
3. 计算Dw的长度并计算单位向量。
4. 计算L，即海浪的长度。
5. 计算kDw的长度。
6. 计算Ph，其中包括将A和一个指数函数与常量相乘并除以k的四次方，然后乘以kDw的平方根。
7. 计算最终结果，其中包括将随机样本和Ph值相乘并乘以1/√2，然后将结果返回为一个复数。
8. 可以选择返回复数的共轭。

In [ ]:
```
def h0(k, conjugate = False):
    samples = np.random.normal(mu, sigma, 2)
    epsilon_real = samples[0]
    epsilon_imag = samples[1]
    k_length = math.sqrt(k[0]**2 + k[1]**2)

    if k_length == 0:
        k_unit = (0, 0)
    else:
        k_unit = (k[0]/k_length, k[1]/k_length)

    Dw_length = math.sqrt(Dw[0]**2 + Dw[1]**2)
    Dw_unit = (Dw[0]/Dw_length, Dw[1]/Dw_length)
    L = Vw**2/g

    kDw_length = abs(k_unit[0] * Dw_unit[0] + k_unit[1] * Dw_unit[1])
    if k_length == 0:
        Ph = 0
    else:
        Ph = math.sqrt(A * np.exp(-1/((k_length*L)**2)) / (k_length**4) * (kDw_len


    result = complex(1/math.sqrt(2) * epsilon_real * Ph, 1/math.sqrt(2) * epsilon_
    if conjugate:
        result = complex(1/math.sqrt(2) * epsilon_real * Ph, -1/math.sqrt(2) * eps

    return result
```

In [ ]:
```
def h(k, t):
    k_length = math.sqrt(k[0]**2 + k[1]**2)
    w = math.sqrt(g*k_length)
    e_iwkt = np.exp(w * t * 1j)
    e_neg_iwkt = np.exp(-w * t * 1j)
    neg_k = (-k[0], -k[1])
    h = h0(k, False) * e_iwkt + h0(neg_k, True) * e_neg_iwkt
    return h
```

In [ ]:
```
def make_term(m_or_n, x_or_z):
    return np.exp(2j * np.pi * (m_or_n + N/2)/N * x_or_z)
```

Brute Force算法（暴力算法），用于计算二维Fourier变换的离散近似值。

In [ ]:
```
def BF(X, Z, t):
    update_y = np.zeros((N, N))
    for x_index in range(X.shape[0]):
```

```
        for z_index in  range(Z.shape[0]):
            x_value = X[x_index]
            z_value = Z[z_index]
            res_H = 0
            v = (x_value * Lx / N, z_value * Lz / M)

            for k_x_index in range(X.shape[0]):
                for k_z_index in  range(Z.shape[0]):
                    k_x = X[k_x_index]
                    k_z = Z[k_z_index]

                    k = (2 * math.pi * k_x / Lx, 2 * math.pi * k_z / Lz)
                    k_dot_v = k[0]*v[0] + k[1]*v[1]
                    e_ikx = complex(math.cos(k_dot_v), math.sin(k_dot_v))

                    res_H += (h(k, t) * e_ikx)
            update_y[x_index, z_index] = res_H.real
    return update_y
```

基于暴力算法的二维傅里叶变换，用于计算二维场的频域表示。

In [ ]:
```
def bf_vector(X, Z, t):
    h_hat = np.zeros((N+1, N+1),dtype=np.complex_)
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            k = (2 * np.pi * X[x_index] / Lx, 2 * np.pi *  Z[z_index] / Lz)
            h_hat[x_index][z_index] = h(k, t)

    update_y = np.zeros((Lx+1, Lz+1))
    for x_index in range(X.shape[0]):
        for z_index in  range(Z.shape[0]):
            factor = (-1)**(int(X[x_index]) + int(Z[z_index]))
            x_value = X[x_index]
            z_value = Z[z_index]
            res_H = 0
            for n in range(N):
                k_x = X[n]
                e_2pinxiN = np.exp(2j * np.pi * (k_x + N/2)/N * x_value)

                subsum = 0
                for m in range(M):
                    k_z = Z[m]
                    e_2pimziN = np.exp(2j * np.pi * (k_z + N/2)/N * z_value)

                    k = (2 * np.pi * k_x / Lx, 2 * np.pi *  k_z / Lz)
                    h_hat = h(k, t)
                    subsum += e_2pimziN * h_hat

                res_H += e_2pinxiN * subsum
            res_H *= factor
            update_y[x_index, z_index] = abs(res_H)

    return update_y
```

优化版的brute force algorithm，使用预处理技巧来提高效率。

In [ ]:
```
def bf_vector_precalch(X, Z, t):
    h_hat = np.zeros((N+1, N+1),dtype=np.complex128)
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            k = (2 * np.pi * X[x_index] / Lx, 2 * np.pi *  Z[z_index] / Lz)
            h_hat[x_index][z_index] = h(k, t).real
```

```
    update_y = np.zeros((N+1, N+1))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            x_value = X[x_index]
            z_value = Z[z_index]
            z_hat = make_term(Z.reshape((N+1, 1)), z_value)
            x_hat = make_term(X, x_value)
            # shape check
            factor = (-1) ** (int(X[x_index]) + int(Z[z_index]))
            res_H = 0
            for n in range(N):
                subsum = 0
                for m in range(M):
                    subsum += z_hat[m] * h_hat[n][m]
                res_H += x_hat[n] * subsum
            update_y[x_index][z_index] =  res_H.real*factor
    return update_y
```

通过预先计算和优化矩阵运算来提高计算效率

In [ ]:

```
def bf_vector_precalch_dot(X, Z, t):
    h_hat = np.zeros((N, N),dtype=np.complex128)

    # this part can be parallelize
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            k = (2 * np.pi * X[x_index] / Lx, 2 * np.pi *  Z[z_index] / Lz)
            h_hat[x_index][z_index] = h(k, t).real # question: what difference?

    # this part can be make faster
    update_y = np.zeros((N, N))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            x_value = X[x_index]
            z_value = Z[z_index]
            z_hat = make_term(Z.reshape((N, 1)), z_value)
            x_hat = make_term(X, x_value)
            factor = (-1) ** (int(X[x_index]) + int(Z[z_index]))
            update_y[x_index][z_index] = np.dot(x_hat, np.dot(h_hat, z_hat)).real*
    return update_y
```

执行二维离散傅里叶变换(DFT)，对输入的二维矩阵进行频域变换，并返回变换后的结果。

In [ ]:

```
def DFT(X, Z, t):
    h_hat = np.zeros((N, N),dtype=np.complex128)
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            k = (2 * np.pi * X[x_index] / Lx, 2 * np.pi *  Z[z_index] / Lz)
            h_hat[x_index][z_index] = h(k, t)

    factor = np.zeros((N, N))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            factor[x_index][z_index] = (-1) ** (int(X[x_index]) + int(Z[z_index]))

   # this part can be make faster
    update_y = np.zeros((N, N))
    # for z_index in range(Z.shape[0]):
    x_value = X[x_index]
    z_value = Z[z_index]
    N_array = np.arange(N)
```

```
    M_array = np.arange(N).reshape((N, 1))

    # z_hat = np.exp(2j * np.pi * M_array/N * z_value)
    # firstFFT = np.dot(h_hat, z_hat)
    firstFFT = np.apply_along_axis(dft, 1, h_hat)
    firstFFT = firstFFT.T
    secondFFT = np.apply_along_axis(dft, 1, firstFFT)
    update_y = secondFFT.T.real

    return np.multiply(update_y, factor)
```

使用FFT算法来计算二维数组的傅里叶变换的函数，提高计算速度

In [ ]:
```
def FFT(X, Z, t):
    h_hat = np.zeros((N, N),dtype=np.complex128)
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            k = (2 * np.pi * X[x_index] / Lx, 2 * np.pi *  Z[z_index] / Lz)
            h_hat[x_index][z_index] = h(k, t)

    factor = np.zeros((N, N))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            factor[x_index][z_index] = (-1) ** (int(X[x_index]) + int(Z[z_index]))

   # this part can be make faster
    update_y = np.zeros((N, N))
    x_value = X[x_index]
    z_value = Z[z_index]
    N_array = np.arange(N)
    M_array = np.arange(N).reshape((N, 1))

    # z_hat = np.exp(2j * np.pi * M_array/N * z_value)
    # firstFFT = np.dot(h_hat, z_hat)
    firstFFT = np.apply_along_axis(fft_recursion, 1, h_hat)
    firstFFT = firstFFT.T
    secondFFT = np.apply_along_axis(fft_recursion, 1, firstFFT)
    update_y = secondFFT.T.real

    return np.multiply(update_y, factor)
```

使用MPI并行计算的FFT函数，实现了分布式计算的快速傅里叶变换算法，可以加速傅里叶变换的计算过程。

In [ ]:
```
def FFT_P(X, Z, t, rank, num_of_workers):
    h_hat = np.zeros((N, N),dtype=np.complex128)
    if rank == 0:
        # do this with cuda
        for x_index in range(X.shape[0]):
            for z_index in range(Z.shape[0]):
                k = (2 * np.pi * X[x_index] / Lx, 2 * np.pi *  Z[z_index] / Lz)
                h_hat[x_index][z_index] = h(k, t)
    h_hat = comm.bcast(h_hat, root=0)

    factor = np.zeros((N, N))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            factor[x_index][z_index] = (-1) ** (int(X[x_index]) + int(Z[z_index]))

   # this part can be make faster
    update_y = np.zeros((N, N))
    # firstFFT = np.apply_along_axis(parallel_FFT, 1, h_hat, rank, num_of_workers)
```

```python
    firstFFT = parallel_FFT(h_hat, rank, num_of_workers)
    firstFFT = comm.bcast(firstFFT, root=0)
    firstFFT = firstFFT.T

    # distribute this results to others
    secondFFT = parallel_FFT(firstFFT, rank, num_of_workers)
    # secondFFT = np.apply_along_axis(parallel_FFT, 1, firstFFT, rank, num_of_worke
    if rank == 0:
        update_y = secondFFT.T.real
        result = np.multiply(update_y, factor)
        return result
    else:
        return None
```

使用GPU加速计算FFT，并返回计算结果和GPU计算时间。

In [ ]:

```python
def FFT_C(X, Z, t):
    h_hat = np.zeros((N, N),dtype=np.complex128)
    # xoroshiro128p_normal_float32 is problematic, use precalculated random
    random_0 = np.random.normal(mu, sigma, N * N * 2)
    random_1 = np.random.normal(mu, sigma, N * N * 2)
    h_hat = np.zeros(N * N,dtype=np.complex128)
    GPU_start = timer()
    h_kernal[N, N](h_hat, t, random_0, random_1)
    GPU_end = timer()
    GPU_time = GPU_end - GPU_start
    h_hat = h_hat.reshape((N, N))

    factor = np.zeros((N, N))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            factor[x_index][z_index] = (-1) ** (int(X[x_index]) + int(Z[z_index]))

   # this part can be make faster
    update_y = np.zeros((N, N))
    x_value = X[x_index]
    z_value = Z[z_index]
    N_array = np.arange(N)
    M_array = np.arange(N).reshape((N, 1))

    # z_hat = np.exp(2j * np.pi * M_array/N * z_value)
    # firstFFT = np.dot(h_hat, z_hat)
    firstFFT = np.apply_along_axis(fft_recursion, 1, h_hat)
    firstFFT = firstFFT.T
    secondFFT = np.apply_along_axis(fft_recursion, 1, firstFFT)
    update_y = secondFFT.T.real

    return np.multiply(update_y, factor), GPU_time
```

实现了一个并行的快速傅里叶变换算法，用于加速信号处理，并使用MPI通信库实现分布式计算。

In [ ]:

```python
def FFT_PC(X, Z, t, rank, num_of_workers):
    h_hat = np.zeros((N, N),dtype=np.complex128)
    # xoroshiro128p_normal_float32 is problematic, use precalculated random
    random_0 = np.random.normal(mu, sigma, N * N * 2)
    random_1 = np.random.normal(mu, sigma, N * N * 2)
    if rank == 0:
        h_hat = np.zeros(N * N,dtype=np.complex128)
        GPU_start = timer()
        h_kernal[N, N](h_hat, t, random_0, random_1)
        GPU_end = timer()
```

```
        GPU_time = GPU_end - GPU_start
        h_hat = h_hat.reshape((N, N))

    h_hat = comm.bcast(h_hat, root=0)


    # factor array is simple so every worker should calculate their own one
    factor = np.zeros((N, N))
    for x_index in range(X.shape[0]):
        for z_index in range(Z.shape[0]):
            factor[x_index][z_index] = (-1) ** (int(X[x_index]) + int(Z[z_index]))

   # this part can be make faster
    update_y = np.zeros((N, N))
    # firstFFT = np.apply_along_axis(parallel_FFT, 1, h_hat, rank, num_of_workers)
    firstFFT = parallel_FFT(h_hat, rank, num_of_workers)
    firstFFT = comm.bcast(firstFFT, root=0)
    firstFFT = firstFFT.T

    # distribute this results to others
    secondFFT = parallel_FFT(firstFFT, rank, num_of_workers)
    # secondFFT = np.apply_along_axis(parallel_FFT, 1, firstFFT, rank, num_of_worke
    if rank == 0:
        update_y = secondFFT.T.real
        result = np.multiply(update_y, factor)
        return result, GPU_time
    else:
        return None, 0
```

并行计算程序的主程序，根据传入的命令行参数选择不同的计算方法，对输入的数据进行计算并输出结果，最后通过 matplotlib 绘制出 3D 图形动画。同时该程序还使用了 MPI 库实现并行化计算。

In [ ]:

```
if __name__ == '__main__':
    args = sys.argv[1:]
    method = args[0]

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    GPU_total_time = 0

    # compute and draw
    start = time.time()
    for t in range(frame):
        if method == "BF":
            yarray[:,:,t] = BF(x, z, t)
        if method == "DFT":
            yarray[:,:,t] = DFT(x, z, t)
        if method == "FFT":
            yarray[:,:,t] = FFT(x, z, t)
        if method == "FFT_P":
            if comm.size == 1:
                # reduced to single score FFT
                yarray[:,:,t] = FFT(x, z, t)
            else:
                yarray[:,:,t] = FFT_P(x, z, t, rank, comm.size)
        if method == "FFT_PC":
            if comm.size == 1:
                # reduced to single score FFT
                result, GPU_time = FFT_C(x, z, t)
                yarray[:,:,t] = result
                GPU_total_time += GPU_time
```

```
        else:
            result, GPU_time = FFT_PC(x, z, t, rank, comm.size)
            yarray[:,:,t] = result
            GPU_total_time += GPU_time
end = time.time()

if rank == 0:
    print("computation time needed:", end - start)
    print("GPU time needed:", GPU_total_time)
    fig = plt.figure(figsize=(12,12))
    ax = fig.add_subplot(111, projection='3d')
    x_mesh = x_mesh.flatten()
    z_mesh = z_mesh.flatten()
    def update_plot(frame_number, yarray, plot):
        plot[0].remove()
        plot[0] = ax.plot_trisurf(x_mesh, z_mesh, yarray[:,:,frame_number].fla
    plot = [ax.plot_trisurf(x_mesh, z_mesh, yarray[:,:,0].flatten(), linewidth
    ax.set_zlim(-25, 25)
    ani = animation.FuncAnimation(fig, update_plot, frame, fargs=(yarray, plot
```