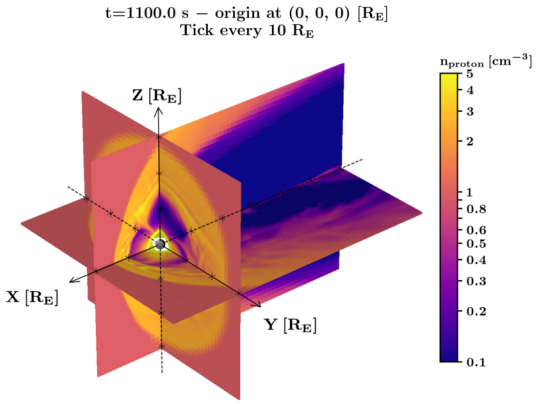
Inno4Scale

April 29, 2024

1 Vlasiator - A Global Hybrid-Vlasov Simulation Model

Vlasiator [@palmroth2018] is an open-source simulation software used to model the behavior of plasma in the Earth's magnetosphere, a region of space where the solar wind interacts with the Earth's magnetic field. Vlasiator models collisionless space plasma dynamics by solving the 6-dimensional Vlasov equation, using a hybrid-Vlasov approach. It uses a 3D Cartesian grid in real space, with each cell storing another 3D Cartesian grid in velocity space. The velocity mesh contained in each spatial cell in the simulation domain has been represented so far by a sparse grid approach, fundamentally based on an associative container such as a key-value hashtable. Storing a 3D VDF at every spatial cells increases the memory requirements exponentially both during runtime and for storing purposes. Our proposal revolves around developing innovative solutions to compressing the VDFs during runtmime.



2 VDF Compression

2.1 Initialization

Let's read in a vdf from a sample file and see what that looks like.

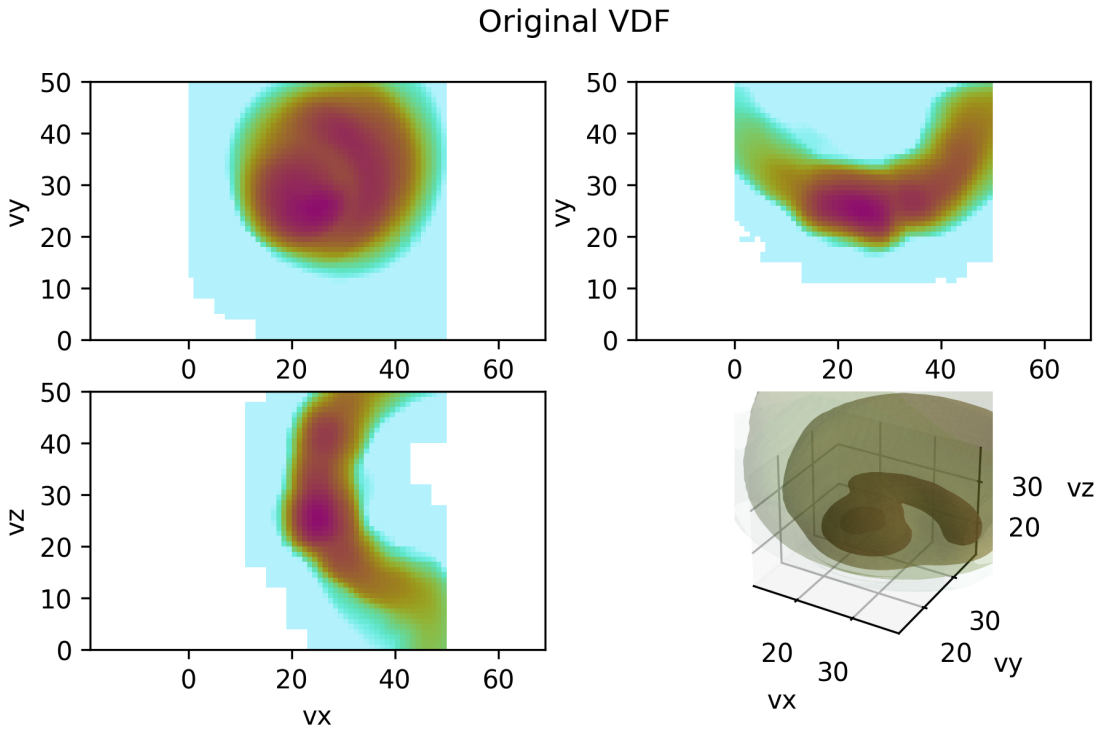
[3]:
```
import sys,os
import warnings
warnings.filterwarnings('ignore')
# sys.path.append('/home/mjalho/analysator')
import tools as project_tools
import numpy as np
import matplotlib as mpl
```

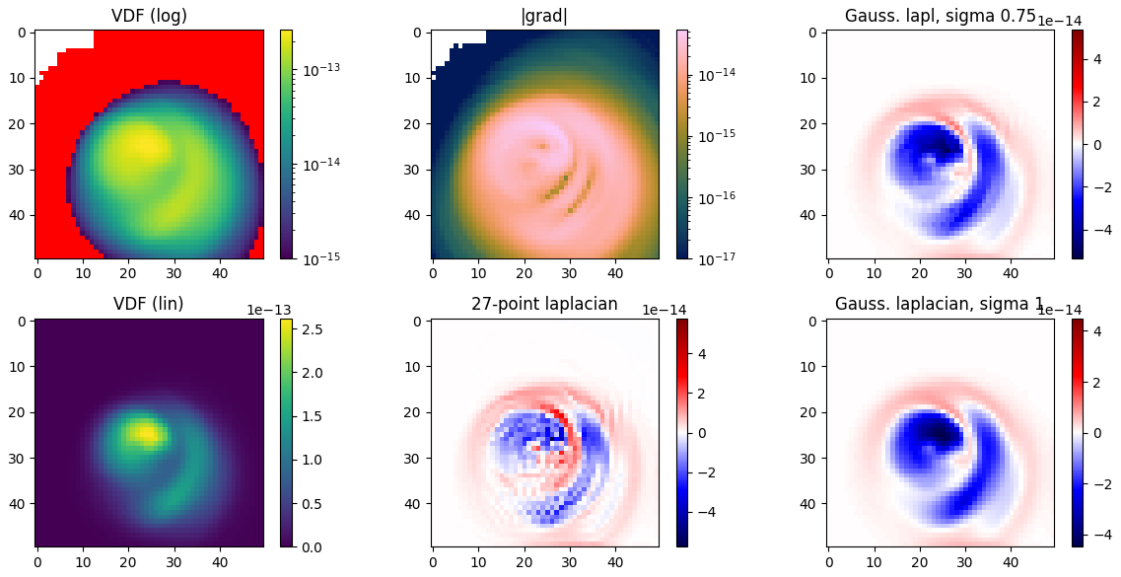
```
import matplotlib.pyplot as plt
mpl.rcParams.update(mpl.rcParamsDefault)
import matplotlib.colors as colors
# plt.rcParams['figure.figsize'] = [7, 7]
import ctypes
import pyzfp,zlib
import mlp_compress
from skimage import measure
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

import pytools
```

[4]:
```
file="assets/bulk1.0001280.vlsv";cid=356780649;
#Read the VDF into a 3D uniform mesh and plot it
vdf=project_tools.extract_vdf(file,cid,25)
# np.save("sample_vdf.bin",np.array(vdf,dtype=np.double));
np.array(vdf,dtype=np.double).tofile("sample_vdf.bin")
nx,ny,nz=np.shape(vdf)
print(f"VDF shape = {np.shape(vdf)}")
fig = plt.figure(figsize=[7,4], dpi=300);
ax1 = plt.subplot(221)
ax2 = plt.subplot(222)
ax3 = plt.subplot(223)
ax4 = plt.subplot(224, projection='3d')
# , ax2, ax3) = plt.subplots(1, 3)


cmap = 'viridis'
cmap = 'hawaii_r'
norm = colors.LogNorm(vmin=1e-15,vmax=3e-13)
im1 = ax1.pcolormesh(vdf[:,:,nz//2],norm=norm,rasterized=False,cmap=cmap)
# im1 = ax1.imshow(vdf[:,:,nz//2],norm=norm,cmap=cmap, origin='lower')
im2 = ax2.pcolormesh(vdf[nx//2,:,:].T,norm=norm,rasterized=False,cmap=cmap)
im3 = ax3.pcolormesh(vdf[:,ny//2,:],norm=norm,rasterized=False,cmap=cmap)
ax1.axis('equal')
ax1.set_xlabel('vx')
ax1.set_ylabel('vy')
ax2.axis('equal')
ax2.set_xlabel('vz')
ax2.set_ylabel('vy')
ax3.axis('equal')
ax3.set_xlabel('vx')
ax3.set_ylabel('vz')

level = 1e-14
for level in [3e-15,1e-14,3e-14,1e-13,2e-13]:
    verts, faces, normals, values = measure.marching_cubes(vdf, level)
```

```
    mesh = Poly3DCollection(verts[faces], shade=True, facecolors=mpl.
 ↪colormaps[cmap](norm(level)), alpha = norm(level)**2)
    ax4.add_collection3d(mesh)

ax4.set_xlim(nx/4,3*nx/4)
ax4.set_ylim(ny/4,3*ny/4)
ax4.set_zlim(nz/4,3*nz/4)
ax4.set_xlabel('vx')
ax4.set_ylabel('vy')
ax4.set_zlabel('vz')

# cax = fig.add_axes([0.7,0.1,0.2,0.05])
# fig.colorbar(im1, cax=cax, location='bottom')
fig.suptitle("Original VDF")
plt.show()

project_tools.plot_vdf_discrete_laplacians(vdf)
```

```
Found population proton
Getting offsets for population proton
VDF shape = (50, 50, 50)
```





The vdf shown above is sampled on a uniform 3D velocity mesh and contains 64bit floating point numbers that represent the phase space density. We can calculate the total size of this VDF is bytes using `sys.getsizeof(vdf)`.

[5]:
```
vdf_mem=sys.getsizeof(vdf)
num_stored_elements=len(vdf[vdf>1e-15])
print(f"VDF takes {vdf_mem} B.")
```

```
VDF takes 500144 B.
```

Now in Vlasiator we have countlesss VDFs since there is one per spatial cell. It would be great if we could compress them efficiently.

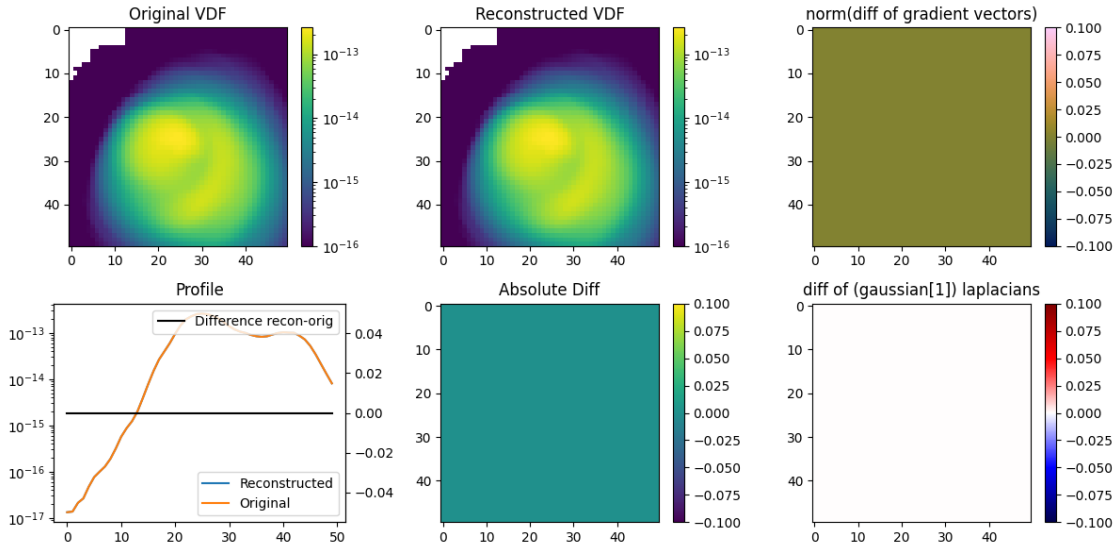
2.2 Compression algorithms

2.2.1 Zlib: lossless floating point compression

We can try to do so by using zlib which is a form of lossless compression. The reconstruction is accurate, by definition, but the compression ratio is small.

[6]:
```
compressed_vdf = zlib.compress(vdf)
compressed_vdf_mem=len(compressed_vdf)
compression_ratio=vdf_mem/compressed_vdf_mem
print(f"Achieved compression ratio using zlib= {round(compression_ratio,2)}.")
decompressed_vdf = zlib.decompress(compressed_vdf)
recon = np.frombuffer(decompressed_vdf, dtype=vdf.dtype).reshape(vdf.shape)
project_tools.plot_vdfs(vdf,recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Achieved compression ratio using zlib= 1.54.
```



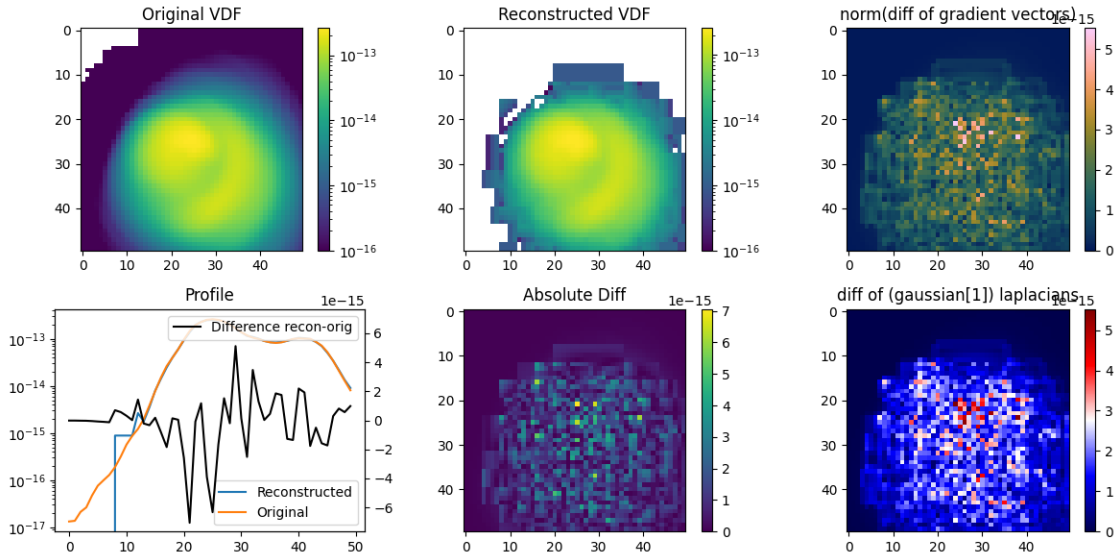

```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (0.0, 0.0,
'0.e+00', '0.e+00') %.
L1,L2 rNorms= (0.0, 0.0).
```

2.2.2 zfp: lossy floating point compression

We can use a lossy compression method like zfp[@zfp] to get even higher compression ratios.

[7]:
```
"""
Compresses a VDF  using ZFP (Zstandard Compressed FP)
Input:VDF - numpy array
Output: recon (Reconstructed VDF) - numpy array
"""
tolerance = 1e-13
compressed_vdf = pyzfp.compress(vdf, tolerance=tolerance)
compressed_vdf_mem=len(compressed_vdf)
compression_ratio=vdf_mem/compressed_vdf_mem
print(f"Achieved compression ratio using zfp= {round(compression_ratio,2)}.")
recon = pyzfp.decompress(compressed_vdf,vdf.shape,vdf.dtype,tolerance)
project_tools.plot_vdfs(vdf,recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Achieved compression ratio using zfp= 87.32.
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (0.083, 0.0,
'2.506e+00', '2.490e+00') %.
L1,L2 rNorms= (0.046, 0.032).
```

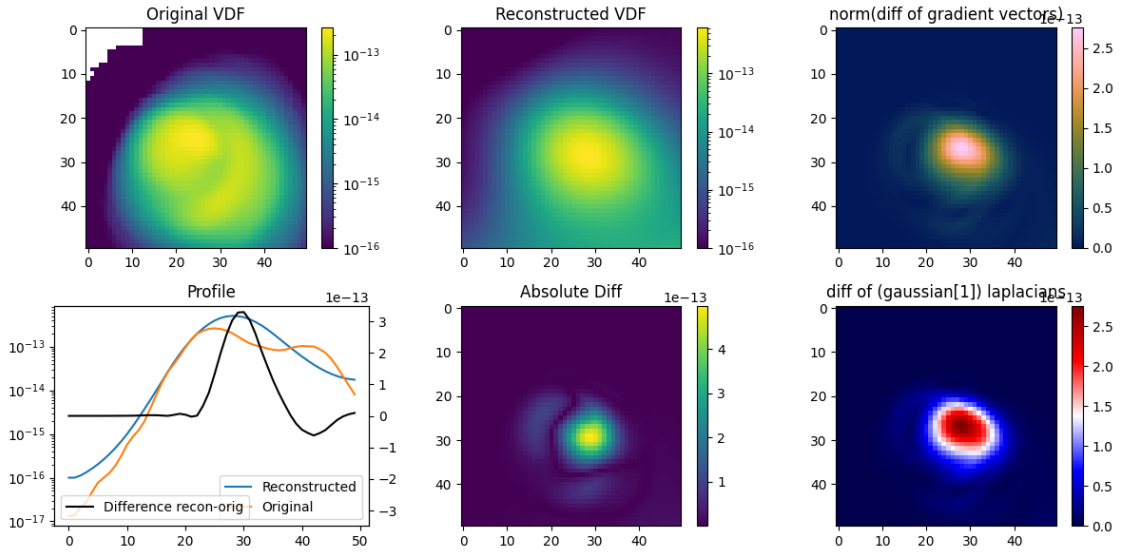
2.2.3 Multilevel Perceptron (MLP)

This is based on [@park2019].

[8]:
```
"""
Compresses a VDF using an MLP (Multilayer Perceptron).
 Input: "sample_vdf.bin" - Binary file containing the VDF data
        order - Order of the fourier features
        epochs - Number of training epochs for the MLP model
        n_layers - Number of layers in the MLP model
        n_neurons - Number of neurons in each layer of the MLP model
 Output: recon (Reconstructed VDF) - NumPy array representing the reconstructed␣
  ↪volume data
"""
order=0
epochs=20
n_layers=4
n_neurons=25
nx,ny,nz=np.shape(vdf)
recon=mlp_compress.compress_mlp("sample_vdf.
  ↪bin",order,epochs,n_layers,n_neurons,nx)
recon=np.array(recon,dtype=np.double)
recon= np.reshape(recon,np.shape(vdf),order='C')
```

```
project_tools.plot_vdfs(vdf,recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Reading VDF from sample_vdf.bin
Cost at epoch 0 is 5.4913
Cost at epoch 1 is 0.5205
Cost at epoch 2 is 0.4788
Cost at epoch 3 is 0.4622
Cost at epoch 4 is 0.4682
Cost at epoch 5 is 0.4689
Cost at epoch 6 is 0.4521
Cost at epoch 7 is 0.4543
Cost at epoch 8 is 0.4631
Cost at epoch 9 is 0.4497
Cost at epoch 10 is 0.4528
Cost at epoch 11 is 0.4494
Cost at epoch 12 is 0.4405
Cost at epoch 13 is 0.4546
Cost at epoch 14 is 0.4428
Cost at epoch 15 is 0.4382
Cost at epoch 16 is 0.4236
Cost at epoch 17 is 0.4002
Cost at epoch 18 is 0.3445
Cost at epoch 19 is 0.2113
Bytes serialized 11456/11456.
Done in 14.06 s.  Compression ratio = 87.66x .
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
```

```
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (36.642, 2.439,
'1.127e+02', '7.305e+01') %.
L1,L2 rNorms= (0.98, 0.77).
```

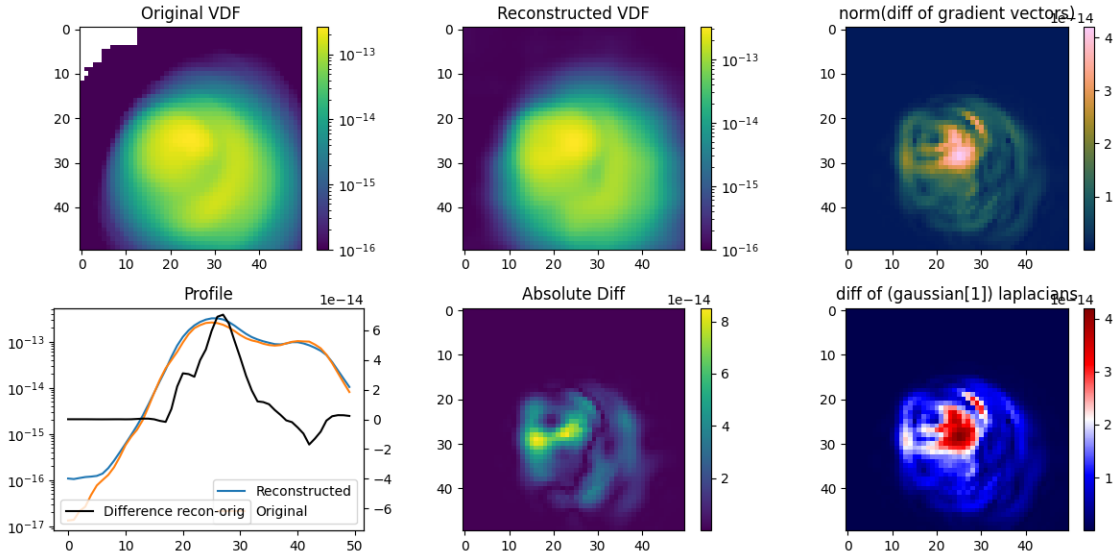
2.2.4 MLP with Fourier features

We will compress the VDF using an MLP with Fourier Features, which significantly improves MLP performance [@2020fourier].

[9]:
```
"""
Compresses a VDF using an MLP (Multilayer Perceptron).
 Input: "sample_vdf.bin" - Binary file containing the VDF data
        order - Order of the fourier features
        epochs - Number of training epochs for the MLP model
        n_layers - Number of layers in the MLP model
        n_neurons - Number of neurons in each layer of the MLP model
 Output: recon (Reconstructed VDF) - NumPy array representing the reconstructed␣
  ↪volume data
"""
order=16
epochs=20
n_layers=4
n_neurons=25
nx,ny,nz=np.shape(vdf)
recon=mlp_compress.compress_mlp("sample_vdf.
  ↪bin",order,epochs,n_layers,n_neurons,nx)
recon=np.array(recon,dtype=np.double)
recon= np.reshape(recon,np.shape(vdf),order='C')
project_tools.plot_vdfs(vdf,recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Reading VDF from sample_vdf.bin
Cost at epoch 0 is 3.9385
Cost at epoch 1 is 0.0447
Cost at epoch 2 is 0.0280
Cost at epoch 3 is 0.0218
Cost at epoch 4 is 0.0175
Cost at epoch 5 is 0.0148
Cost at epoch 6 is 0.0133
Cost at epoch 7 is 0.0123
Cost at epoch 8 is 0.0110
Cost at epoch 9 is 0.0105
Cost at epoch 10 is 0.0103
Cost at epoch 11 is 0.0098
Cost at epoch 12 is 0.0097
Cost at epoch 13 is 0.0095
Cost at epoch 14 is 0.0097
Cost at epoch 15 is 0.0092
```

```
Cost at epoch 16 is 0.0088
Cost at epoch 17 is 0.0090
Cost at epoch 18 is 0.0091
Cost at epoch 19 is 0.0088
Bytes serialized 30656/30656.
Done in 30.81 s.  Compression ratio = 32.67x .
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (2.674, 1.074,
'1.514e+01', '1.598e+01') %.
L1,L2 rNorms= (0.161, 0.224).
```

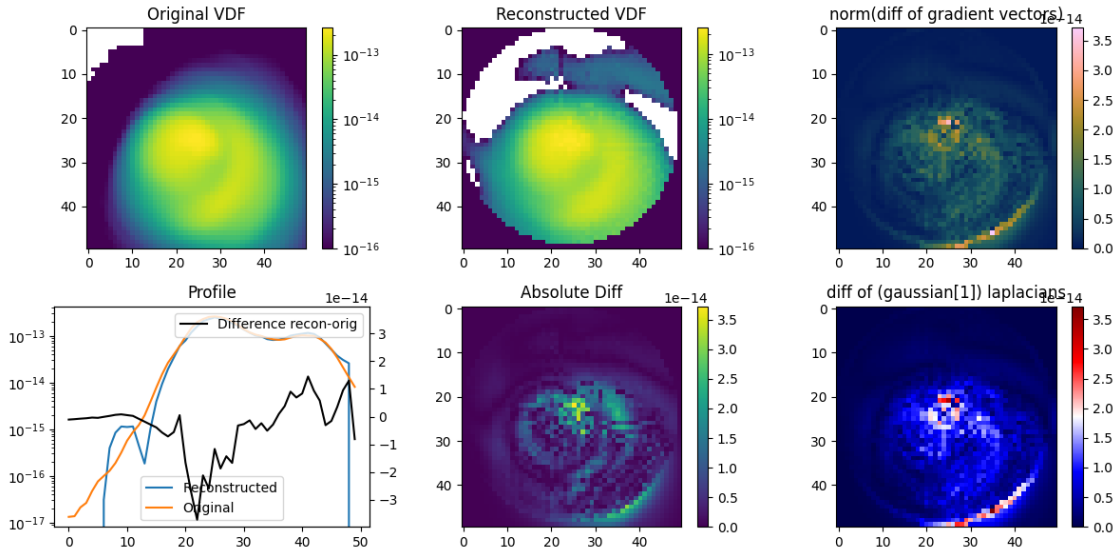
2.2.5 Spherical Harmonic Decomposition

Spherical harmonics have been suggested as a usable approximation for VDFs in our subject domain [@vinas_gurgiolo_2009]. Here we investigate if the method can be employed as a compression method. This prototype should be improved by using e.g., the Misner [@misner_2004] method.

[10]:
```
"""
Compresses a VDF using a spherical harmonic decompostion
Input: "sample_vdf.bin" - Binary file containing the VDF data
       degree - Degree of the spherical harmonic decomposition (l)
Output: recon (Reconstructed VDF) - NumPy array representing the reconstructed␣
 ↪volume data
"""

degree=10
nx,ny,nz=np.shape(vdf)
```

```
recon=mlp_compress.compress_sph("sample_vdf.bin",degree,nx)
recon=np.array(recon,dtype=np.double)
recon= np.reshape(recon,np.shape(vdf),order='C')
project_tools.plot_vdfs(vdf,recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Reading VDF from sample_vdf.bin
Compression ratio = 41.322315x .
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (25.905, 6.178,
'1.180e+02', '8.576e+01') %.
L1,L2 rNorms= (0.368, 0.318).
```

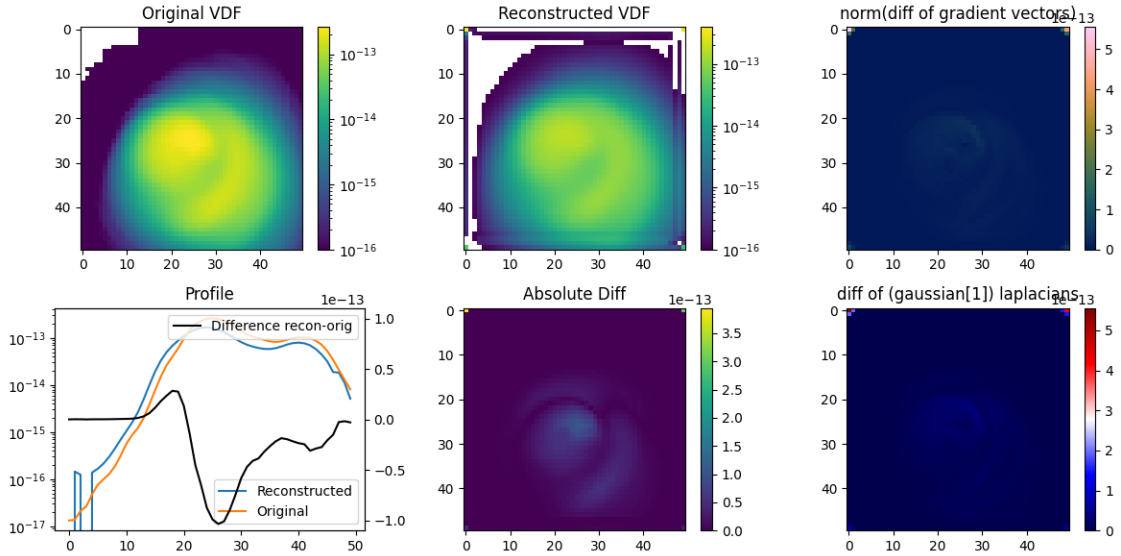
2.2.6 Convolutional Neural Network (CNN)

[11]:
```
"""
Function: train_and_reconstruct

Description:
This function takes an input array and trains a Convolutional Neural Network␣
↪(CNN) model to reconstruct the input array.
It uses Mean Squared Error (MSE) loss and the Adam optimizer for training.

Inputs:
- input_array (numpy array): The input array to be reconstructed.
- num_epochs (int, optional): The number of training epochs.
- learning_rate (float, optional): The learning rate for the Adam optimize.
```

```
Outputs:
    Reconstructed vdf array
    Size of model in bytes
"""

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv3d(1, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv3d(16, 32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv3d(32, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv3d(64, 1, kernel_size=3, padding=1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.conv4(x)
        return x

def train_and_reconstruct(input_array, num_epochs=30, learning_rate=0.001):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    input_tensor = torch.tensor(input_array, dtype=torch.float32).unsqueeze(0).
 ↪unsqueeze(0).to(device)  # Add batch and channel dimensions, move to device
    model = CNN().to(device)  # Move model to device
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    for epoch in range(num_epochs):
        optimizer.zero_grad()
        output_tensor = model(input_tensor)
        loss = criterion(output_tensor, input_tensor)
        loss.backward()
        optimizer.step()

        if (epoch+1) % 100== 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

    with torch.no_grad():
        output_tensor = model(input_tensor)
    reconstructed_array = output_tensor.squeeze(0).squeeze(0).cpu().numpy()
```

```
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()
    size = (param_size + buffer_size)
    return reconstructed_array, size


vdf_temp=vdf.copy()
vdf_temp[vdf_temp<1e-16]=1e-16
vdf_temp = np.log10(vdf_temp)
input_array=vdf_temp
recon,total_size= train_and_reconstruct(input_array,100)
recon = 10 ** recon
recon[recon <= 1e-16] = 0
vdf_size=nx*ny*nz*8
print(f"Compresion achieved using a CNN = {round(vdf_size/total_size,2)}")
project_tools.plot_vdfs(vdf,recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Epoch [100/100], Loss: 0.1088
Compresion achieved using a CNN = 3.5
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (192.513,
10.353, '2.e+02', '2.e+02') %.
```

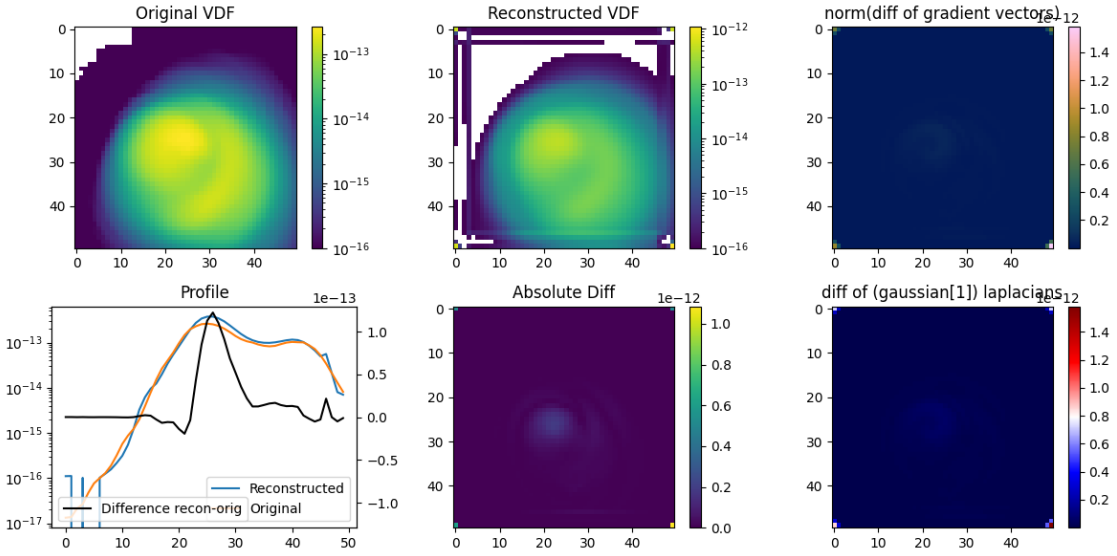
```
L1,L2 rNorms= (51.932, 1.0).
```

CNN with minibatches Here we still use a CNN but this time we use minibatch training and batch normalization layers to try and improve performance.

[12]:
```
"""
Function: train_and_reconstruct

Description:
This function takes an input array and trains a Convolutional Neural Network␣
 ↪(CNN) model to reconstruct the input array.
It uses Mean Squared Error (MSE) loss and the Adam optimizer for training.

Inputs:
- input_array (numpy array): The input array to be reconstructed.
- num_epochs (int, optional): The number of training epochs.
- learning_rate (float, optional): The learning rate for the Adam optimizer
Outputs:
    Reconstructed vdf array
    Size of model in bytes
"""
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv3d(1, 16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm3d(16)
        self.conv2 = nn.Conv3d(16, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm3d(32)
        self.conv3 = nn.Conv3d(32, 64, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm3d(64)
        self.conv4 = nn.Conv3d(64, 1, kernel_size=3, padding=1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.relu(self.bn3(self.conv3(x)))
        x = self.conv4(x)
        return x

def train_and_reconstruct(input_array, num_epochs=30, learning_rate=0.001,␣
 ↪batch_size=32):
```

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    input_tensor = torch.tensor(input_array, dtype=torch.float32).unsqueeze(0).
 ↪unsqueeze(0).to(device)  # Move input tensor to device
    model = CNN().to(device)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    for epoch in range(num_epochs):
        for i in range(0, input_tensor.size(0), batch_size):
            optimizer.zero_grad()
            batch_input = input_tensor[i:i+batch_size]
            output_tensor = model(batch_input)
            loss = criterion(output_tensor, batch_input)
            loss.backward()
            optimizer.step()

    with torch.no_grad():
        output_tensor = model(input_tensor)
    reconstructed_array = output_tensor.squeeze(0).squeeze(0).cpu().numpy()

    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()
    size = (param_size + buffer_size)
    return reconstructed_array, size

vdf_temp = vdf.copy()
vdf_temp[vdf_temp < 1e-16] = 1e-16
vdf_temp = np.log10(vdf_temp)
input_array = vdf_temp
recon, total_size = train_and_reconstruct(input_array, 100)

recon = 10 ** recon
recon[recon <= 1e-16] = 0
vdf_size = nx * ny * nz * 8
print(f"Compression achieved using a CNN = {round(vdf_size / total_size, 2)}")

project_tools.plot_vdfs(vdf, recon)
project_tools.print_comparison_stats(vdf, recon)
```

```
Compression achieved using a CNN = 3.48
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (197.508,
28.721, '2.e+02', '2.e+02') %.
L1,L2 rNorms= (158.657, 1.0).
```

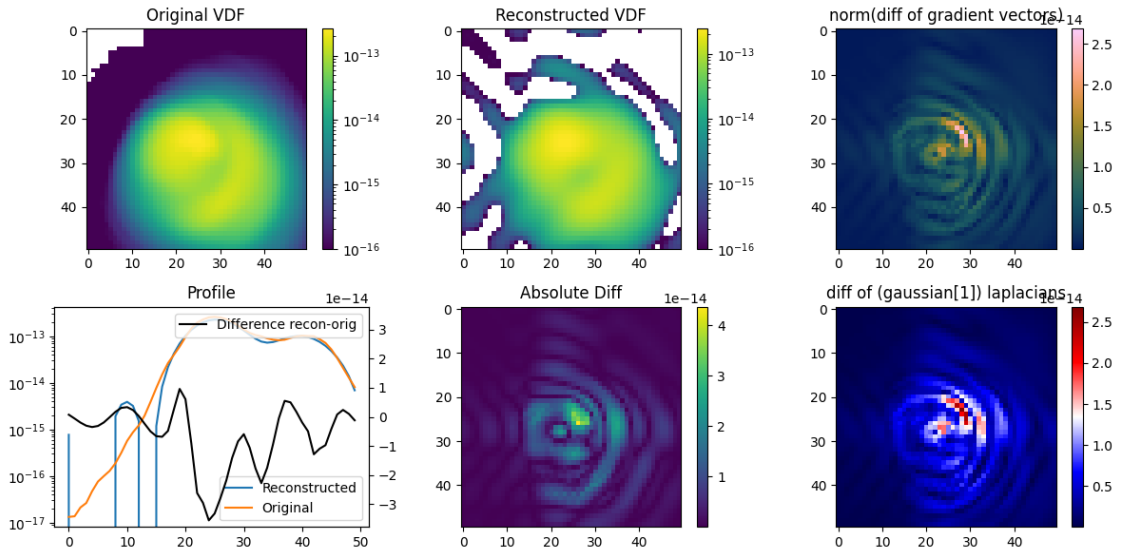
2.2.7 Hermite Decomposition

[13]:
```
"""
Loads the original 3D VDF and fits it to a Maxwellian distribution.
  Input: vdf - numpy array representing the original 3D VDF
  Output: vdf_herm_3d Reconstructed VDF using Hermite Decomposition
"""
### load original 3d vdf and fit Maxwellian
vdf_3d=vdf.copy()
print('loading done')
vdf_size=nx*ny*nz*8

#### Fit Maxwellian
v_min,v_max,n_bins=0,nx,nx ### define limits and size of velocity axes

amp,ux,uy,uz,vthx,vthy,vthz=1e-14,nx,nx,nx,10,10,10 ### initial guess for scipy␣
 ↪curve fit
guess=amp,ux,uy,uz,vthx,vthy,vthz ### initial guess for scipy curve fit

max_fit_3d,params=project_tools.max_fit(vdf_3d,v_min,v_max,n_bins,guess) ###␣
 ↪fitting
print('Maxwell fit done')
```

```
#### forward transform ####
mm=15 ### PUT THE NUMBER OF HARMONICS
norm_amp,u,vth=params[0],params[1:4],params[4:7] ### get the maxwellin fit␣
 ↪parameters of thermal and bulk velocity

vdf_3d_norm=vdf_3d/norm_amp ### normalize data
vdf_3d_flat= vdf_3d_norm.flatten() ### flatten data

v_xyz=project_tools.get_flat_mesh(v_min,v_max,n_bins) ### flattening the mesh␣
 ↪nodes coordinates
herm_array=np.array(project_tools.herm_mpl_arr(m_pol=mm,v_ax=v_xyz,u=params[1:
 ↪4],vth=params[4:7])) ### create array of hermite polynomials

hermite_matrix=project_tools.
 ↪coefficient_matrix(vdf_3d_flat,mm,herm_array,v_xyz) ### calculate the␣
 ↪coefficients of the Hermite transform
print('Forward transform done')
total_size =5*8+8*np.prod(np.shape(hermite_matrix))

#### inverse transform ####
inv_herm_flat=project_tools.inv_herm_trans(hermite_matrix, herm_array, v_xyz)␣
 ↪### inverse Hermite transform
vdf_herm_3d = (np.reshape(inv_herm_flat,(n_bins,n_bins,n_bins)))*norm_amp ###␣
 ↪reshaping back to 3d array and renormalization
print('Inverse transform done')
print(f"Compresion achieved using Hermite = {round(vdf_size/total_size,2)}")
project_tools.plot_vdfs(vdf,vdf_herm_3d)
project_tools.print_comparison_stats(vdf,vdf_herm_3d)
```

```
loading done
Maxwell fit done
array with base polynomials created
Forward transform done
mode number 0
mode number 1
mode number 2
mode number 3
mode number 4
mode number 5
mode number 6
mode number 7
mode number 8
mode number 9
mode number 10
mode number 11
```

```
mode number 12
mode number 13
mode number 14
Inverse transform done
Compresion achieved using Hermite = 36.98
```



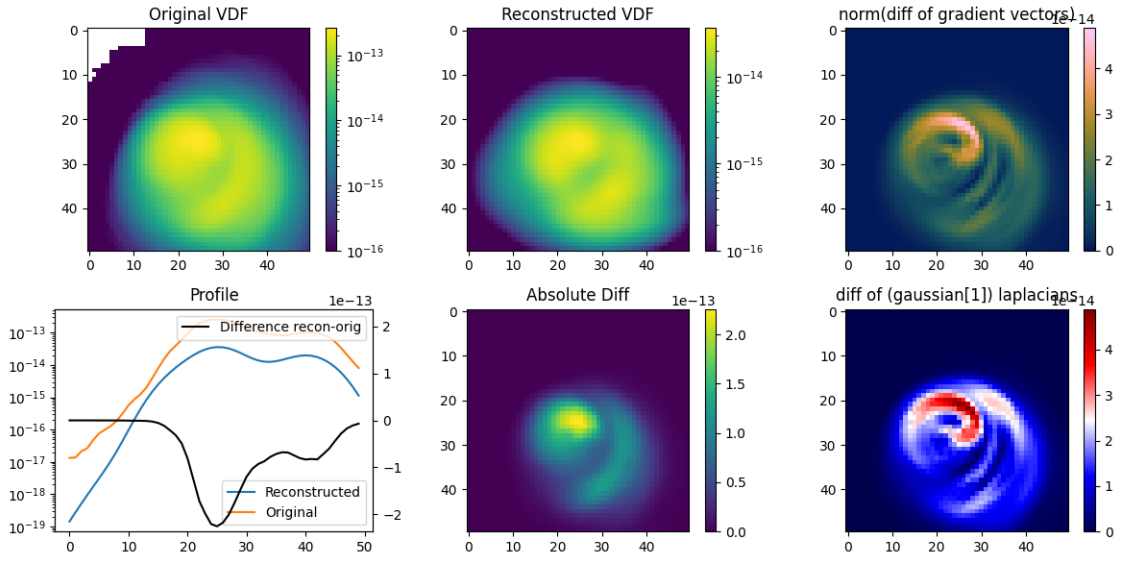

```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (10.802, 2.579,
'5.591e+01', '3.604e+01') %.
L1,L2 rNorms= (0.175, 0.143).
```

2.2.8 Gaussian Mixture Model (GMM)

[14]:
```
"""
 Loads the original 3D VDF and performs Gaussian Mixture Model (GMM)␣
 ↪decomposition.
 Input: vdf - NumPy array representing the original 3D VDF
 Output: vdf_rec Reconstructed VDF using GMM
"""
#### load original 3d vdf
vdf_3d=vdf.copy()

### define number of populations and normalization parameter
n_pop=15
norm_range=300

### RUN GMM
means,weights,covs,norm_unit=project_tools.run_gmm(vdf_3d,n_pop,norm_range)
```

```
### reconstruction resolution and limits of v_space axes
n_bins=nx
v_min,v_max=0,nx

### reconstruction of the vdf
vdf_rec=project_tools.
  ↪reconstruct_vdf(n_pop,means,covs,weights,n_bins,v_min,v_max)
vdf_rec=vdf_rec*norm_unit*norm_range
total_size =5*8+8*np.prod(np.shape(np.array(covs)))+8*np.prod(np.shape(np.
  ↪array(weights)))+8*np.prod(np.shape(np.array(means)))
print(f"Compresion achieved using GMM = {round(vdf_size/total_size,2)}")

project_tools.plot_vdfs(vdf,vdf_rec)
project_tools.print_comparison_stats(vdf,vdf_rec)
```

```
reconstruction: n pop done 0
reconstruction: n pop done 1
reconstruction: n pop done 2
reconstruction: n pop done 3
reconstruction: n pop done 4
reconstruction: n pop done 5
reconstruction: n pop done 6
reconstruction: n pop done 7
reconstruction: n pop done 8
reconstruction: n pop done 9
reconstruction: n pop done 10
reconstruction: n pop done 11
reconstruction: n pop done 12
reconstruction: n pop done 13
reconstruction: n pop done 14
Compresion achieved using GMM = 625.0
```



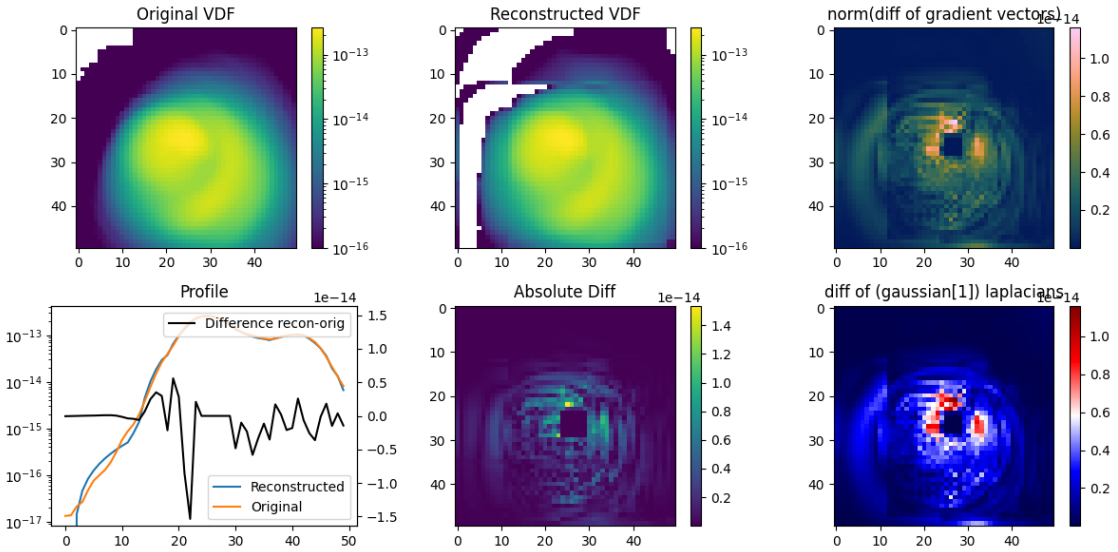
```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (125.695, 5.661,
'1.182e+02', '1.12e+02') %.
L1,L2 rNorms= (0.773, 4.233).
```

2.2.9 Octree based polynomial approximation

- Modify `maxiter` parameter or `tol` parameter to get different levels of accuracy.
- This is pretty slow since its just a single thread cpu implementation with very conservative refinement policy.

[16]:
```
from juliacall import Main as jl
jl.Pkg.activate("src/jl_env")
jl.Pkg.instantiate()
jl.include("src/octree.jl")
vdf_3d = vdf.copy()
A, b, img, reco, cell, tree = jl.VDFOctreeApprox.compress(vdf_3d, maxiter=500,␣
 ↪alpha=0.0, beta=1.0, nu=2, tol=3e-1, verbose=False)
project_tools.plot_vdfs(vdf, reco)
project_tools.print_comparison_stats(vdf,reco)
vdf_size = nx * ny * nz * 8
print(f"Number of leaves: {len(tree)} * basis size: {len(b)} =␣
 ↪{len(tree)*len(b)}")
print(f"Compression achieved with Octree = {round((len(b)*len(tree)*3)/
 ↪vdf_size,3)}. Assuming 8 octals per leaf for geometry representation.")
```

```
  Activating project at `~/dev/asterix/src/jl_env`
WARNING: replacing module VDFOctreeApprox.
```



```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (1.345, 0.533,
'2.55e+01', '1.347e+01') %.
L1,L2 rNorms= (0.068, 0.06).
Number of leaves: 295 * basis size: 27 = 7965
Compression achieved with Octree = 0.024. Assuming 8 octals per leaf for
geometry representation.
```

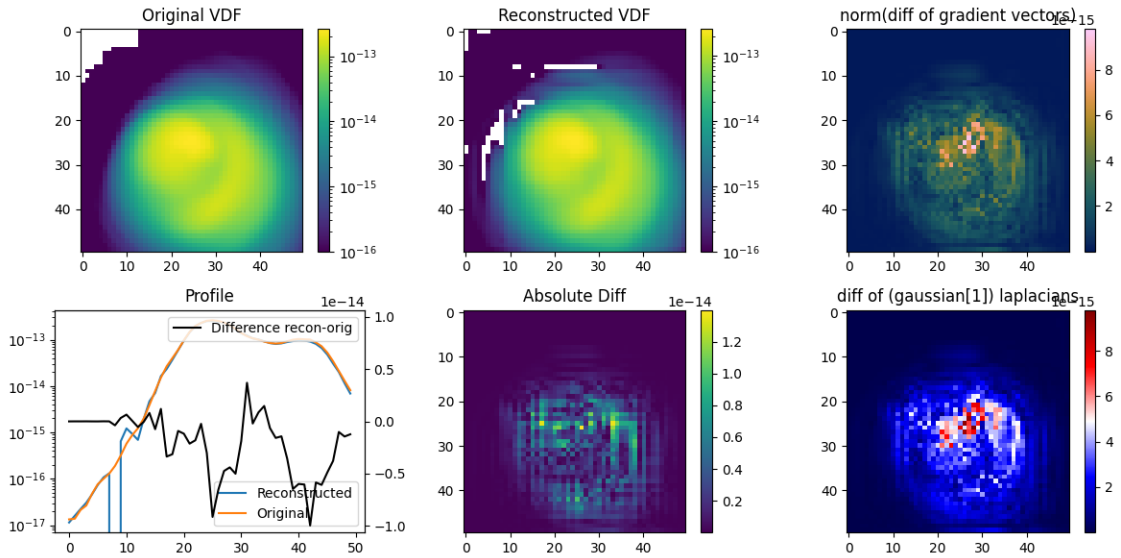
2.2.10 Discrete Cosine Transform (DCT)

DCT is simple to implement and is based on linear operations. This is widely used as the JPEG compression standard for images, but it can be extended to 3D distributions as well. Blockyness is apparent, and there is little room for fine-tuning: block size and number of retained DCT components.

[17]:
```
from scipy.fft import dctn, idctn

vdf_3d = vdf.copy()
orig_shape = vdf_3d.shape
vdf_3d[np.isnan(vdf_3d)] = 0

blocksize = 8
paddings = (np.ceil(np.array(vdf_3d.shape)/8)).astype(int)*8 - vdf_3d.shape
paddings = ((0,paddings[0]),(0,paddings[1]),(0,paddings[2]))
vdf_3d = np.pad(vdf_3d, paddings)

# dct_data = dctn(vdf_3d)
# print(dct_data.shape)
# vdf_rec = idctn(dct_data)
# print(vdf_rec.shape)

block_data = np.zeros_like(vdf_3d)
for i in range(0,vdf_3d.shape[0], blocksize):
    for j in range(0, vdf_3d.shape[1], blocksize):
        for k in range(0, vdf_3d.shape[2], blocksize):
            block_data[i:i+blocksize,j:j+blocksize, k:k+blocksize] =␣
 ↪dctn(vdf_3d[i:i+blocksize,j:j+blocksize, k:k+blocksize])

keep_n = 4
zeroed = np.zeros_like(block_data)
for i in range(keep_n):
    for j in range(keep_n):
        for k in range(keep_n):
```

```
            zeroed[i::blocksize,j::blocksize,k::blocksize] = block_data[i::
 ↪blocksize,j::blocksize,k::blocksize]

# fig, ax = plt.subplots()
# ax.pcolor(np.log(np.abs(zeroed[:,0,:])))


volume_compressed = np.prod(keep_n*np.ceil(np.array(vdf_3d.shape)/8))
volume_orig = np.prod(vdf_3d.shape)
compression = volume_orig/volume_compressed

vdf_rec = np.zeros_like(vdf_3d)
for i in range(0,vdf_3d.shape[0], blocksize):
    for j in range(0, vdf_3d.shape[1], blocksize):
        for k in range(0, vdf_3d.shape[2], blocksize):
            vdf_rec[i:i+blocksize,j:j+blocksize, k:k+blocksize] =␣
 ↪idctn(zeroed[i:i+blocksize,j:j+blocksize, k:k+blocksize])

print("compression:", compression)
# vdf_rec = idctn(dct_data)
vdf_rec = vdf_rec[0:orig_shape[0],0:orig_shape[1],0:orig_shape[2]]

project_tools.plot_vdfs(vdf,vdf_rec)
project_tools.print_comparison_stats(vdf,vdf_rec)
```

```
compression: 8.0
```



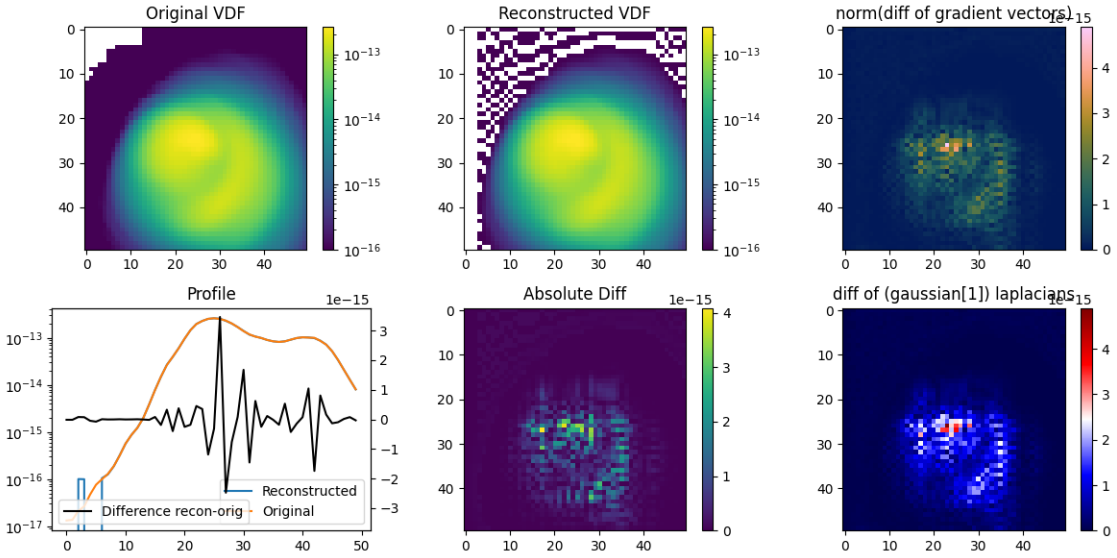

```
(3, 50, 50, 50)
(3, 50, 50, 50)
```

```
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (0.394, 0.114,
'6.583e+00', '7.287e+00') %.
L1,L2 rNorms= (0.046, 0.044).
```

2.2.11 Principal Component Analysis (PCA)

[18]:
```
import numpy as np
from sklearn.decomposition import PCA
vdf_temp=vdf.copy()
vdf_temp[vdf_temp<1e-16]=1e-16
vdf_temp = np.log10(vdf_temp)
arr=vdf_temp.copy()
arr = arr.reshape(arr.shape[0], -1)

# perform pca
cov_matrix = np.cov(arr, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
pca = PCA(n_components=30)
compressed = pca.fit_transform(arr)
#reconstruct the data
recon = pca.inverse_transform(compressed)
nx,ny,nz=np.shape(vdf_temp)
recon=np.reshape(recon,(nx,ny,nz))
recon = 10 ** recon
recon[recon <= 1e-16] = 0
print(f"Compression achieved using PCA = {round(vdf_temp.size / compressed.
  ↪size, 2)}")
project_tools.plot_vdfs(vdf, recon)
project_tools.print_comparison_stats(vdf,recon)
```

```
Compression achieved using PCA = 83.33
```



```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (0.031, 0.001,
'4.832e-01', '1.836e-01') %.
L1,L2 rNorms= (0.006, 0.007).
```

2.2.12 Discrete Wavelet Transform (DWT)

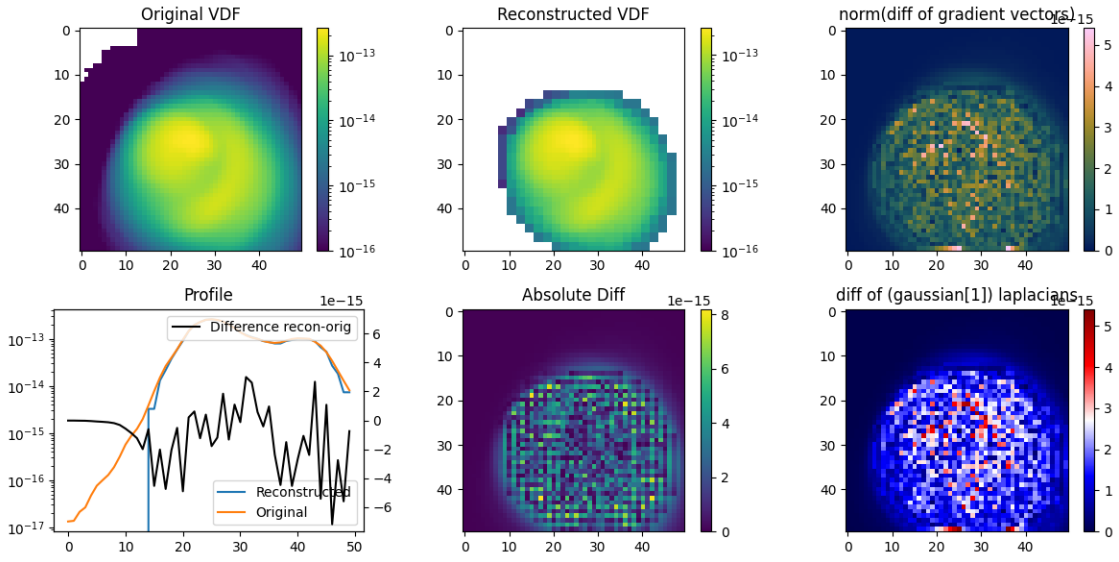
Here we employ the PyWavelets package for prototyping compression via Discrete Wavelet Transforms [@lee_2019].

First, we test quantizing the coefficients as a sanity check.

[19]:
```
import pywt

# print(pywt.wavelist())

vdf_3d = vdf.copy()
orig_shape = vdf_3d.shape
vdf_3d[np.isnan(vdf_3d)] = 0

comp_type = np.int8
quant = np.iinfo(comp_type).max/3
# quant = np.finfo(comp_type).max/3
# quant_min = np.iinfo(comp_type).min/3

# norm = np.max([np.nanmax(vdf_3d)/quant_max, np.nanmin(vdf_3d)/quant_min])
# print(quant_max, quant_min, norm)

norm = np.nanmax(vdf_3d)/quant

vdf_3d /= norm
vdf_3d[vdf_3d<0]=0
print(np.nanmax(vdf_3d),np.nanmin(vdf_3d), norm)


coeffs3 = pywt.dwtn(vdf_3d,'bior1.3')

# print(coeffs3)
coeffs3_compress = coeffs3.copy()

for k, v in coeffs3.items():
    print(np.min(v),np.max(v), len(v))
    coeffs3_compress[k] = v.astype(comp_type)
```

```
vdf_rec = pywt.idwtn(coeffs3_compress,'bior1.3')*norm




volume_compressed = 0
for k,v in coeffs3_compress.items():
    volume_compressed += sys.getsizeof(v)
volume_orig = 0
for k,v in coeffs3.items():
    volume_orig += sys.getsizeof(v)
compression = volume_orig/volume_compressed

print("compression:", compression)

project_tools.plot_vdfs(vdf,vdf_rec)
project_tools.print_comparison_stats(vdf,vdf_rec)
```

```
42.333332 0.0 6.181394286686531e-15
-0.13590407 123.29804 27
-14.305042 10.792248 27
-6.5794706 10.859857 27
-1.700069 1.7653055 27
-13.229418 8.781471 27
-1.8120434 1.5420057 27
-1.755705 1.834994 27
-0.80956554 0.69174093 27
compression: 3.9782115297321834
```



```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (6.009, 0.834,
'1.236e+01', '1.46e+01') %.
L1,L2 rNorms= (0.082, 0.058).
```

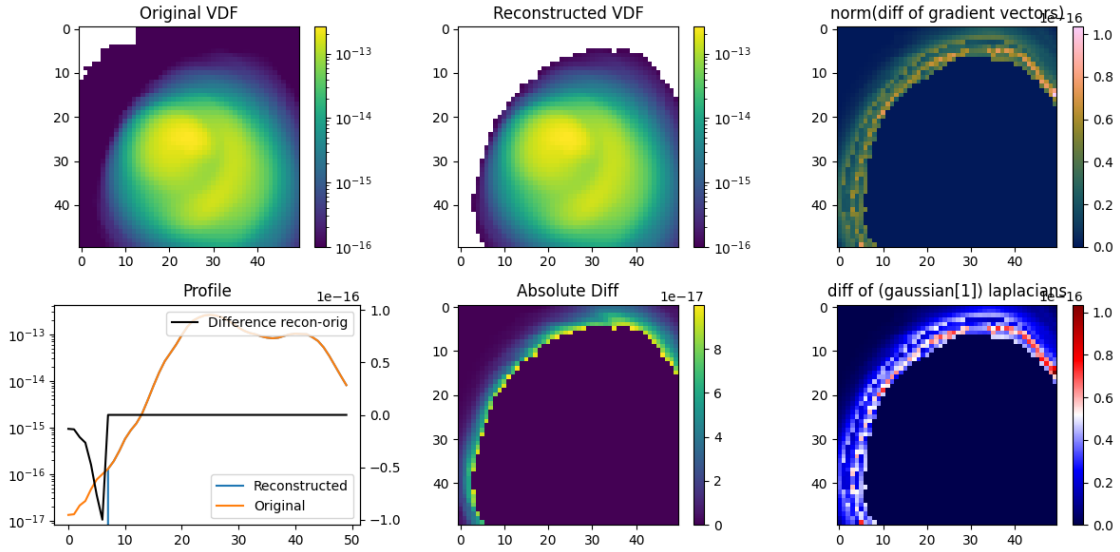
Next, we try a naive thresholding operation on the coefficients for reconstruction. The results are encouraging, and a more elaborate thresholding scheme could help with capturing details at the fringes.

[20]:
```
import pywt

print(pywt.wavelist(kind='discrete'))

vdf_3d = vdf.copy()
orig_shape = vdf_3d.shape
vdf_3d[np.isnan(vdf_3d)] = 0

comp_type = np.float32
# quant = np.iinfo(comp_type).max/3


norm = 1# np.nanmax(vdf_3d)/quant

vdf_3d /= norm
vdf_3d[vdf_3d<0]=0
print(np.nanmax(vdf_3d),np.nanmin(vdf_3d), norm)

wavelet = 'db4'#'bior1.3'

dwtn_mlevel = pywt.dwtn_max_level(vdf_3d.shape,wavelet)
level_delta = 2
print("Decomposing to ", dwtn_mlevel-level_delta, "levels out of ", dwtn_mlevel)
coeffs3 = pywt.wavedecn(vdf_3d,wavelet=wavelet, level = dwtn_mlevel-2)

# print(coeffs3)
coeffs3_comp = coeffs3.copy()
print(type(coeffs3_comp))

zeros = 0
nonzeros = 0
threshold = 1e-16
for i,a in enumerate(coeffs3_comp):
    print(type(a))
    zero_app = False
    # print(a.shape)
    if(type(a) == type(np.ndarray(1))):
        coeffs3_comp[i] = a
```

```
        mask = np.abs(a) < threshold
        zeros += np.sum(mask)
        nonzeros += np.sum(~mask)
        # nonzeros += np.prod(a.shape)
        coeffs3_comp[i][mask] = 0
    else:
        for k,v in a.items():
            mask = np.abs(v) < threshold
            coeffs3_comp[i][k] = v
            coeffs3_comp[i][k][mask] = 0
            zeros += np.sum(mask)
            nonzeros += np.sum(~mask)

print("number of zeros:", zeros, "nonzeros:", nonzeros)
vdf_rec = pywt.waverecn(coeffs3_comp,wavelet=wavelet)*norm

compression = (np.prod(vdf_3d.shape)/nonzeros)


# volume_compressed = 0
# for k,v in coeffs3_compress.items():
#     volume_compressed += sys.getsizeof(v)
# volume_orig = 0
# for k,v in coeffs3.items():
#     volume_orig += sys.getsizeof(v)
# compression = volume_orig/volume_compressed

print("compression:", compression)

project_tools.plot_vdfs(vdf,vdf_rec)
project_tools.print_comparison_stats(vdf,vdf_rec)
```

```
['bior1.1', 'bior1.3', 'bior1.5', 'bior2.2', 'bior2.4', 'bior2.6', 'bior2.8',
'bior3.1', 'bior3.3', 'bior3.5', 'bior3.7', 'bior3.9', 'bior4.4', 'bior5.5',
'bior6.8', 'coif1', 'coif2', 'coif3', 'coif4', 'coif5', 'coif6', 'coif7',
'coif8', 'coif9', 'coif10', 'coif11', 'coif12', 'coif13', 'coif14', 'coif15',
'coif16', 'coif17', 'db1', 'db2', 'db3', 'db4', 'db5', 'db6', 'db7', 'db8',
'db9', 'db10', 'db11', 'db12', 'db13', 'db14', 'db15', 'db16', 'db17', 'db18',
'db19', 'db20', 'db21', 'db22', 'db23', 'db24', 'db25', 'db26', 'db27', 'db28',
'db29', 'db30', 'db31', 'db32', 'db33', 'db34', 'db35', 'db36', 'db37', 'db38',
'dmey', 'haar', 'rbio1.1', 'rbio1.3', 'rbio1.5', 'rbio2.2', 'rbio2.4',
'rbio2.6', 'rbio2.8', 'rbio3.1', 'rbio3.3', 'rbio3.5', 'rbio3.7', 'rbio3.9',
'rbio4.4', 'rbio5.5', 'rbio6.8', 'sym2', 'sym3', 'sym4', 'sym5', 'sym6', 'sym7',
'sym8', 'sym9', 'sym10', 'sym11', 'sym12', 'sym13', 'sym14', 'sym15', 'sym16',
'sym17', 'sym18', 'sym19', 'sym20']
2.6167902e-13 0.0 1
Decomposing to  0 levels out of  2
<class 'list'>
```

```
<class 'numpy.ndarray'>
number of zeros: 59755 nonzeros: 65245
compression: 1.9158556211203923
```




```
(3, 50, 50, 50)
(3, 50, 50, 50)
Velocity Moment relative differences (n,V,P_diag, P_frobenius)= (0.038, 0.005,
'5.151e-02', '9.978e-02') %.
L1,L2 rNorms= (0.0, 0.001).
```

Lastly, stationary wavelet transforms might have desirable qualities. However, computing these is very slow with the current package, and this output is disabled for now.

[21]:
```
if False:
    import pywt

    print(pywt.wavelist(kind='discrete'))

    vdf_3d = vdf.copy()
    orig_shape = vdf_3d.shape
    vdf_3d[np.isnan(vdf_3d)] = 0

    mlevel = 6
    paddings = (np.ceil(np.array(vdf_3d.shape)/2**mlevel)).
 ↪astype(int)*2**mlevel - vdf_3d.shape
    paddings = ((0,paddings[0]),(0,paddings[1]),(0,paddings[2]))
    vdf_3d = np.pad(vdf_3d, paddings)

    comp_type = np.float32
```

```
    # quant = np.iinfo(comp_type).max/3


    norm = 1# np.nanmax(vdf_3d)/quant

    vdf_3d /= norm
    vdf_3d[vdf_3d<0]=0
    # print(np.nanmax(vdf_3d),np.nanmin(vdf_3d), norm)

    wavelet = 'db4'#'bior1.3'
    coeffs3 = pywt.swtn(vdf_3d, wavelet, mlevel)

    # print(coeffs3)
    coeffs3_comp = coeffs3.copy()

    zeros = 0
    nonzeros = 0
    threshold = 1e-15
    for i,a in enumerate(coeffs3_comp):
        # print(type(a))
        # print(a.shape)
        if(type(a) == type(np.ndarray(1))):
            coeffs3_comp[i] = a
            mask = np.abs(a) < threshold
            zeros += np.sum(mask)
            nonzeros += np.sum(~mask)
            coeffs3_comp[i][mask] = 0
        else:
            for k,v in a.items():
                mask = np.abs(v) < threshold
                coeffs3_comp[i][k] = v
                coeffs3_comp[i][k][mask] = 0
                mask = np.abs(v) < threshold
                zeros += np.sum(mask)
                nonzeros += np.sum(~mask)

    print("number of zeros:", zeros, "nonzeros:", nonzeros)
    vdf_rec = pywt.iswtn(coeffs3_comp, wavelet)*norm

    compression = ((zeros+nonzeros)/nonzeros)


    # volume_compressed = 0
    # for k,v in coeffs3_compress.items():
    #     volume_compressed += sys.getsizeof(v)
    # volume_orig = 0
    # for k,v in coeffs3.items():
```

```
    #     volume_orig += sys.getsizeof(v)
    # compression = volume_orig/volume_compressed

    print("compression:", compression)
    vdf_rec = vdf_rec[0:orig_shape[0],0:orig_shape[1],0:orig_shape[2]]

    project_tools.plot_vdfs(vdf,vdf_rec)
    project_tools.print_comparison_stats(vdf,vdf_rec)
```

2.2.13 Vector Quantised-Variational AutoEncoder (VQ-VAE)

Prototype to merged to repository! This approach involves a different approach of first learning an encoder-decoder scheme from a large set of existing data for compression and reconstruction and the packaging this scheme as a compression tool that can be called per-VDF. Our use-case has already large amounts of training data, setting up large-scale training and assessment is to be done.