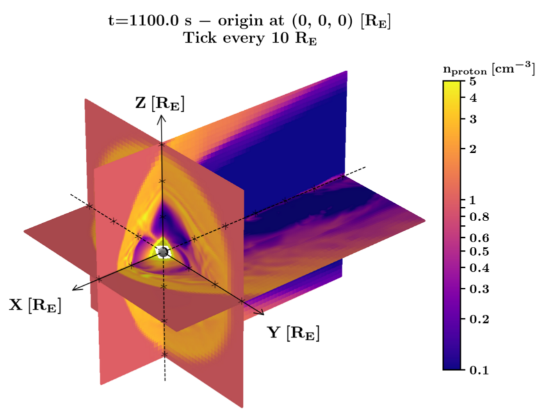
Inno4Scale

April 11, 2024

0.1 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.



0.2 VDF Compression

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

[ ]:
```
import sys,os
# 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

import pytools
```

[ ]:
```
file="/home/kstppd/Desktop/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,(ax1, ax2) = plt.subplots(1, 2)
cax = fig.add_axes([0.95,0.25,0.05,0.5])
im1 = ax1.imshow(vdf[:,:,nz//2],norm=colors.LogNorm(vmin=1e-15,vmax=3e-13))
im2 = ax2.imshow(vdf[:,ny//2,:],norm=colors.LogNorm(vmin=1e-15,vmax=3e-13))



fig.colorbar(im1, cax=cax)
fig.suptitle("Original VDF")
plt.show()
```

0.2.2 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)`.

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

0.2.3 Now in Vlasiator we have countlesss VDFs since there is one per spatial cell. It would be great if we could compress them efficiently. We can try to do so by using zlib which is a form of lossless compression.

[ ]:
```
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)
```

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

[ ]:
```
"""
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)
```

0.2.5 We will compress the VDF using an MLP. [@park2019]

[ ]:
```
"""
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=10
n_layers=4
n_neurons=25
recon=mlp_compress.compress_mlp("sample_vdf.
 ↪bin",order,epochs,n_layers,n_neurons)
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)
```

0.2.6 We will compress the VDF using an MLP with Fourier Features. [@2020fourier]

[ ]:
```
"""
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=12
n_layers=4
n_neurons=25
recon=mlp_compress.compress_mlp("sample_vdf.
 ↪bin",order,epochs,n_layers,n_neurons)
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)
```

0.2.7 Now we use a Spherical Harmonic Decomposition to perform the compression.

[ ]:
```
"""
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
recon=mlp_compress.compress_sph("sample_vdf.bin",degree)
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)
```

0.2.8 Now we use a CNN to perform the compression.

[ ]:
```
"""
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)
```

0.2.9 Here we still use a CNN but this time we use minibatch training and batch normalization layers.

[ ]:
```
"""
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):
```

```python
    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)
```

0.2.10 Now we use Hermite Decomposition to perform the compresion

[ ]:
```
"""
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)
```

0.2.11 Now we use a Gausian Mixture Morel to perform the compresion

```
[ ]: """
 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)
```