Louis Grunfelder, Student at ISAE-SUPAERO

PAS Velocity Distribution Function (VDF) Plotting Tutorial

Introduction:

Greetings !

Solar Orbiter is a space mission designed to explore the inner regions of the heliosphere with various instruments capable of characterizing the solar environment and studying the dynamics of coronal structures. The purpose of this space probe is to discover the fundamental processes that link the internal and surface phenomena of the Sun, the general mass and energy flows in the corona, as well as the creation and acceleration of the solar wind.

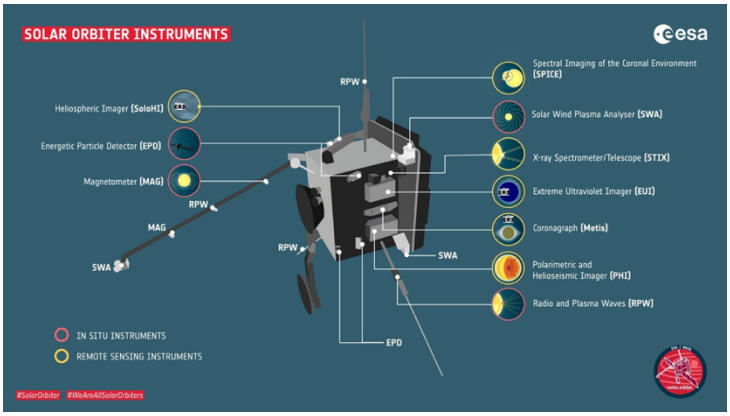


Figure 1: Instruments of Solar Orbiter consisting of 4 in situ instruments (SWA, RPW, MAG, and EPD) and 6 remote sensing instruments (SoloHI, SPICE, STIX, EUI, Metis, and PHI).

This probe is made of various instruments. There are overall 10 instrument suites as illustrated in Figure 1. We focus on the Solar Wind Analyzer (SWA) suite. The SWA suite consist in the following instruments. The first one is PAS (the one we are focusing on in this guide) which is a “Proton Alpha Sensor” that measures the main components in the solar wind , which are protons (H+) and alpha particles (He2+). The second one is HIS, the “Heavy Ion Sensor” that measures that measures the mass and charge state of the heavy ions in the solar wind. The third one is the EAS, “Electron Sensor” that measures the electronic component in the solar wind.

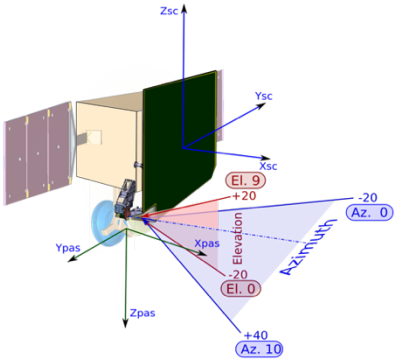


Figure 2: Focus on PAS instrument and it’s

We focus on PAS on this work. The PAS instrument was designed to measure the ion distribution functions of the solar wind. In particular, the instrument is capable to measure the velocity distribution functions (VDF) of protons and alphas. The global goal is to compute the macroscopic parameters of the solar wind such as temperature, density and velocity. Nevertheless, the measurements from PAS also allow us to investigate kinetic properties of the solar wind using the detailed

distribution functions. For instance, one may determine to which mathematical description or formulation the solar wind distribution function is the closer to, to understand the underlying physics related to the origin and the making of the solar wind.

.

More specifically, the instruments has 2 modes of functioning :

-Normal Mode: the instrument measures the 3D ion distribution function every 4 seconds, in the form of matrices of 96 energies, 11 azimuthal angles and 9 elevation angles (as illustrated on Figure 2)

-Burst Mode: measurement rate is increased to 20 Hz (0.1s), with reduced distribution functions, with 24 energies, 5 elevation angles and 7 azimuthal angles.

The code the user will se below was adapted from the original code provided by Jaye Verniero and colleagues (https://github.com/jlverniero/PSP_Data_Analysis_Tutorials/blob/main/PSP_SPAN-I_VDF_Plot_Tutorial.ipynb).
This code is available through Github with permission. For more recent version, please contact (rkieokaew@irap.omp.eu).

Table of contents:

All the code the user will see below was run on python 3.9 with specific versions of some modules that can be found in the requirements.txt part in the Appendix of the document

Plotting Tutorial for VDF in the XYZ frame

The user will now be guided through the code to make the following plots of the velocity distribution function:

- Plot of Vx / Vz in the elevation plane (with the magnetic field vector)
- Plot of Vx / Vy in the azimuth plane (with the magnetic field vector)

The user guide will be divided in different steps to make it more understandable :

Step 1

First, we import the cdf reader from cdflib (the library able to read the data encoded in the cdf format which is used for PAS when the date is received). We also import numpy

for math operations and datetime for time unit conversions. The modules below such as warnings are not essential but useful in order to make the code and the plots clearer.

```
import cdflib
import numpy as np
import os.path
import math as m
import matplotlib.pyplot as plt
from spacepy import pycdf

from datetime import datetime
import bisect

from matplotlib import ticker, cm
import warnings
warnings.filterwarnings("ignore")

from warnings import simplefilter
simplefilter(action='ignore', category=DeprecationWarning)
```

Step 2

The user will now have to download the data and look into the cdf files to retrieve the variables necessary to the calculations. The files used are those of Solar Orbiter downloaded on https://soar.esac.esa.int/soar/ at the right date and on the L2 level. For example if the user want to download the data for the date 26/03/2022 the file will have the following name solo_L2_swa-pas-vdf_20220326_V02.cdf (this file contains only the data on the vdf). The user will also have to download the file containing the magnetic field data useful later in the code. For the same date as before the file will have the following name solo_L2_mag-srf-normal_20220326_V01.cdf

```
#open CDF file

dat = cdflib.CDF('./Data/solo_L2_swa-pas-vdf_20220326_V02.cdf')
infile = cdflib.CDF('./Data/solo_L2_mag-srf-normal_20220326_V01.cdf')

#print variable names in CDF files
#print(dat._get_varnames())
```

Step 3

In the following step we are defining some variables from the cdf files. Afterwards, the data is reshaped. Indeed, the code only creates tables with values that are repeting themselves but it is only to make the calculations easier. Moreover, at the end of the second cell 2 sets of values are removed from the vdf. These values are the ones at the periphery of the field of view in the azimuth plane. These ones are removed in order to have symmetric vdf containing as many values in the elevation plane than in the azimuth plane to ease the calculations, this will not change the results. In addition, the fact that changing from 11 to 9 azimuthal bins allows us to have minimal changes in the original code.

```
epoch_ns        = dat['Epoch']
elevation       = dat['Elevation']
azimuth         = dat['Azimuth']
energy          = dat['Energy']
vdf             = dat['vdf']
mag_field       = infile['B_SRF']
epoch_mag       = infile['EPOCH']

## Making the data in the good shape ##

temp = []
for i in range (9):
    for j in range(96):
        for k in range(9):
            temp.append(energy[j])
energyLine = np.array(temp)
#print(energyLine.shape)


temp1 = []
for i in range (864):
    for j in range(9):
        temp1.append(elevation[j])
elevationLine = np.array(temp1)
#print(elevationLine.shape)

temp2 = []
for i in range (9):
    for j in range (864):
        temp2.append(azimuth[i])
azimuthLine = np.array(temp2)
#print(azimuthLine.shape)

vdf_temp = np.delete(vdf,10,0)
vdfLine = np.delete(vdf_temp,9,0)
#print(vdf.shape)
```

Step 4

In this step the user will have to choose a timeslice by specifying the year, month, day, hour, minute, and second. Make sure that the date specified in the cell corresponds to the date in the file of the second step to avoid a timeindex error. The functions “bisect” are used to retrieve the closest point of data that correspond to the date the user asked for. Moreover, as the data for the VDF and the magnetic field were not sampled at the same rate, in order to have the same point in time for the two data (VDF and magnetic), two time indexes have to be calculated for the same datetime or as close as possible. For now it is set up to take a single date, but in the last step of this tutorial, a time range will be implemented in order to make plots over a larger range (hours,days,weeks..).

```
#convert time
import datetime
epoch = cdflib.cdfepoch.to_datetime(epoch_ns)
epoch_mag_conv = cdflib.cdfepoch.to_datetime(epoch_mag)
#print(epoch)

year=2022
month=3
day=26
hour = 17
minute = 24
second = 0

timeSlice  = datetime.datetime(year, month, day, hour, minute, second)
print('Desired timeslice:',timeSlice,'\n')

#find index for desired timeslice
tSliceIndex      = bisect.bisect_left(epoch,timeSlice)
tSliceIndex_mag  = bisect.bisect_left(epoch_mag_conv,timeSlice)
print('Time Index For Vdf:',tSliceIndex)
print('Time of closest data point for Vdf:',epoch[tSliceIndex],'\n')
print('Time Index For Mag:',tSliceIndex_mag)
print('Time of closest data point for Mag:',epoch_mag_conv[tSliceIndex_mag])
```

Step 5

This step is about slicing the data to the right time slice (the one the user has chosen in the previous step) and retrieving the magnetic field to compute the magnetic field vector (Bx,By,Bz) and to make the calculation of Vpar/Vperp (the velocities perpendicular and parallel to the magnetic field) that will be used later in the code.

```
elevationSlice  = elevationLine
azimuthSlice    = azimuthLine
energySlice     = energyLine
vdfSlice        = vdfLine[tSliceIndex,:]

## Magnetic coordinates used to calculate the magnetic field vector

Bx = mag_field[tSliceIndex_mag,0]
By = mag_field[tSliceIndex_mag,1]
Bz = mag_field[tSliceIndex_mag,2]

## Magnetic coordinates used to calculate Vpar/Vperp
Bx1 = mag_field[tSliceIndex_mag,0]
By1 = mag_field[tSliceIndex_mag,1]
Bz1 = mag_field[tSliceIndex_mag,2]
```

Moreover, to make the calculation and the code easier, we reshape all the data to have a similar shape to the VDF (more readable format reflecting 9-azimuth-direction bins, 32 energy bins, and 9-elevation-direction bins).

```
elevationReshaped  = elevationSlice.reshape((9,96,9))
azimuthReshaped    = azimuthSlice.reshape((9,96,9))
energyReshaped     = energySlice.reshape((9,96,9))

mass_p = 0.010438870      #proton mass in units eV/c^2 where c = 299792 km/s
charge_p = 1              #proton charge in units eV
```

Step 6

As we have now prepared all the parameters, the user is ready to make plots of VDF. Since the VDF is 3D, we have possibilities of plotting them in 2D, either in the Vx-Vz (elevation plane) or the Vx-Vy (azimuth plane). First the code below allows the user to Vx-Vz. So to do that we cut through the elevation plane that is on dimension 0 for the azimuth, elevation, and energy and on the first axis for the vdf.

The limits of the plots must be chosen manually depending on the solar wind speed and its spread from the bulk flow speed. Furthermore, the origin of the magnetic field vector can be chosen manually either on the center of the bulk flow velocity or the center of the plot etc..
Here are the coordinate transformation from the azimuthal and elevation space to the velocity spacethat are used in the code :

$$V_X^{SRF} = -\cos(El)\cdot\cos(Az),$$
$$V_Y^{SRF} = \cos(El)\cdot\sin(Az),$$
$$V_Z^{SRF} = -\sin(El).$$

```
#elevation is along dimension 0, while azimuth is along 1
#first cut through elevation
elevation_cut = 0

azimuth_plane   = azimuthReshaped[elevation_cut,:,:]
elevation_plane = elevationReshaped[elevation_cut,:,:]
energy_plane    = energyReshaped[elevation_cut,:,:]

#Convert to velocity units in each energy channel
vel_plane       = np.sqrt(2*charge_p*energy_plane/mass_p)

df_elevation=np.transpose(np.nansum(vdfSliceFin,axis=1))*(10**12)

#Rotate from energy-angle space to cartesian (vx,vy,vz) space (still in the SRF frame)
vx_plane_elevation = -vel_plane * np.cos(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
vy_plane_elevation = vel_plane * np.sin(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
vz_plane_elevation = -vel_plane * np.sin(np.radians(elevation_plane))

fig,ax = plt.subplots()
cs = ax.contourf(vx_plane_elevation, vz_plane_elevation, df_elevation,cmap='jet',locator=ticker.LogLocator(subs=(
cbar = fig.colorbar(cs)
cbar.set_label(f'f $(cm^2 \\ s \\ sr \\ eV)^{-1}$')

origin = [-350], [50]
magnetic = ax.quiver(*origin,Bx,Bz,color='k',scale=120)

xmin = -800
xmax = -200
ymin = -200
ymax = 300

ax.set_xlim(xmin,xmax)
ax.set_ylim(ymin,ymax)
ax.set_aspect(1)
ax.set_xlabel('$v_x$  (km/s)')
ax.set_ylabel('$v_z$  (km/s)')
ax.set_title(epoch[tSliceIndex].strftime('%d-%h-%Y  %T.%f'))
#plt.savefig('./26_02_2022/VDF_XZ_' + str(day) + '_' + str(month) + '_' + str(year) + '_' + str(hour) +'h_' + str

plt.hide()
```

Next, the user can do a visualization on the azimuth to display Vx-Vy. Here, we have to cut through the azimuth plane that is on dimension 0 for the azimuth, elevation, and energy and on the first axis for the vdf.

```
#now repeat for azimuth dimension
azimuth_cut = 1

azimuth_plane   = azimuthReshaped[:,:,azimuth_cut]
elevation_plane = elevationReshaped[:,:,azimuth_cut]
energy_plane    = energyReshaped[:,:,azimuth_cut]
vel_plane       = np.sqrt(2*charge_p*energy_plane/mass_p)

df_azimuth=np.nansum(vdfSliceFin,axis=0)*(10**12)

vx_plane_azimuth = -vel_plane * np.cos(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
vy_plane_azimuth = vel_plane * np.sin(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
vz_plane_azimuth = -vel_plane * np.sin(np.radians(elevation_plane))

fig,ax = plt.subplots()
cs = ax.contourf(vx_plane_azimuth, vy_plane_azimuth,df_azimuth,locator=ticker.LogLocator(subs=(1,2,3)),cmap='jet'
cbar = fig.colorbar(cs,ticklocation='top')
cbar.set_label(f'f $(cm^2 \\ s \\ sr \\ eV)^{-1}$')

origin = [-350], [-50]
magnetic = ax.quiver(*origin,Bx,By,color='k',scale=120)

xmin1 = -800
xmax1 = -200
ymin1 = -200
ymax1 = 300

ax.set_xlim(xmin1,xmax1)
ax.set_ylim(ymin1,ymax1)
ax.set_aspect(1)
ax.set_xlabel('$v_x$  (km/s)')
ax.set_ylabel('$v_y$  (km/s)')
ax.set_title(epoch[tSliceIndex].strftime('%d-%h-%Y  %T.%f'))
plt.savefig('./26_02_2022/VDF_XY_' + str(day) + '_' + str(month) + '_' + str(year) + '_' + str(hour) +'h_' + str(

#plt.show()
```

Magnetic Field-Aligned-Coordinates

Now that the user has plotted the VDF in the XYZ coordinates , now we will guide the user to plot the VDF in the magnetic field direction. Indeed to do that the user will rotate the processed data into field-aligned coordinates. Two processes are defined below, one of which will output the elements of a rotation matrix from instrument coordinates to field-aligned coordinates (made up of the three unit vectors defining those coordinates), and another which will rotate an input vector from instrument to field-aligned coordinates. This process will make several assumptions, the largest being that the magnetic field is perfectly defined by each measurement over the course of the PAS collection period.

Here is the first process :

```
def fieldAlignedCoordinates(Bx, By, Bz):
    '''
    INPUTS:
        Bx, By, Bz = rank1 arrays of magnetic field measurements in instrument frame
    '''
    import numpy as np


    Bmag = np.sqrt(Bx**2 + By**2 + Bz**2)

    # Define field-aligned vector
    Nx = Bx/Bmag
    Ny = By/Bmag
    Nz = Bz/Bmag

    # Make up some unit vector
    if np.isscalar(Nx):
        Rx = 0
        Ry = 0.9
        Rz = 0
    else:
        Rx = np.zeros(Nx.len())
        Ry = np.ones(len(Nx))
        Rz = np.zeros(len(Nx))

    # Find some vector perpendicular to field NxR
    TEMP_Px = ( Ny * Rz ) - ( Nz * Ry )  # P = NxR
    TEMP_Py = ( Nz * Rx ) - ( Nx * Rz )  # This is temporary in case we choose a vector R that is not unitary
    TEMP_Pz = ( Nx * Ry ) - ( Ny * Rx )


    Pmag = np.sqrt( TEMP_Px**2 + TEMP_Py**2 + TEMP_Pz**2 ) #Have to normalize, since previous definition does not

    Px = TEMP_Px / Pmag # for R=(0,1,0), NxR = P ~= RTN_N
    Py = TEMP_Py / Pmag
    Pz = TEMP_Pz / Pmag

    Qx = ( Pz * Ny ) - ( Py * Nz )   # N x P
    Qy = ( Px * Nz ) - ( Pz * Nx )
    Qz = ( Py * Nx ) - ( Px * Ny )

    return(Nx, Ny, Nz, Px, Py, Pz, Qx, Qy, Qz)
```

```
# ###
# ### TRANSFORM VECTOR DATA INTO FIELD-ALIGNED COORDINATES
# ###

def rotateVectorIntoFieldAligned(Ax, Ay, Az, Nx, Ny, Nz, Px, Py, Pz, Qx, Qy, Qz):
    # For some Vector A in the SAME COORDINATE SYSTEM AS THE ORIGINAL B-FIELD VECTOR:

    An = (Ax * Nx) + (Ay * Ny) + (Az * Nz)  # A dot N = A_parallel
    Ap = (Ax * Px) + (Ay * Py) + (Az * Pz)  # A dot P = A_perp (~RTN_N (+/- depending on B), perpendicular to s/c
    Aq = (Ax * Qx) + (Ay * Qy) + (Az * Qz)  #

    return(An, Ap, Aq)
```

In the second process the user will have to compute the average_table (in order to center the plots at the (0;0) coordinates) which is given by the function here below :

```
def average_table(tableau):
    total = 0
    elements = 0
    for ligne in tableau:
        for element in ligne:
            total += element
            elements += 1
    moyenne = total / elements
    return moyenne
```

As we have now defined the field aligned coordinates above, we will perform the frame transformation and make a plot as follows :

```
(Nx, Ny, Nz, Px, Py, Pz, Qx, Qy, Qz) = fieldAlignedCoordinates(Bx1, By1, Bz1)

vx_av = average_table(vx_plane_elevation)
vy_av = average_table(vy_plane_elevation)
vz_av = average_table(vz_plane_elevation)

(vn_plane, vp_plane, vq_plane) = rotateVectorIntoFieldAligned(vx_plane_elevation-vx_av, vy_plane_elevation-vy_av,

fig,ax=plt.subplots()
cs=ax.contourf(-vp_plane, -vn_plane, df_elevation,locator=ticker.LogLocator(subs=(1,2,3)),cmap='jet')
cbar = fig.colorbar(cs)
cbar.set_label(f'f $(cm^2 \\ s \\ sr \\ eV)^{-1}$')

ax.set_xlim(-300,300)
ax.set_ylim(-300,300)
ax.set_aspect(1)
ax.set_ylabel('$v_\perp$ km/s')
ax.set_xlabel('$v_\parallel$ km/s')
ax.set_title(epoch[tSliceIndex].strftime('%d-%h-%Y  %T.%f'))
plt.savefig('./26_02_2022/VDF_ParPerp_' + str(day) + '_' + str(month) + '_' + str(year) + '_' + str(hour) +'h_' +


plt.show()
```

All-in-one code within a time interval

Below the user has all the codes in one block. Here, the user can make the plots of all the desired timeslices by specifying the range of hours and range of minutes in the "for" loops.

```
import cdflib
import numpy as np
import os.path
import math as m
import matplotlib.pyplot as plt
from spacepy import pycdf

import wget

from datetime import datetime
import bisect

from matplotlib import ticker, cm
import warnings
warnings.filterwarnings("ignore")

from warnings import simplefilter
simplefilter(action='ignore', category=DeprecationWarning)

###### Open CDF file #####
dat1 = cdflib.CDF('./Data/solo_L2_swa-pas-vdf_20220325_V02.cdf')
infile = cdflib.CDF('./Data/solo_L2_mag-srf-normal_20220325_V01.cdf')

##### Retrieving The Variables #####

epoch_ns        = dat1['Epoch']
elevation       = dat1['Elevation']
azimuth         = dat1['Azimuth']
energy          = dat1['Energy']
vdf             = dat1['vdf']
mag_field       = infile['B_SRF']
epoch_mag       = infile['EPOCH']

mass_p = 0.010438870      #proton mass in units eV/c^2 where c = 299792 km/s
charge_p = 1              #proton charge in units eV
```

```
## Making the data in the good shape ##

temp = []
for i in range (9):
    for j in range(96):
        for k in range(9):
            temp.append(energy[j])
energyLine = np.array(temp)

temp1 = []
for i in range (864):
    for j in range(9):
        temp1.append(elevation[j])
elevationLine = np.array(temp1)

temp2 = []
for i in range (9):
    for j in range (864):
        temp2.append(azimuth[i])
azimuthLine = np.array(temp2)

vdf_temp = np.delete(vdf,10,0)
vdfLine = np.delete(vdf_temp,9,0)

###### Choosing the right time slice that we want #####

import datetime
epoch = cdflib.cdfepoch.to_datetime(epoch_ns)
epoch_mag_conv = cdflib.cdfepoch.to_datetime(epoch_mag)

for hour in range (0,24) : ## Selecting the hour interval of the plots
    for minute in range (0,60,15): ## Selecting the minute interval in each hour

        # Choose the right date

        year=2022
        month=3
        day=25
        second = 0
```

```
        timeSlice  = datetime.datetime(year, month, day, hour, minute, second)
        print('Desired timeslice:',timeSlice,'\n')

        tSliceIndex      = bisect.bisect_left(epoch,timeSlice)
        tSliceIndex_mag  = bisect.bisect_left(epoch_mag_conv,timeSlice)
        print('Time Index For Vdf:',tSliceIndex)
        print('Time of closest data point for Vdf:',epoch[tSliceIndex],'\n')
        print('Time Index For Mag:',tSliceIndex_mag)
        print('Time of closest data point for Mag:',epoch_mag_conv[tSliceIndex_mag])

##### Making The Data in the right format to make calculations #####

        elevationSlice  = elevationLine
        azimuthSlice    = azimuthLine
        energySlice     = energyLine
        vdfSlice        = vdfLine[tSliceIndex,:]

        Bx = mag_field[tSliceIndex_mag,0]
        By = mag_field[tSliceIndex_mag,1]
        Bz = mag_field[tSliceIndex_mag,2]

        Bx1 = mag_field[tSliceIndex_mag,0]
        By1 = mag_field[tSliceIndex_mag,1]
        Bz1 = mag_field[tSliceIndex_mag,2]

        vdf_temp = np.delete(vdfSlice,10,0)
        vdfSliceFin = np.delete(vdf_temp,9,0)

        elevationReshaped  = elevationSlice.reshape((9,96,9))
        azimuthReshaped    = azimuthSlice.reshape((9,96,9))
        energyReshaped     = energySlice.reshape((9,96,9))

##### Making the XYZ plots for the VDF #####

### elevation is along dimension 0, while azimuth is along 1
#first cut through elevation

        elevation_cut = 0

        azimuth_plane   = azimuthReshaped[elevation_cut,:,:]
        elevation_plane = elevationReshaped[elevation_cut,:,:]
```

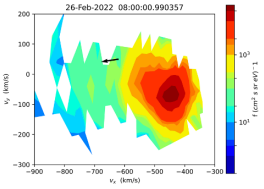
```
        energy_plane    = energyReshaped[elevation_cut,:,:]
        vel_plane       = np.sqrt(2*charge_p*energy_plane/mass_p)

        df_elevation=np.transpose(np.nansum(vdfSliceFin,axis=1))*(10**12)

        vx_plane_elevation = -vel_plane * np.cos(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
        vy_plane_elevation = vel_plane * np.sin(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
        vz_plane_elevation = -vel_plane * np.sin(np.radians(elevation_plane))

        fig,ax = plt.subplots()
        cs = ax.contourf(vx_plane_elevation, vz_plane_elevation, df_elevation,cmap='jet',locator=ticker.LogLocato
        cbar = fig.colorbar(cs)
        cbar.set_label(f'f $(cm^2 \\ s \\ sr \\ eV)^{-1}$')

        xmin = -900
        xmax = -300
        ymin = -250
        ymax = 200

        ax.set_xlim(xmin,xmax)
        ax.set_ylim(ymin,ymax)

        origin = [-420], [20]
        magnetic = ax.quiver(*origin,Bx,Bz,color='k',scale=120)

        ax.set_xlabel('$v_x$  (km/s)')
        ax.set_ylabel('$v_z$  (km/s)')
        ax.set_aspect(1)
        ax.set_title(epoch[tSliceIndex].strftime('%d-%h-%Y  %T.%f'))
        plt.savefig('./26_02_2022/VDF_XZ_' + str(day) + '_' + str(month) + '_' + str(year) + '_' + str(hour) +'h_

### now repeat for azimuth dimension
        azimuth_cut = 1

        azimuth_plane   = azimuthReshaped[:,:,azimuth_cut]
        elevation_plane = elevationReshaped[:,:,azimuth_cut]
        energy_plane    = energyReshaped[:,:,azimuth_cut]
        vel_plane       = np.sqrt(2*charge_p*energy_plane/mass_p)

        df_azimuth=np.nansum(vdfSlice,axis=0)*(10**12)
        vx_plane_azimuth = -vel_plane * np.cos(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
        vy_plane_azimuth = vel_plane * np.sin(np.radians(azimuth_plane)) * np.cos(np.radians(elevation_plane))
        vz_plane_azimuth = -vel_plane * np.sin(np.radians(elevation_plane))

        fig,ax = plt.subplots()
        cs = ax.contourf(vx_plane_azimuth, vy_plane_azimuth,df_azimuth,locator=ticker.LogLocator(subs=(1,2,3)),cm
        cbar = fig.colorbar(cs,ticklocation='top')
        cbar.set_label(f'f $(cm^2 \\ s \\ sr \\ eV)^{-1}$')

        xmin1 = -900
        xmax1 = -300
        ymin1 = -250
        ymax1 = 200

        ax.set_xlim(xmin1,xmax1)
        ax.set_ylim(ymin1,ymax1)

        origin1 = [-420], [20]
        magnetic = ax.quiver(*origin1,Bx,By,color='k',scale=120)

        ax.set_xlabel('$v_x$  (km/s)')
        ax.set_ylabel('$v_y$  (km/s)')
        ax.set_aspect(1)
        ax.set_title(epoch[tSliceIndex].strftime('%d-%h-%Y  %T.%f'))
        plt.savefig('./26_02_2022/VDF_XY_' + str(day) + '_' + str(month) + '_' + str(year) + '_' + str(hour) +'h_

        (Nx, Ny, Nz, Px, Py, Pz, Qx, Qy, Qz) = fieldAlignedCoordinates(Bx1, By1, Bz1)

        vx_av = average_table(vx_plane_elevation)
        vy_av = average_table(vy_plane_elevation)
        vz_av = average_table(vz_plane_elevation)

        (vn_plane, vp_plane, vq_plane) = rotateVectorIntoFieldAligned(vx_plane_elevation-vx_av, vy_plane_elevatic

        fig,ax=plt.subplots()
        cs=ax.contourf(vp_plane, vn_plane, df_elevation,locator=ticker.LogLocator(subs=(1,2,3)),cmap='jet')
        cbar = fig.colorbar(cs)
        cbar.set_label(f'f $(cm^2 \\ s \\ sr \\ eV)^{-1}$')

        ax.set_xlim(-400,400)
        ax.set_ylim(-250,400)
```

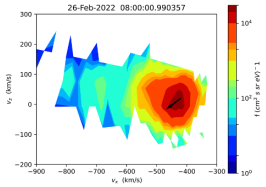
```
        ax.set_aspect(1)
        ax.set_xlabel('$v_\perp$ km/s')
        ax.set_ylabel('$v_\parallel$ km/s')
        ax.set_title(epoch[tSliceIndex].strftime('%d-%h-%Y  %T.%f'))
        plt.savefig('./26_02_2022/VDF_ParPerp_' + str(day) + '_' + str(month) + '_' + str(year) + '_' + str(hour)

        plt.show()
```

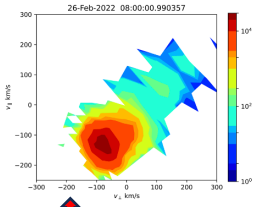
Benchmarking of the code for the two frames

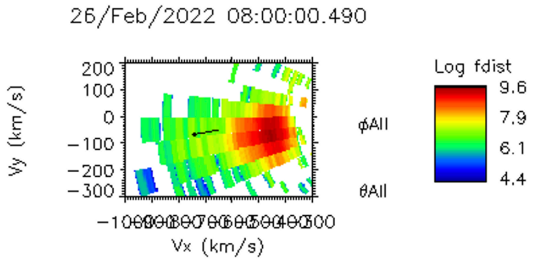
Benchmarking :

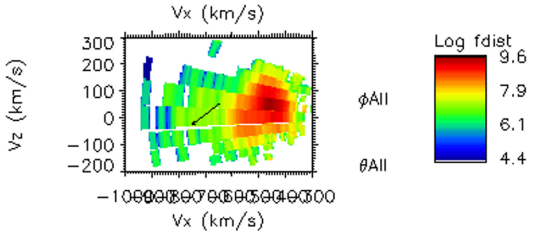
26-02-2022
8.am

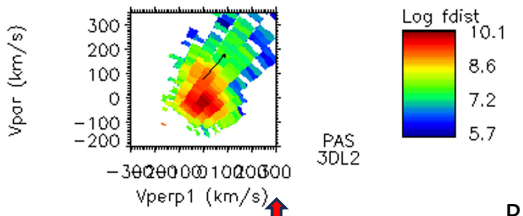






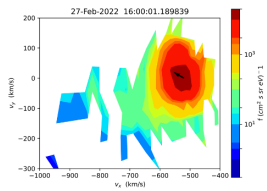


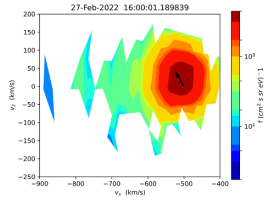


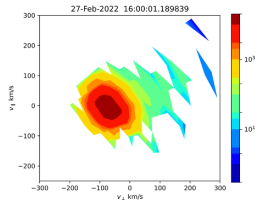


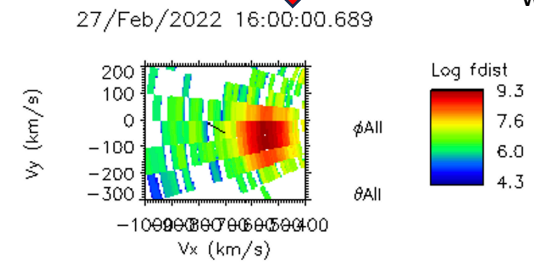
Plot produced with the code presented before

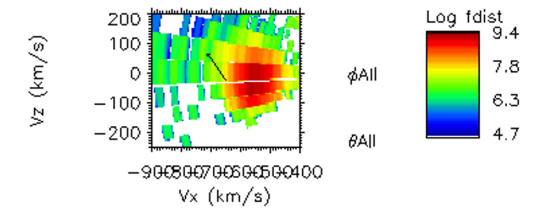
Plot produced with CLWEB

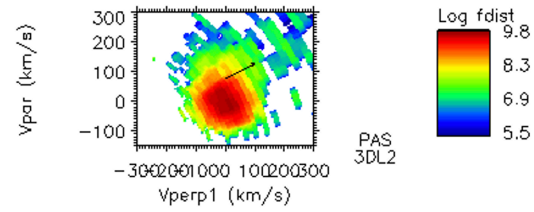










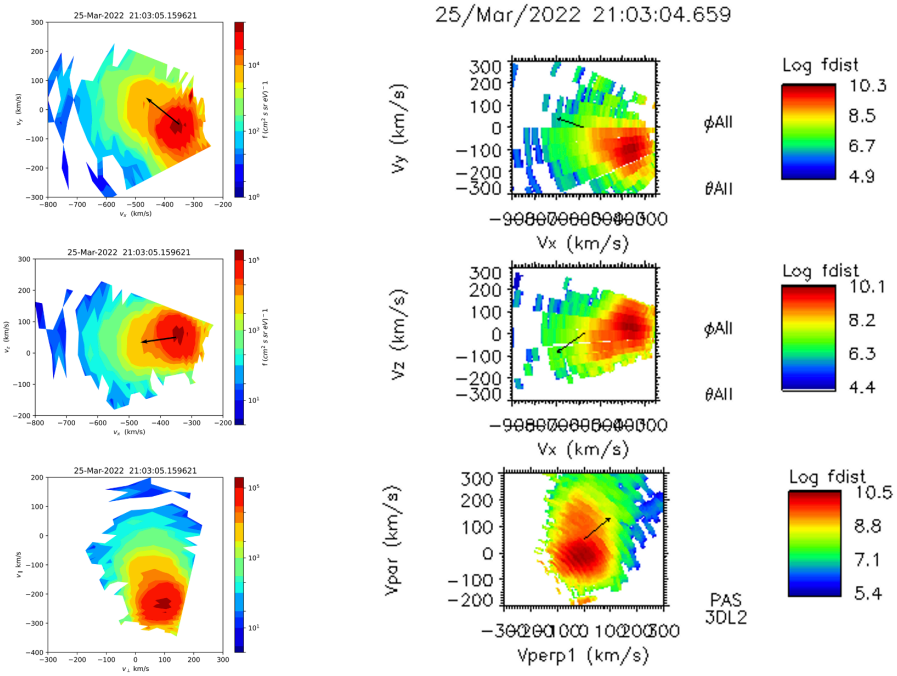


Benchmarking :

27-02-2022
4.pm

Benchmarking :

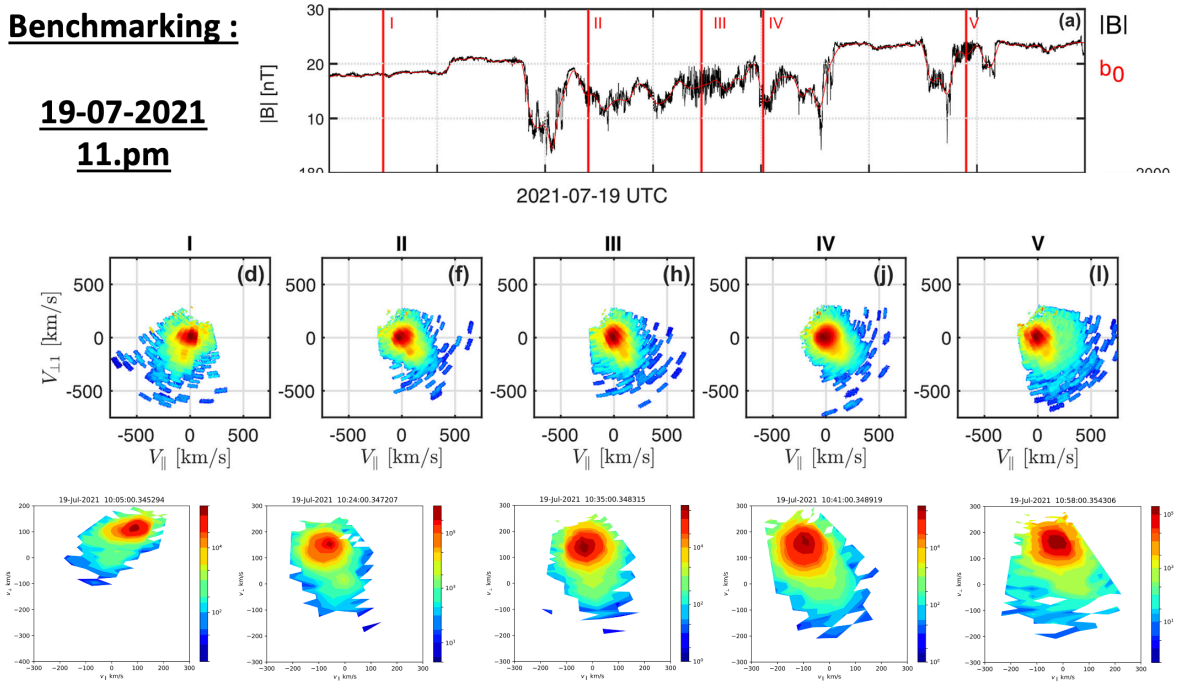
25-03-2022
9.pm



This benchmarking (below) is different from above as only the VDF in the field-aligned coordinates is benchmarked against the paper published by Dimmock et al. (2022). The user can find the paper at the following address https://onlinelibrary.wiley.com/doi/abs/10.1029/2022JA030754.

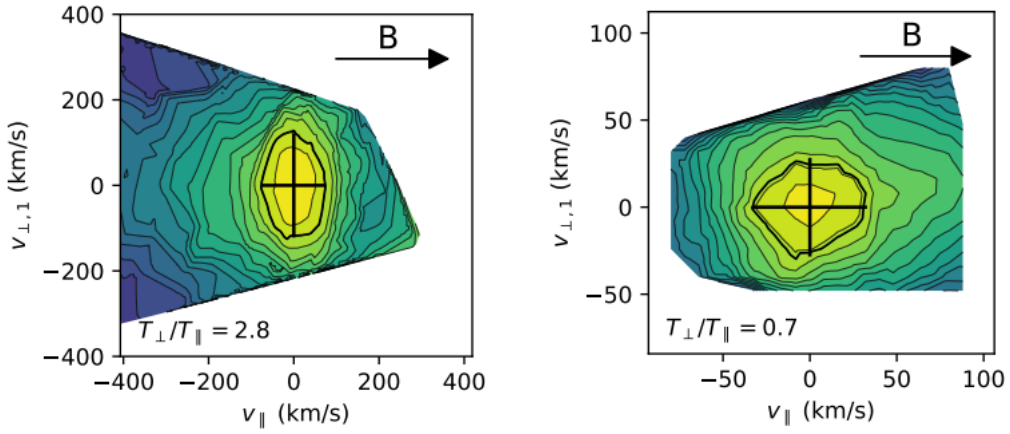
Benchmarking :

19-07-2021
11.pm



Conclusion & Future Work

To conclude, we have developed the tools to plot VDF useful for investigating kinetic physics of the major species of the solar wind ions measured by PAS. Particularly, the user can identify the core population of protons where the counts are high and often the beam population that follows the direction of the magnetic field. These two populations (especially the beam one) have not been much studied in the literature. The development of these tools will facilitate such the future studies. For instance, the user could try different fit functions such as a Maxwellian or a bi-Maxwellian distribution to describe mathematically the observed distribution function. An example of the plots below from Stansby et al (2018) may be considered for a future development of these tools.



These are examples of a fast solar wind distribution function data and corresponding fit for data of solar wind given by the mission Helios 2 in 1976 (https://link.springer.com/article/10.1007/s11207-018-1377-3)

Appendix

Requirements.txt :

```
cdflib==0.4.9
numpy==1.24.2
spacepy==0.4.1
```