KG1LH$_{py}$Documentation

Release 1.0.0.0

Bruno Viola

Jul 23, 2019

CONTENTS

CHAPTER

ONE

PROJECT SUMMARY

This code is meant to be an useful tool for the KG1 responsible officer or anybody else with rights to operate with JET interpherometer related data (i.e. must have rights to write public ppf and access data)

CHAPTER

TWO

TUTORIAL

In this section I will explain how to use the tool and what is possible to achieve.

2.1 Installation process

To run and use the code the user must first install it and all its dependacies.

The code is stored in a git repository

from terminal

>> git clone https://git.ccfe.ac.uk/bviola/kg1lh_py.git -b master /your/folder

2.2 Running the code from Terminal

To tun the code:: cd /u/username/work/ python GO_kg1lh.py -h

usage: GO_kg1lh.py [-h] [-d DEBUG] [-r read_uid] [-u uid_write] [-c code]

Run GO_kg1lh

optional arguments: -h, –help show this help message and exit -d DEBUG, –debug DEBUG Debug level. 0: Info, 1: Warning, 2: Debug, 3: Error, 4: Debug plus; default level is INFO

Alternatively is possible to run the code specifying the debug level to increase verbosity and show debug/warning/error messages.

By default the debug level is **INFO**

The Main control function for the code are:

-p	–pulse	Shot number to process. Shot number must be provided.
-r	–read_uid	UID to read KG1 PPF (EFIT by default is JETPPF) from. Default is "JETPPF".
-u	–uid_write	UID to write KG1 PPF data with. if this is blank then no data is written. Default = "".
-c	–code	Choose which ppf to compute (KG1L/KG1H). Default is KG1L.
-d	–DEBUG	Debug level.0: Error, 1: Warning, 2: Info, 3: Debug, 4: Debug Plus. Default is 2.
-ch	–ch	choose how many channels to use. Mostly for testing purposes. Default is 8.
-a	–algorithm	choose which algorithm to use for filtering the LIDs. User can choose between: -fortran: is the same algorithm used in the fortran code (very slow!) -rolling_mean: filters the data using a rolling mean windows -rolling_mean_pandas: same as above, using a different Python library (to test code speed) Default is rolling_mean.
-pl	–plot	**Plot data and save figs: used mostly for debugging** code. Default is True.
-t	–test	**Run in test mode. In this mode, the code will comp** if the KG1V/LIDx variables have already been validated. If –uid_write=JETPPF then no PPF will be written, to avoid over-writing data. Default = False.

Return codes:

0	All OK.
1	Some channels were unavailable for processing.
2	some channles were not validated.
5	init error.
9	No validated LID channels in KG1V.
11	All available channels have validated PPFs.
20	No KG1V data.
21	Could not read SF for KG1V.
22	Error reading KG1V line-of-sight data.
23	could not filter data.
24	could not perform time loop.
25	could not filter data.
30	No EFIT data.
31	No points in EFIT.
65	The initialisation file could not be read in.
66	Problem reading the geometry file.
67	Failed to write PPF.
71	Invalid shot number.
72	No PPF exists for shot.
100	TEST MODE - NO PPF IS WRITTEN.

CHAPTER

THREE

MAIN CONTROL FUNCTION

3.1 kg1_main module

Class that runs CORMAT_py GUI

GO_kg1lh.**myself**()

GO_kg1lh.**map_kg1_efit_RM_pandas**(*arg*)
new algorithm to filter kg1v/lid data using pandas rolling mean

the sampling windown is computed as ratio between the "old fortran" rampling and the kg1v sampling :param arg: :return:

GO_kg1lh.**map_kg1_efit_RM**(*arg*)
new algorithm to filter kg1v/lid data using rolling mean the sampling window is computed as ratio between the "old fortran" rampling and the kg1v sampling :param arg: :return:

GO_kg1lh.**map_kg1_efit**(*arg*)
original algorithm used in kg1l fortran code to filter kg1v/lid data :param arg: :return:

GO_kg1lh.**time_loop**(*arg*)
computes time loop on efit time base. calls flush every step to initialise, get x-point flux, get intersections with Line of sight, get tangent flux to line of sight

Parameters **arg** –

Returns

GO_kg1lh.**main**(*shot_no*, *code*, *read_uid*, *write_uid*, *number_of_channels*, *algorithm*, *plot*, *test=False*)
Program to calculate the line averaged density for all channels of C kg1v. Other outputs are the tangent flux surface C and the distance to the edge divided by the chord length (curvature C of the edge for channel 4). :param shot_no:shot number to be processed :param code: KG1L or KG1H :param read_uid: UID for reading PPFs :param write_uid: UID for writing PPFs :param number_of_channels: number of channel to process (testing purposes only) :param algorithm: choose what algorithm to use for density filtering :param plot: True if user wants to plot data :param test: Set to True for testing mode, meaning the following:

- If write_uid is given and it is NOT JETPPF then a PPF will be written.

Returns

Return Codes:

0 : All OK 1 : Some channels were unavailable for processing. 2 : some channles were not validated 5 : init error 9 : No validated LID channels in KG1V 11 : All available channels already have validated PPF data. 20 : No KG1V data 21 : Could not read SF for KG1V 22 : Error reading KG1V line-of-sight data 23 : could not filter data 24 : could not perform time loop 25 : could not filter data 30 : No EFIT data 31 : No points in EFIT

65 : The initialisation file could not be read in. 66 : Problem reading the geometry file. 67 : Failed to write PPF. 71 : Invalid shot number 72 : No PPF exists for shot 100: TEST MODE - NO PPF IS WRITTEN

GO_kg1lh.**bin_**(*QTextStream*) → QTextStream

GO_kg1lh.**hex_**(*QTextStream*) → QTextStream

GO_kg1lh.**oct_**(*QTextStream*) → QTextStream

CHAPTER

FOUR

MODULES

4.1 consts module

Class for reading in and storing kg1 constants, signal names etc.

class consts.**Consts**(*config_name*, *code_version*)

Bases: object

DFR_DCN = 1.143e+19

DFR_MET = 1.876136e+19

MAT11 = 9.088193e+18

MAT12 = -5.536807e+18

MAT21 = 5.754791e-05

MAT22 = -9.445996e-05

CORR_NE = array([-5.60e+18, 9.10e+18, 3.50e+18, 1.46e+19, 2.58e+19, 2.02e+19, -2.10e+18, -7.60e+18, 3.49e+19, 2.37e+19,

CORR_VIB = array([-9.513e-05, 5.770e-05, -3.740e-05, 1.528e-04, 3.438e-04, 2.479e-04, -1.326e-04, -2.277e-04, 4.007e-04, 2.

FJ_DCN = array([0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3])

FJ_MET = array([1, 0, 1, -1, -3, -2, 2, 3, -3, -1, 1, 3, -2, -1, 1, 2])

JXB_FAC = [-1.6e-05, -3e-05, -4.5e-05, -3e-05]

get_phase_node_dcn(*chan*, *sig_type*)

Return the appropriate JPF node name for the DCN phase, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1r, kg1c_ldraw, kg1c_ld, kg1c_ldcor or kg1v

Returns JPF node name

get_phase_node_met(*chan*, *sig_type*)

Return the appropriate JPF node name for the MET phase, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1r, kg1c_ldraw, kg1c_ld, kg1c_ldcor or kg1v

Returns JPF node name

get_amp_node_dcn (*chan*, *sig_type*)
Return the appropriate JPF node name for the DCN amplitude signal, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1r, kg1c_ldraw, kg1c_ld, kg1c_ldcor or kg1v

Returns JPF node name

get_amp_node_met (*chan*, *sig_type*)
Return the appropriate JPF node name for the MET amplitude signal, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1r, kg1c_ldraw, kg1c_ld, kg1c_ldcor or kg1v

Returns JPF node name

get_sts_node_dcn (*chan*, *sig_type*)
Return the appropriate JPF node name for the DCN KG1C STS signal, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1c_ldraw, kg1c_ld or kg1c_ldcor

Returns JPF node name

get_sts_node_met (*chan*, *sig_type*)
Return the appropriate JPF node name for the MET KG1C STS signal, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1c_ldraw, kg1c_ld or kg1c_ldcor

Returns JPF node name

get_fj_node_dcn (*chan*, *sig_type*)
Return the appropriate JPF node name for the DCN KG1C FJ signal, given the signal type & channel number

Parameters

- **chan** – Channel number
- **sig_type** – Signal type: kg1c_ldraw, kg1c_ld or kg1c_ldcor

Returns JPF node name

get_fj_node_met (*chan*, *sig_type*)
Return the appropriate JPF node name for the MET KG1C FJ signal, given the signal type & channel number

Parameters

- **chan** – Channel number

- **sig_type** – Signal type: kg1c_ldraw, kg1c_ld or kg1c_ldcor

Returns JPF node name

set_time_windows(*ip_times*, *nbi_times*, *flattop_times*)
Set time windows

Parameters

- **ip_times** – [start ip, end ip]
- **nbi_times** – [start_nbi, end_nbi]
- **flattop_times** – [start_flat, end_flat]

4.2 library module

library.**test_logger**()
function to test logger :return:

library.**reconnect**(*signal*, *newhandler=None*, *oldhandler=None*)
function used to connect a signal to a different handler :param signal: :param newhandler: :param oldhandler: :return:

library.**is_empty**(*any_structure*)

library.**are_eq**(*a*, *b*)
checks if two lists are equal :param a: :param b: :return:

library.**autoscale_data**(*ax*, *data*)
autoscale plot :param ax: :param data: :return:

library.**find_nearest**(*array*, *value*)

Parameters

- **array** –
- **value** –

Returns returns value and index of the closest element in the array to the value

library.**find_in_list_array**(*array*, *value*)

library.**find_listelements_in_otherlist2**(*list1*, *list2*, *tstep*)

Parameters

- **list1** –
- **list2** –
- **tstep** – minimum distance between two data points

Returns

library.**find_within_range**(*array*, *minvalue*, *maxvalue*)

library.**norm**(*data*)
normalise data

library.**normalise**(*signal*, *kg1_signal*, *dis_time*)

Parameters

- **signal** – second trace

- **kg1_signal** – KG1 signal
- **dis_time** – disruption time

Returns Use ratio of maximum of signal - kg1 as the normalisation factor. Exclude region around the disruption.

library.**get_seq**(*shot_no*, *dda*, *read_uid='JETPPF'*)

Parameters

- **shot_no** – pulse number
- **dda** –
- **read_uid** –

Returns get sequence of a ppf

library.**get_min_max_seq**(*shot_no*, *dda='KG1V'*, *read_uid='JETPPF'*)

Parameters

- **shot_no** –
- **dda** –
- **read_uid** –

Returns return min and max sequence for given pulse, dda and readuid

min is the unvalidated sequence max is the last validated sequence

library.**check_SF**(*read_uid*, *pulse*)

Parameters

- **read_uid** –
- **pulse** –

Returns list of Status Flags

library.**extract_history**(*filename*, *outputfile*)

running this script will create a csv file containing a list of all the ppf that have been created with Cormat_py code

the script reads a log file (generally in /u/user/work/Python/Cormat_py)

and writes an output file in the current working directory

the file is formatted in this way shot: {} user: {} date: {} seq: {} by: {} user is the write user id by is the userid of the user of the code the output is appended and there is a check on duplicates

if the user have never run KG1_py code the file will be empty

Parameters

- **filename** – name of KG1L (or KG1H) diary to be read
- **outputfile** – name of the output file

Returns

library.**check_string_in_file**(*filename*, *string*)

Parameters

- **filename** –

- **string** –

Returns checks if the string is in that file

library.**equalsFile**(*firstFile*, *secondFile*, *blocksize=65536*)

Parameters

- **firstFile** –
- **secondFile** –
- **blocksize** –

Returns returns True if files are the same,i.e. secondFile has same checksum as first

library.**pyqt_set_trace**()
Set a tracepoint in the Python debugger that works with Qt

library.**copy_changed_kg1_to_save**(*src*, *dst*, *filename*)

Parameters

- **src** –
- **dst** –
- **filename** –

Returns copies file from src folder to dst

library.**delete_files_in_folder**(*folder*)

4.3 ppf_write

Wrapper for opening, writing & closing a PPF.

TO DO: Implement Time-Dependent Status Flags. I don't seem to be able to set tdsf's AND specify an itref in order to use a previous dtype's time-vector.

added output file with log shot,user,date,seq

ppf_write.**check_uid**(*shot_no*, *write_uid*)
Open PPF to check if UID is valid, then abort

Parameters

- **shot_no** – shot number
- **write_uid** – write_uid

Returns 1 if UID is invalid, 0 otherwise

ppf_write.**open_ppf**(*shot_no*, *write_uid*, *comment='CORRECTED KG1 DATA FROM KG1C AND KG1R '*)
Open PPF for writing

Parameters

- **shot_no** – shot number
- **write_uid** – write UID

Returns error code from PPF system. It will be 0 if there is no error.

ppf_write.**write_ppf**(*shot_no*, *dda*, *dtype*, *data*, *time=None*, *comment=None*, *unitd=None*, *unitt=None*, *itref=-1*, *nt=None*, *global_status=None*, *status=None*)

Write PPF DDA/DTYPE

Parameters

- **shot_no** – shot number
- **dda** – DDA
- **dtype** – DTYPE
- **data** – numpy array of data
- **time** – numpy array of time
- **comment** – comment
- **unitd** – units for data
- **unitt** – units for time
- **itref** – reference for timebase
- **nt** – size of the time vector
- **global_status** – status for the DTYPE
- **status** – time-dependent status

Returns error code (0 if everything is OK), itref for timebase written

ppf_write.**close_ppf**(*shot_no*, *write_uid*, *version*)

Close PPF

Parameters **shot_no** – shot number

Returns error code, 0 if everything is OK

4.4 status_flag module

Simple program to access status flags contains a main that creates a database for a given pulse list

status_flag.**find_disruption**(*pulse*)

given a pulse return if it has disrupted (True) or not (False) returns n/a if there is no information about disruptions :param pulse: :return: boolean

status_flag.**initread**()

initialize ppf :return:

status_flag.**GetSF**(*pulse*, *dda*, *dtype*)

Parameters

- **pulse** –
- **dda** – string e.g. 'kg1v'
- **dtype** – string e.g. 'lid3'

Returns SF := status flag

status_flag.**Getnonvalidatedpulses**(*pulselist*, *dtypelist*, *SF_validated*)

Parameters

- **pulselist** – list of pulses
- **dtypelist** – string e.g. 'lid1','lid2',...
- **SF_validated** – integer rapresenting SF for validated shots

Returns array of integer with pulse numbers,status flag list

status_flag.**GETfringejumps**(*pulse*, *FJC_dtypelist*)

Parameters

- **pulse** – pulse number
- **FJC_dtypelist** – string e.g. 'FC1','FC2',...

Returns array of integer, rapresenting fringe jumps corrections

status_flag.**main**(*pulse1*, *pulse2*, *FJthres*, *outputfilename*)

:param pulse1:intial pulse :param pulse2: final pulse :param FJthres: threshold to be used to check number of fringe jumps in pulse :param outputfilename: output database name :return: database containing all pulses inside the given interval whose median of fringe jumps exceed the given threshold and marks if there was a disruption or not

status_flag.**printdict**(*goodpulses_sorted*)

CHAPTER

FIVE

CLASSES

5.1 SignalBase

Class for reading and storing a signal from the PPF or JPF system, with functionality for filtering, resampling, and calculating the differences between adjacent time points.

signal_base.**decimate_ZP**(*x*, *q*, *n=None*, *ftype='iir'*, *axis=-1*, *zero_phase=False*)

Downsample the signal by using a filter. By default, an order 8 Chebyshev type I filter is used. A 30 point FIR filter with hamming window is used if *ftype* is 'fir'. Parameters ———- x : ndarray

The signal to be downsampled, as an N-dimensional array.

q [int] The downsampling factor.

n [int, optional] The order of the filter (1 less than the length for 'fir').

ftype [str {'iir', 'fir'}, optional] The type of the lowpass filter.

axis [int, optional] The axis along which to decimate.

zero_phase [bool] Prevent phase shift by filtering with `filtfilt` instead of `lfilter`.

y [ndarray] The down-sampled signal.

resample

The `zero_phase` keyword was added in 0.17.0. The possibility to use instances of `lti` as `ftype` was added in 0.17.0.

signal_base.**gcd**(*a*, *b*)

Compute the greatest common divisor of a and b

signal_base.**lcm**(*a*, *b*)

Compute the lowest common multiple of a and b

class signal_base.**SignalBase**(*constants*)

Bases: `object`

read_data_ppf(*dda*, *dtype*, *shot_no*, *read_bad=False*, *read_uid='JETPPF'*, *seq=0*)

Read in and store PPF data :param dda: DDA :param dtype: DTYPE :param shot_no: shot number :param read_bad: If set to true, data is read in for all sequence numbers (even status == 4) :param read_uid: UID to use to read PPF data :param seq: sequence number to read in

read_data_jpf(*signal_name*, *shot_no*, *use_64bit=False*)

Read in and store JPF data :param signal_name: Node name for JPF :param shot_no: shot number :param use_64bit: If set to true the data is stored as 64 bit float

read_data_jpf_1D (*signal_name*, *shot_no*)
Read in JPF data with only one dimension :param signal_name: signal name :param shot_no: shot number

filter_signal (*family*, *ncoeff=None*, *percent=None*, *start_time=0*, *end_time=0*)
Filter the signal using wavelet filtering :param family: wavelet family to use for filtering :param ncoeff: number of coefficients to retain in the filtering :param percent: percentage of coefficients to retain in the filtering :param start_time: Time from which to start the filtering :param end_time: Time to finish the filtering :return: numpy array containing the filtered data from start_time - end_time

get_time_inds (*start_time*, *end_time*)
Get the index of the times corresponding to start_time and end_time :param start_time: start time :param end_time: end time :return: index of start_time, index of end_time

resample_signal (*resample_method*, *new_time*)
Resample the signal, to a different timebase, by -interpolation, -zeropadding :param resample_method: method to use :return: numpy array of resampled data

get_differences (*npoints*)
Get the difference between the data npoints apart. :param npoints: Number of points over which to calculate the difference :return: numpy array of difference

get_second_differences (*npoints*)
Get the second differential over npoints :param npoints: Number of points over which to calculate the difference :return: numpy array of second differential

delete_points (*ind_points*)
Delete points with indices ind_points :param ind_points: indices of points to delete

5.2 Signalkg1

Class for reading and storing KG1 signals.

Inherits from SignalBase (signal_base.py).

Additional functionality for correcting fringe jumps and storing status flags

class signal_kg1.**SignalKg1** (*constants*, *shot_no*)
Bases: *signal_base.SignalBase*

uncorrect_fj (*corr*, *index*, *fringe_vib=None*)
Uncorrect a fringe jump by corr, from the time corresponding to index onwards. Not used ATM. Will need more testing if we want to use it... Suspect isclose is wrong. 07mar2019 used this function instead of is close as there is an issue with types and the value we are looking for sometimes are not found

Parameters

- **corr** – Correction to add to the data
- **index** – Index from which to make the correction

correct_fj (*corr*, *time=None*, *index=None*, *store=True*, *corr_dcn=None*, *corr_met=None*, *lid=None*)
Shifts all data from time onwards, or index onwards, down by corr. Either time or index must be specified

Parameters

- **corr** – The correction to be subtracted
- **time** – The time from which to make the correction (if this is specified index is ignored)
- **index** – The index from which to make the correction

- **store** – To record the correction set to True
- **corr_dcn** – Only for use with lateral channels. Stores the correction, in terms of the number of FJ in DCN laser (as opposed to in the combined density)
- **corr_met** – Only for use with lateral channels. Stores the correction, in terms of the number of FJ in the MET laser (as opposed to the correction in the vibration)

5.3 Kg1PPFData

Class to read and store KG1 PPF data for one channel. Reads in LIDX, FCX, MIRX, JXBX, TYPX

class kg1_ppf_data.**Kg1PPFData**(*constants*, *pulse*)

Bases: *signal_base.SignalBase*

read_data(*shot_no*, *read_uid='JETPPF'*)

Read in PPF data for KG1V for a given channel :param shot_no: shot number :param chan: channel :param read_uid: read UID :return: True if data was read in successfully, False otherwise

get_coord(*shot_no*)

Get vacuum vessel temperature & extract spatial coordinates of KG4 chords from text file. Function copied from A. Boboc's kg4r_py code.

Parameters **shot_no** – shot number

get_jpf_point(*shot_no*, *node*)

Get a single value from the JPF ie. Convert Nord data to real number Function copied from A. Boboc's kg4r_py code.

Parameters

- **shot_no** – shot number
- **node** – JPF node

5.4 MagData

Class to read and store magnetics data

class mag_data.**MagData**(*constants*)

Bases: object

MIN_IP = 0.3

PER_IP_FLAT = 0.8

CBVAC = 5.1892e-05

MIN_BVAC = -1.5

read_data(*shot_no*)

Read in magnetics data

Parameters **shot_no** – shot number

Returns True if data was read successfully and there is ip False otherwise.

_find_ip_times()

Find the start and end time of the ip and the flat-top.

Returns False if there is no IP True otherwise

_find_bvac_times()
Find the start and end time of the Bvac > 1.5T

Returns False if there is no Bvac True otherwise

5.5 KG1LData

Class to read and store all efit data

class efit_data.**EFITData**(*constants*)
Bases: *signal_base.SignalBase*

read_data(*shot_no*, *read_uid='JETPPF'*)
Read in efit (RMAG)

Parameters **shot_no** – shot number

5.6 EFITData

Class to read and store all efit data

class efit_data.**EFITData**(*constants*)
Bases: *signal_base.SignalBase*

read_data(*shot_no*, *read_uid='JETPPF'*)
Read in efit (RMAG)

Parameters **shot_no** – shot number

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

c

e

g

k

l

p

s

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W