MARIGOLD

Release 0.0.1

adix

Apr 05, 2024

CONTENTS:

*M*ultiphase *A*nalysis of *R*aw *I*nformation for *G*lobal *O*r *L*ocal *D*ata

CHAPTER

ONE

MARIGOLD PACKAGE

1.1 Functions

color_cycle()	Custom generator for colors
deepcopy(x[, memo, _nil])	Deep copy operation on arbitrary Python objects.
dump_data_from_tabs([dump_file, skip_dir])	
extractIskandraniData([dump_file])	
extractLocalDataFromDir(path[, dump_file, ...])	Function for getting all local data from spreadsheets in a directory, path
extractPitotData([dump_file, in_dir, ...])	
extractProbeData([dump_file, in_dir, ...])	
extractYangData([dump_file])	
loadData(data_file)	
loadIskandraniData([data_file])	
loadPitotData([data_file])	
loadProbeData([data_file])	
loadYangData([data_file])	
marker_cycle()	Custom generator for markers

1.1.1 color_cycle

MARIGOLD.**color_cycle**()

Custom generator for colors

1.1.2 dump_data_from_tabs

`MARIGOLD.`**`dump_data_from_tabs`**(*dump_file='PITA_Database.dat'*, *skip_dir=''*) → None

1.1.3 extractIskandraniData

`MARIGOLD.`**`extractIskandraniData`**(*dump_file='Iskandrani_Database.dat'*) → None

1.1.4 extractLocalDataFromDir

`MARIGOLD.`**`extractLocalDataFromDir`**(*path: str*, *dump_file='database.dat'*, *in_dir=[]*, *require_terms=['jf']*, *skip_terms=['CFD', 'Copy']*, *sheet_type='adix_template'*, *append_to_json=None*, *pitot_sheet=False*, ***kwargs*) → None

Function for getting all local data from spreadsheets in a directory, path

Does not recursively descend, only checks in the path given

Still under construction, but should support sheet types 'adix_template4' 'ryan_template' 'adix_template' (maybe rename this quan_template)

Also can try to infer the sheet type, xlsm will be adix_template4, if it has P5, 6, or 7 it will be classified as an adix_template, different angles ryan, etc. The inference can also be made by appending _adix or _quan or _ryan to the excel sheets being processed

Custom sheet types may be supported, but they still have to generally follow the classic template structure. The setup information has to be in sheet '1' and all the local data in sheet '2'. Ranges must be specified as a list of lists

Q1_ranges = [[angle1, [index1, index2.,..]], [angle2, [index1, index2.,..]], . . .]

with the starts and ends

[Q1_start, Q1_end, Q2_start, . . .]

Add on pitot starts/ends if pitot_sheet = True

Need to specify Q1_check, Q2_check, etc. as well

1.1.5 extractPitotData

`MARIGOLD.`**`extractPitotData`**(*dump_file='Pitot_Database.dat'*, *in_dir=[]*, *require_terms=[]*, *skip_terms=['CFD', 'Copy']*) → None

1.1.6 extractProbeData

`MARIGOLD.`**`extractProbeData`**(*dump_file='database.dat'*, *in_dir=[]*, *require_terms=None*, *skip_terms=['CFD', 'Copy']*) → None

1.1.7 extractYangData

`MARIGOLD.`**`extractYangData`**`(`*dump_file='Yang_Database.dat'*`)` → list

1.1.8 loadData

`MARIGOLD.`**`loadData`**`(`*data_file*`)` → list

1.1.9 loadIskandraniData

`MARIGOLD.`**`loadIskandraniData`**`(`*data_file='Iskandrani_Database.dat'*`)` → list

1.1.10 loadPitotData

`MARIGOLD.`**`loadPitotData`**`(`*data_file='Pitot_Database.dat'*`)` → list

1.1.11 loadProbeData

`MARIGOLD.`**`loadProbeData`**`(`*data_file='PITA_Database.dat'*`)` → list

1.1.12 loadYangData

`MARIGOLD.`**`loadYangData`**`(`*data_file='Yang_Database.dat'*`)` → list

1.1.13 marker_cycle

`MARIGOLD.`**`marker_cycle`**`()`

Custom generator for markers

1.2 Classes

`Condition`(jgref, jgloc, jf, theta, port, ...)	Class to handle the local probe data
`Iskandrani_Condition`(jf, jg)	
`Yang_Condition`(jf, jg)	
`datetime`(year, month, day[, hour[, minute[, ...)	The year, month and day arguments are required.

1.2.1 Condition

class `MARIGOLD.`**`Condition`**(*jgref: float*, *jgloc: float*, *jf: float*, *theta: int*, *port: str*, *database: str*)

Bases: `object`

Class to handle the local probe data

Data is stored in the Condition.phi property. It's actually 3 layers of dictionary phi [angle] gives a dictionary with the various r/R phi [angle][r/R] gives a dictionary with the MIDAS output The MIDAS output is itself a dictionary, with the keys listed in the "tab_keys" array So phi[angle][r/R]['alpha'] should give you the void fraction at r/R for phi = angle This structure is initialized with zeros for the MIDAS output at the pipe center and wall

Attributes Summary

`debugFID`	

Methods Summary

`TD_FR_ID`()	
`__call__`(phi_in, r_in, param[, interp_method])	Returns the value of param at (phi, r). Can get raw data, linear interp, or spline interp
`approx_vf`([n])	Method for approximating vf with power-law relation.
`approx_vf_Kong`([n])	Method for approximating vf from Kong.
`area_avg`(param[, even_opt, recalc])	Method for calculating the area-average of a parameter, "param"
`calc_W`()	Calculates the wake deficit function, W, from the experimental data
`calc_avg_lat_sep`()	Calculates average lateral separation distance between bubbles
`calc_cd`([method, rho_f, vr_cheat, mu_f])	Method for calculating drag coefficient
`calc_dpdz`([method, rho_f, rho_g, mu_f, ...])	Calculates the pressure gradient, dp/dz, according to various methods.
`calc_errors`(param1, param2)	Calculates the errors, , between two parameters (param1 - param2) in midas_dict
`calc_grad`(param[, recalc])	Calculates gradient of param based on the data in self.
`calc_linear_interp`(param)	Makes a LinearNDInterpolator for the given param.
`calc_linear_xy_interp`(param)	Makes a LinearNDInterpolator for the given param in x y coords
`calc_mu3_alpha`()	Calculates the third moment of alpha
`calc_mu_eff`([method, mu_f, mu_g, alpha_max])	Method for calculating effective viscosity.
`calc_sigma_alpha`()	Calculates the second moment of alpha
`calc_vgj`([warn_approx])	Method for calculating Vgj, by doing
`calc_vgj_model`()	Method for calculating Vgj based on models
`calc_void_cov`()	Calculates the void covariance
`calc_vr`([warn_approx])	Method for calculating relative velocity.

continues on next page

Table 1 – continued from previous page

`calc_vr_model`([method, c3, n, iterate_cd, quiet])	Method for calculating relative velocity based on models
`calc_vwvg`()	Calculates void weighted Vgj
`circ_segment_area_avg`(param, hstar[, ...])	Method for calculating the area-average of a parameter, "param" over the circular segment defined by h
`circ_segment_void_area_avg`(param, hstar[, ...])	Method for calculating the void-weighted area-average of a parameter over the circular segment defined by h
`find_hstar_pos`([method, void_criteria])	Returns the vertical distance from the top of the pipe to the bubble layer interface
`fit_spline`(param)	Fits a RectBivariateSpline for the given param.
`interp_area_avg`(param[, interp_type])	Function to area-average param, using the spline interpolation of param
`line_avg`(param, phi_angle[, even_opt])	Line average of param over line defined by phi_angle
`line_avg_dev`(param, phi_angle[, even_opt])	Second moment of param over line defined by phi_angle
`max`(param[, recalc])	Return maximum value of param in the Condition
`max_line`(param, angle)	Return maximum value of param at a given angle
`max_line_loc`(param, angle)	Return r/R location of maximum value of param at a given angle
`max_loc`(param)	Return location of maximum param in the Condition
`min`(param[, recalc, nonzero])	Return minimum value of param in the Condition
`min_loc`(param)	Return minimum value of param in the Condition
`mirror`([sym90, axisym, uniform_rmesh, ...])	Mirrors data, so we have data for every angle
`plot_contour`(param[, save_dir, show, ...])	Method to plot contour of a given param
`plot_isoline`(param, iso_axis, iso_val[, ...])	Plot profiles of param over iso_axis at iso_val
`plot_profiles`(param[, save_dir, show, ...])	Plot profiles of param over x_axis, for const_to_plot, i.e. over r/R for = [90, 67.5 .
`plot_spline_contour`(param[, save_dir, show, ...])	Plots a contour from a spline interpolation
`plot_surface`(param[, save_dir, show, ...])	Method to plot a surface of a given param
`pretty_print`([print_to_file, FID, mirror])	Prints out all the information in a Condition in a structured way
`rough_FR_ID`()	Identifies the flow regime for the given condition, by some rough methods
`spline_circ_seg_area_avg`(param, hstar[, int_err])	Function to area-average over a circular segment defined by h, using the spline interpolation of param
`spline_void_area_avg`(param)	Function to void-weighted area-average param over a circular segment defined by h, using the spline interpolation of param
`top_bottom`(param[, even_opt])	Honestly, I forgot what this does
`void_area_avg`(param[, even_opt])	Method for calculating the void-weighted area-average of a parameter

Attributes Documentation

`debugFID = None`

Methods Documentation

`TD_FR_ID()` → None

`__call__`(*phi_in: ndarray*, *r_in: ndarray*, *param: str*, *interp_method='None'*) → ndarray

Returns the value of param at (phi, r). Can get raw data, linear interp, or spline interp
phi in radians

Can also return the value at (x, y) if linear_xy is selected as the interp method. phi -> x, r -> y

`approx_vf`(*n=7*) → None

Method for approximating vf with power-law relation.

vf_approx = (n+1)*(2*n+1) / (2*n*n) * (jf / (1-self.area_avg('alpha'))) * (1 - abs(rstar))**(1/n)

Will not overwrite vf data if it already exists, but will always store data in midas_dict['vf_approx'] even if midas_dict['vf'] has data

`approx_vf_Kong`(*n=7*) → None

Method for approximating vf from Kong. TODO

Not currently implemented

`area_avg`(*param: str*, *even_opt='first'*, *recalc=True*) → float

Method for calculating the area-average of a parameter, "param"

param can be anything MIDAS outputs, but usually of interest are "alpha" or "alpha_ug" If you're not sure what somethings named, try Condition.phi[90][1.0].keys()

Uses Simpson's rule for integration, even_opt passed to that. Will save the previously calculated area averages in Condition.area_avgs[param], and won't recalculate unless recalc = True

Returns the area-averaged parameter, or None if the method failed

`calc_W()`

Calculates the wake deficit function, W, from the experimental data

`calc_avg_lat_sep()`

Calculates average lateral separation distance between bubbles

Average lateral separation, given by = ugl / f

stores in midas dict under lambda. Also estimates based on ,

 ~ Db /

Mirrors the data

`calc_cd`(*method='Ishii-Zuber'*, *rho_f=998*, *vr_cheat=False*, *mu_f=0.001*)

Method for calculating drag coefficient

If vr = 0, assume cd = 0

Options are Ishii-Zuber and Schiller-Naumann, but both use Reb = (1 - midas_dict['alpha']) * midas_dict['Dsm1'] * rho_f * midas_dict['vr'] / midas_dict['mu_m'] vr from calc_vr() mu_m from calc_mu_eff()

calc_dpdz(*method='LM'*, *rho_f=998*, *rho_g=1.225*, *mu_f=0.001*, *mu_g=1.18e-05*, *LM_C=25*)

Calculates the pressure gradient, dp/dz, according to various methods. Can access later with self.dpdz

'LM' -> Lockhart Martinelli, assuming turbulent-turbulent, C = LM_C

calc_errors(*param1: str*, *param2: str*)

Calculates the errors, , between two parameters (param1 - param2) in midas_dict

Stores:

- error, "eps_param1_param2", param1 - param2
- relative, "eps_rel_param1_param2", (param1 - param2) / param2
- absolute relative, "eps_abs_rel_param1_param2", **|param1 - param2|** / param2
- square, "eps_sq_param1_param2", (param1 - param2)**2
- relative square, "eps_rel_sq_param1_param2", ((param1 - param2)/param2)**2

If param2 = 0, relative errors are considered 0

calc_grad(*param: str*, *recalc=False*) → None

Calculates gradient of param based on the data in self.

Stored in self's midas_dict as grad_param_r, grad_param_phi, etc. Will only be called once, unless recalc is True.

calc_linear_interp(*param: str*) → None

Makes a LinearNDInterpolator for the given param.

Access with self.linear_interp[param], phi in radians

calc_linear_xy_interp(*param: str*) → None

Makes a LinearNDInterpolator for the given param in x y coords

Can access with self.linear_xy_interp[param]

calc_mu3_alpha()

Calculates the third moment of alpha

Returns <(alpha-<alpha>)^3>/<alpha>^3

Stored in self.mu3_alpha

calc_mu_eff(*method='Ishii'*, *mu_f=0.001*, *mu_g=1.803e-05*, *alpha_max=1.0*)

Method for calculating effective viscosity.

Also calculates mixture viscosity, stored in _eff and _m, resepectively. Note that this fuction does mirror the data

Right now the only method implemented is Ishii's

calc_sigma_alpha()

Calculates the second moment of alpha

Returns <(alpha-<alpha>)^2>/<alpha>^2

Stored in self.sigma_alpha

`calc_vgj`(*warn_approx=True*) → None

Method for calculating Vgj, by doing

j_local = midas_dict['alpha'] * midas_dict['ug1'] + (1 - midas_dict['alpha']) * midas_dict['vf'] vgj = midas_dict['ug1'] - j_local

stored in midas_dict['vgj']

warn_approx is a flag to print out a warning statement if vf is being approximated, which will happen if not found

`calc_vgj_model`()

Method for calculating Vgj based on models

midas_dict['vgj_model'] = (1 - midas_dict['alpha']) * midas_dict['vr_model']

`calc_void_cov`()

Calculates the void covariance

Returns <alpha^2>/<alpha>^2

Stored in self.void_cov

`calc_vr`(*warn_approx=True*) → None

Method for calculating relative velocity. Will approximate vf if it cannot be found.

Note that if vg = 0, then this method says vr = 0. This will happen when no data is present, such as in the bottom of the pipe in horizontal, when this is not necessarily true

warn_approx is a flag to print out a warning statement if vf is being approximated

`calc_vr_model`(*method='wake_1'*, *c3=-0.15*, *n=1*, *iterate_cd=True*, *quiet=True*)

Method for calculating relative velocity based on models

Stored under "vr_method" in midas_dict as well as "vr_model"

TODO implement Ishii-Chawla

Implemented options: - "wake_1" vr = - c3 * vf * Cd**(1./3) - "wake_alpha" vr = - c3 * (1-)^n * vf * Cd**(1./3) - "wake_alpha2" vr = - c3 * (*(1-))^n * vf * Cd**(1./3) - "wake_lambda" = - c3 * vf * Cd**(1./3) * Db**(2./3) * **(-2./3)

`calc_vwvg`() → None

Calculates void weighted Vgj

uses self.jgloc / self.area_avg('alpha'), which is not a great method

`circ_segment_area_avg`(*param: str*, *hstar: float*, *ngridr=25*, *ngridphi=25*, *int_err=0.0001*) → float

Method for calculating the area-average of a parameter, "param" over the circular segment defined by h

Basically, if you slice the pipe at h, area average everything above h

to smooth it out, integrate over an interpolated mesh

returns integrand result

`circ_segment_void_area_avg`(*param: str*, *hstar: float*, *ngridr=25*, *ngridphi=25*, *int_err=0.0001*) → float

Method for calculating the void-weighted area-average of a parameter over the circular segment defined by h

Basically, if you slice the pipe at h, void-weighted area average everything above h

to smooth it out, integrate over an interpolated mesh

returns integrand result

`find_hstar_pos`(*method='max_dsm'*, *void_criteria=0.05*) → float

Returns the vertical distance from the top of the pipe to the bubble layer interface

Methods for determining bubble layer interface - max_dsm - min_grad_y - max_grad_y - max_mag_grad_y - zero_void - percent_void, search down the phi = 90 line, find largest value of rstar where alpha < min_void - Ryan_Ref, uses 1.3 - 1.57e-5 * Ref, proposed by Ryan (2022)

`fit_spline`(*param: str*) → None

Fits a RectBivariateSpline for the given param.

Can access later with self.spline_interp[param]. Must specify the 'param' to fit

`interp_area_avg`(*param: str*, *interp_type='linear'*) → float

Function to area-average param, using the spline interpolation of param

Returns the intengrand result. May be computationally expensive

`line_avg`(*param: str*, *phi_angle: float*, *even_opt='first'*) → float

Line average of param over line defined by phi_angle

Also includes the complementary angle, so the line is a diameter of the pipe

returns integrand result

`line_avg_dev`(*param: str*, *phi_angle: float*, *even_opt='first'*) → float

Second moment of param over line defined by phi_angle

Also includes the complementary angle, so the line is a diameter of the pipe

< param - <param>^2 > / <param>^2

returns integrand result

`max`(*param: str*, *recalc=False*) → float

Return maximum value of param in the Condition

By default, saves the data to a dictionary, Condition.maxs for future reference, unless recalc=True

`max_line`(*param: str*, *angle: float*) → float

Return maximum value of param at a given angle

`max_line_loc`(*param: str*, *angle: float*) → float

Return r/R location of maximum value of param at a given angle

`max_loc`(*param: str*) → tuple

Return location of maximum param in the Condition

returns in the form (r/R, angle in degrees)

`min`(*param: str*, *recalc=False*, *nonzero=False*) → float

Return minimum value of param in the Condition

By default, saves the data to a dictionary, Condition.mins for future reference, unless recalc=True

nonzero=True will find the smallest nonzero value.

`min_loc`(*param: str*) → float

Return minimum value of param in the Condition

By default, saves the data to a dictionary, Condition.mins for future reference, unless recalc=True

nonzero option not implemented, TODO

`mirror`(*sym90=True*, *axisym=False*, *uniform_rmesh=False*, *force_remirror=False*) → None

Mirrors data, so we have data for every angle

First finds all the angles with data, copies anything negative to the other side (deleting the negative entries in the original). Then goes though each of the angles in _angles (22.5° increments) and makes sure each has data. Either copying, assuming some kind of symmetry (either axisym or sym90) or just filling in zeros.

uniform_rmesh will ensure every angle has data for every r/R point. Will linearly interpolate when data on either side is available

Force_remirror is untested, no clue if it's safe or not

Also saves the original mesh (r,) pairs under self.original_mesh

Quadrant definitions:

phi = 90

, - ~ ~ ~ - ,

, ' | ' ,

, | ,

, II | I ,

, | ,

180 ,————————-|————————-, 0

, | ,

, III | IV ,

, | ,

, | , '

' - , _ _ _ , '
270

`plot_contour`(*param: str*, *save_dir='.'*, *show=True*, *set_max=None*, *set_min=None*, *fig_size=4*, *label_str=None*, *rot_angle=0*, *ngridr=50*, *ngridphi=50*, *colormap='hot_r'*, *num_levels=100*, *title=False*, *title_str=''*, *extra_text=''*, *annotate_h=False*, *cartesian=False*, *h_star_kwargs={'method': 'max_dsm', 'min_void': '0.05'}*, *plot_measured_points=False*) → None

Method to plot contour of a given param

Generates a contour plot of any parameter in midas_dict, e.g. 'alpha', 'ai', etc. By default, just shows the figure, but if a save_dir is specified, it will save it there instead. label_str can adjust the label of the colorbar. Can accept Latex format, e.g. r"$lpha$ [-]"

set_max and set_min set the bounds of the contour plot, and the colormap option allows for any colors that matplotlib supports. ngridr, ngridphi, num_levels, all adjust how fine the contour plot is generated.

annotate_h is an option to draw a horizontal line at some given position. This was implemented when investigating where the bubble layer typically stops. h_star_kwargs and cartesian are optinos related to this.

plot_measured points is neat, it plots circles where original data was detected (determined prior to mirroring) `plot_isoline`(*param: str*, *iso_axis: str*, *iso_val: float*, *fig_size=4*, *plot_res=100*, *save_dir='.'*, *show=True*, *extra_text=''*) → None

Plot profiles of param over iso_axis at iso_val

Based on interpolation, so plot_res changes the resolution

`plot_profiles`(*param*, *save_dir='.'*, *show=True*, *x_axis='r'*, *const_to_plot=[90, 67.5, 45, 22.5, 0]*, *include_complement=True*, *rotate=False*, *fig_size=4*, *title=True*) → None

Plot profiles of param over x_axis, for const_to_plot, i.e. over r/R for = [90, 67.5 … 0].

Include_complement will continue with the negative side if x_axis = 'r'

Also has an option to rotate the graph based on self.theta, but it's a little sketchy

`plot_spline_contour`(*param: str*, *save_dir='.'*, *show=True*, *set_max=None*, *set_min=None*, *fig_size=4*, *rot_angle=0*, *ngridr=50*, *ngridphi=50*, *colormap='hot_r'*, *num_levels=100*, *title=False*, *annotate_h=False*, *cartesian=False*, *h_star_kwargs={'method': 'max_dsm', 'min_void': '0.05'}*, *grad='None'*) → None

Plots a contour from a spline interpolation

Will fit the spline if necessary.

`plot_surface`(*param: str*, *save_dir='.'*, *show=True*, *rotate_gif=False*, *elev_angle=145*, *azim_angle=0*, *roll_angle=180*, *title=True*, *ngridr=50*, *ngridphi=50*, *plot_surface_kwargs=None*, *solid_color=False*, *label_str=None*, *title_str=''*) → None

Method to plot a surface of a given param

Can save a static image or rotating gif, starting at elev_angle, azim_angle, roll_angle. These angles also the viewing angle for the static image.

Can specify a label or title str.

Plot_surface_kwargs is how to specify vmin, vmax, colormap, etc.

`pretty_print`(*print_to_file=False*, *FID=None*, *mirror=False*) → None

Prints out all the information in a Condition in a structured way

Specifically, everything in the Condition.phi dictionary, which has angles and r/Rs.

Can either print to a file (specified by FID) or to stdout. Option to mirror the data, if that hasn't already been done

`rough_FR_ID`() → None

Identifies the flow regime for the given condition, by some rough methods

First checks if it matches any given by previous researchers, or the hierarchical clustering algorithm results

Stored in self.FR

1 = bubbly 2 = plug 3 = slug 4 = churn 5 = stratified 6 = stratified wavy 7 = annular

`spline_circ_seg_area_avg`(*param: str*, *hstar: float*, *int_err=0.0001*) → float

Function to area-average over a circular segment defined by h, using the spline interpolation of param

Returns the intengrand result. May be computationally expensive

`spline_void_area_avg`(*param: str*) → float

Function to void-weighted area-average param over a circular segment defined by h, using the spline interpolation of param

Returns the intengrand result. May be computationally expensive

top_bottom(*param*, *even_opt='first'*) → float

Honestly, I forgot what this does

I think it area-averages the way Bottin did, which is not a good way of doing so.

void_area_avg(*param: str*, *even_opt='first'*) → float

Method for calculating the void-weighted area-average of a parameter

<alpha * param> / <alpha>

1.2.2 Iskandrani_Condition

class MARIGOLD.**Iskandrani_Condition**(*jf*, *jg*)

Bases: `object`

Methods Summary

`area_avg`(param)	

Methods Documentation

area_avg(*param*)

1.2.3 Yang_Condition

class MARIGOLD.**Yang_Condition**(*jf*, *jg*)

Bases: `Condition`

Methods Summary

`pretty_print`()	Prints out all the information in a Condition in a structured way

Methods Documentation

pretty_print() → None

Prints out all the information in a Condition in a structured way

Specifically, everything in the Condition.phi dictionary, which has angles and r/Rs.

Can either print to a file (specified by FID) or to stdout. Option to mirror the data, if that hasn't already been done

1.3 Class Inheritance Diagram

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX