GliderFlight Documentation

Release 1.1.0

Lucas Merckelbach

Aug 30, 2023

Contents:

This manual covers the use of the Python module **GliderFlight**.

GliderFlight is written in Python3, and is released as open source software under the MIT License.

CHAPTER 1

GliderFlight for Slocum ocean gliders

1.1 Synopsis

Gliderflight is a python module to calibrate a model that predicts the glider flight through water. The model results can be used to estimate the speed through water, a parameter which is required to compute turbulent dissipation rates from temperature microstructure or shear probe data, collected with a turbulence profiler mounted on top of an ocean glider.

1.2 Changelog

Version 1.2.0

- Added paralel computing of solution for dynamic model
- Added logging module for reporting of diagnostic messages
- The named typle model_result now contains depth
- Several small bug fixes

Version 1.0.1

- Small bug fixes

Verions 1.0.0

- Initial release

1.3 Background

The dissipation rate of turbulent kinetic energy is a parameter that plays a key role in many physical and biogeo chemical processes in oceans and coastal seas. However, direct oceanic measurements of turbulence are relatively scarce, as most observations stem from free-falling profilers, operated from seagoing vessels.

An emerging alternative to ship-based profiling is the use of ocean gliders with mounted turbulence profilers. A required parameter in the processing of microstructure shear and temperature measurements is the speed of flow past the sensors. This speed can be measured directly with additional sensor, such as an electromagnetic current meter or mounted acoustic Doppler current profiler, but often gliders are not equipped with additional velocity sensors. Alternatively, a glider flight model can be used to estimate the speed through water. Such a model is described in the paper *A dynamic flight model for Slocum gliders and implications for turbulence microstructure measurements* [merckelbach2019]. This Python model implements the steady-state and dynamic glider flight models, described therein.

1.4 Documentation

Documentation of this software package can be found at https://gliderflight.readthedocs.io/en/latest/

1.5 Steady-state model

The steady-state model implemented, considers a horizontal and vertical force balance. Vertical forces are a balance between buoyancy, gravity and the vertical components of the lift and drag forces. The horizontal force balance consists of the horizontal components of the lift and drag forces only. These two equations can be solved for the angle of attack and the speed through water, determining the flight at any instance of time.

Input to the model comes from parameters measured by the glider, such as the measured pitch angle (m_pitch), buoyancy change (m_ballast_pumped or m_de_oil_vol) and the in-situ density. Furthermore, the model requires the specification of a number of coefficients:

- mg: mass of the glider (kg)
- Vg: volume of the glider (m^3)
- Cd0: parasite drag coefficient
- epsilon: compressibility of the hull (1/Pa)
- ah: lift angle coefficient due to the hull (1/rad)
- Cd1: induced drag coefficient ($1/rad^2$)

Using the depth-rate from the pressure sensor as only model constraint, the mass (or glider volume) and the parasite drag coefficient can be determined. To determine the lift angle coefficient requires an additional constraint that contains a horizontal velocity component. Details of this procedure are given in [merckelbach2019].

1.6 Dynamic model

In addition to a steady-state model, this code also implements a dynamic model, that is, including the inertial terms. Since this model needs to be integrated, for which the Runge-Kutta method is used, it is more computational expensive. The dynamic model produces more accurate results when forcing conditions change rapidly, such as when crossing a sharp pycnocline or during the transition from dive to climb. Apart from the mathematical model underlying, the interfaces to both models are the same.

1.7 Model calibration and data masking

To calibrate a model, either steady-state or dynamic, we may wish not to include all the data in the evaluation of the cost-function. To that end, data can be masked. The Calibrate class provides boolean operators to do this:

- OR()
- AND()
- NAND()

By default a mask set to False for all data. To mask data for which a condition evaluates to True, the OR() method should be used. For example,

```
gm = SteadyStateCalibrate(rho0=1024)
gm.set_input_data(datadict)

condition = depth<10
gm.OR(condition)
```

which would exclude all data points for which the depth is less than 10 m from the evaluation of the cost-function.

A truth table:

mask	conditon	OR	AND	NAND
0	0	0	0	1
1	0	1	0	1
1	1	1	1	0
0	1	1	0	1

1.8 Example

An example to calibrate a model:

```
# create a dictionary with the data

data = dict(time=t, pressure=P, pitch=pitch, buoyancy_change=deltaV)

gm = SteadyStateCalibrate()
# we have to define mass and volume at the minimum
gm.define(mg=70, Vg=70)

gm.set_input_data(data)

# mask all data below 10 m
gm.OR(pressure*10<10)
# mask all data exceeding 60 m
gm.OR(pressure*10>60)

result = gm.calibrate("mg", "Cd0")

print("Calibrated parameters:")
for k,v in result.items():
    print("{}: {}".format(k,v)
```

(continues on next page)

(continued from previous page)

```
# Instead of printing the parameters from the results, we could also
# get them from the corresponding attributes: print("Cd0:", gm.Cd0).

print("Cd0:", gm.Cd0)

# We also don't need to run the model again either. The model output
# is also accessible from attributes:
#
# gm.t # time
# gm.U # incident velocity
# gm.alpha # angle of attack
# gm.ug    # horizontal speed
# gm.wg    # vertical speed
# gm.w     # vertical water velocity

# if we want to run a model with a given set of parameters

fm = DynamicGliderModel(dt=1, rho0=1024, k1=0.02, k2=0.92)
# copy the settings from the steady state model
fm.copy_settings(gm)

solution = fm.solve(data)

# solution is now a named tuple, according to the definition:
# Modelresult = namedtuple("Modelresult", "t u w U alpha pitch ww")
```

1.9 How to cite

When you publish results that were obtained with this software, please use the following citation:

Merckelbach, L., A. Berger, G. Krahmann, M. Dengler, and J. Carpenter, 2019: A dynamic flight model for Slocum gliders and implications for turbulence microstructure measurements. J. Atmos. Oceanic Technol., 36(2), 281-296, doi:10.1175/JTECH-D-18-0168.1.

1.10 Copyright information

1.11 References

CHAPTER 2

Installing GliderFlight

2.1 Download

The software's repository is hosted on github, from where you can download the source using git or download it as a zip file.

2.2 Installing

After having obtained the source code, you can build it using the standard way of installing python code. On linux this would be

```
$ python3 setup.py build
$ python3 setup.py install
```

Depending on your system setup, the `install` command may require root privileges.

2.2.1 Dependencies

The gliderflight module depends on numpy.

The glidertrim script – useful to check and adjust the ballast trim of glider during deployment – additionally depends on scipy, matplotlib, gsw, and dbdreader. All these packages are available from PyPi and will be downloaded automatically when not present. However, you may prefer to install the complex packages numpy, scipy and matplotlib using your distribution's package manager (when on linux).

See also the file requirements.txt in the root directory.

2.3 PyPi

Glider flight is available from pypi

```
pip install gliderflight
```

CHAPTER 3

Using the GliderFlight module

How to use the GliderFlight module is probably best done with a worked example.

First we need some data to work with. Let's say we have some data files from a Slocum glider. Typically, we would work with the high-density data files, and would need to have access to the engineering data, which are stored in files with the dbd extension, and science data, which are stored in files with the ebd extension. In this example, we use `dbdreader` to read the glider files. The Python module dbdreader can be installed from PyPi using `pip3 install --user dbdreader` or install from source from github

```
import numpy as np

import dbdreader
dbd = dbdreader.MultiDBD(pattern = "/path/to/data/*.[de]bd")
```

Now we have a handle to the data files, we need to extract the required parameters. From the engineering data, we need the pitch and the buoyancy drive. From the science data, we need the CTD parameters. Instead of relying on the time of publishing, we take the ctd sampling time *sci_ctd41cp_timestamp*. Occasionally, the CTD fields contain data that have not been sampled, and default values are returned. These default values can be detected from time stamps to be zero, for example. After reading the values, we simply remove odd ones.

```
tmp = dbd.get_sync("sci_ctd41cp_timestamp", "sci_water_temp", "sci_water_cond", "sci_
↪water_pressure", "m_pitch", "m_ballast_pumped")
_, tctd, T, C, P, pitch, buoyancy_change = np.compress(tmp[1]>0, tmp, axis=1)
```

Since version 0.4.0 of `dbdreader` we can also MultiDBD's method get_CTD_sync():

```
tctd, T, C, P, pitch, buoyancy_change = dbd.get_CTD_sync("m_pitch", "m_ballast_pumped
↪")
```

In the example, we used the sensor name *m_ballast_pumped*, which is appropriate for a shallow glider. When data from a deep glider were used, the name should be replaced by *m_de_oil_vol*.

One of the input parameters to the glider flight model is the in-situ density. So let's compute that one first. For this, we use the Gibbs Seawater module, instllable using pip (for example, `pip3 install --user gsw`), or from source from https://github.com/TEOS-10/GSW-python. Also, we need latitude and longitude information. This information

could be retrieved from the glider parameters `m_gps_lat` and `m_gps_lon`, condensed into a single scalar, as an array of same length as `T`.

```
import gsw
# C is given in S/m, and P in bar
SP = gsw.SP_from_C(C*10, T, P*10)
SA = gsw.SA_from_SP(SP, P*10, lon, lat)
rho = gsw.rho_t_exact(SA, T, P*10)
```

Now we have density, we can pack all the required data into a dictionary:

```
data = dict(time = tctd, pressure = P, pitch = m_pitch, buoyancy_change=buoyancy_
↪change, density=density)
```

Most likely, you would want to use that data to calibrate some model coefficients that change from depolyment to deployment, such as the glider volume, and drag coefficient. To that end, we create an instance from the SteadyStateCalibrate class. and populate it with the data we have got.

```
import gliderflight

gm = gliderflight.SteadyStateCalibrate(rho0=1024)
gm.set_input_data(**data)
# or, alternatively
# gm.set_input_data(tctd, P, pitch, buoyancy_change, rho)
```

When we calibrate a steady-state model, we don't want to include data points were we know the steady-state model is invalid, such as around the transitions from down to up casts, and near the surface. Assuming that we have dive profiles down to 100 m, may discard the first 20 m, and the last 20 m of the dives when optimising the model against the observed pressure rate.

```
condition = np.logical_or(P*10<20, P*10>80)
gm.OR(condition)
```

Before we can start calibrating the model, we need to set glider and deployment specific model coefficients. Let's say we weighted the glider and found its mass to be $m_g = 70$ kg. (If the mass of the glider is not know, it can be guessed. The calibration method will adjust the volume such that the glider *density* is correct. An error in the volume will have a small effect on the buoyancy force calculated. As long as the mass (or the volume) is correct withing a few percent, the errors involved are negligible.

So, we will set the mass and the volume. Also, we will set the parasite drag to a realistic value.

```
gm.define(mg=70.000, Vg=70e-3, Cd0=0.16)
if not gm.undefined_parameters():
    print("We still have undefined parameters...")
    print(gm.undefined_parameters())
```

Now we're good to run the calibration and store the results in calibration_result.

```
calibration_result = gm.calibrate("Vg", "Cd0")
```

upon which we should have a dictionary with the keys "Vg" and "Cd0" and their optimised values. The coefficients are also updated in the model itself. So, `gm.Cd0` would return the same value as reported in the dictionary.

To the the glider flight results, such as angle of attack and incident water velocity, it is not necessary to solve the model again. So, all these parameters are accessible via `gm.modelresult` or using the properties `t`, `ug`, `wg`, `alpha` and `U`.

So, we could now plot the incident water velocity as function of time:

```
import matplotlib.pyplot as plt

f, ax = plt.subplots(1,1)

ax.plot(gm.t, gm.U, label='Incident water velocity')
ax.set_xlabel('time (s)')
ax.set_ylabel('U m s$^{-1}$')
ax.legend()
```

CHAPTER 4

Glidertrim

Glidertrim is a simple commandline utility, that, given one or more pairs of dbd/ebd files, and a target density, computes the optimum weight change. The typical application is that during the deployment a test dive is made, and the resulting dbd/edb files are analysed quickly, producing an objective estimate of how much a glider is overweight or underweight, given a target density.

4.1 Synopsis

glidertrim <glidername> <dbd file> [dbd file] [dbd file]

glidername is the name or identifier of the glider. It is used to write settings in the configuration file ($HOME/.glidertrimrc), so that the settings can be prepared prior to deployment and recorded for later use.

dbd_file is a path to a dbd file and can included wildcards. It is important that the dbd filename is provided only, but the matching ebd file is present in the same directory the dbd file resides in.

4.2 Description

Glidertrim takes a glider id and one or more dbd (with matching ebd) files as input, and given some configuration settings which the user can change, calculates the ideal weight change.

An example is given below:

```
$ glidertrim comet comet-2018-136-00-000.dbd
comet-2018-136-00-000.dbd found. Ok

Enter value for target_density (kg/m^3)        (current: 1026.000000):
Enter value for mg (kg)                        (current: 69.500000):
Enter value for Vg (m^3)                       (current: 0.065000):
Enter value for minlimitdepth (m)              (current: 3.000000):
Enter value for maxlimitdepth (m)              (current: 55.000000):
```

(continues on next page)

(continued from previous page)

```
Enter value for cond_a (m/S)                    (current: 1.000000):
Enter value for cond_b (-)                      (current: 0.000000):
Enter value for buoyancy_engine (shallow|deep) (current: deep):
Enter value for latitude (decimal deg)          (current: 54.000000):
Enter value for longitude (decimal deg)         (current: 8.000000):
Enter value for calibrate_epsilon (yes|no)      (current: no):
Error: 1.1079233e-01  -  Cd0=0.1500  Vg=0.0650
Error: 1.0586534e-01  -  Cd0=0.1575  Vg=0.0650
:
:
Error: 1.7395719e-03  -  Cd0=0.3100  Vg=0.0678
Error: 1.7395722e-03  -  Cd0=0.3100  Vg=0.0678
Error: 1.7395718e-03  -  Cd0=0.3099  Vg=0.0678


Drag coefficient Cd    : 0.309885 (-)
Glider volume Vg       : 0.067768 (m^3)
Glider compressibility : 5.000000 (*e-10)
Glider density         : 1025.555502 (kg/m^3)
Weight change          : 30.122814 (g)

Estimated pitch relationship:
tan(pitch) = T1 * buoyancy(m^3) + T2 * battpos(m) + T3 (tan(pitch0)) + T4 P (kbar)
T1 : 1784.9588582927518
T2 : -23.879872610869665
T3 : 0.09352561010068428
T4 : 0
Press enter to exit
```

A table with configurable parameters is shown below:

Parameter (unit)	Description
target_density (kg/m^3)	target density
mg (kg)	(measured) mass of glider
Vg (m^3)	estimate of glider volume
minlimitdepth (m)	minimum depth allowed in optimisation routine
maxlimitdepth (m)	maximum depth allowed in optimisation routine
cond_a (m/S)	scaling factor for correcting conductivity
cond_b (-)	offset for correcting conductivity
buoyancy_engine (shallow|deep)	sets buoyancy engine used, deep or shallow
latitude (decimal deg)	latitude of experiment (used for density)
longitude (decimal deg)	longitude of experiment (idem)
calibrate_epsilon (yes|no)	whether or not to calibrate for compressibility

The user is presented with default values and has the option to change the value or simply press enter, which retains the defailt value. After entering the last configuration parameter, the utility optimises for the parasite drag coefficient Cd0, the glider volume Vg, and, if calibrate_epsilon is set to "yes", the compressibility.

After the optimised values are found, they are displayed. Among the results returned are the glider density and the required weight change to match the glider's density to the target density.

The results are also shown graphically. The left panel shows the vertical profiles of water velocity (raw and filtered), the glider vertical velocity and the glider speed through water. The right panel shows the in-situ density profile, the target density, and the actual glider density (accounting for the compressibility). The dashed lines refer to the glider's density with increments of 50 g of weight change.

4.3 Estimated pitch relationship

The pitch the glider assumes during diving and climbing depends on the total torque exerted on the glider. Components influening the torque balance are the mass, the so-called h-moment (distance between the centres of buoyancy and gravity), the buoyancy drive, pitch battery position and pressure. Based on a linear regression model the contributions to the pitch by the buoyancy change, battery position, and pressure are estimated. The intended purpose is for glider flight simulations to use the correct pitch, when pitch is not set directly, but through a fixed battery position, for example.

The relationship for the pitch is given by

$$\tan(\phi) = T_1 \cdot V_b + T_2 \cdot b_p + T_3 + T_4 \cdot P,$$

where

V_b is the buoyancy change in m^3

b_p is the battery position in m, and

P is the pressure in kbar.

CHAPTER 5

gliderflight

5.1 gliderflight package

5.1.1 Submodules

5.1.2 gliderflight.gliderflight module

class gliderflight.gliderflight.**Calibrate**(*xtol=0.0001*)

Bases: object

Generic class providing the calibration machinery for steady-state and dynamic flight models

Basic steps are to

- set input data using the set_inputdata() method,
- mask data that should not take part in the minimisation routine (data near the dive apices, at the start of the dive, pycnoclines, etc.
- run the calibration.

Logical operators are used to mask data:

OR, AND, and NAND are implemented, and work on the *mask*

Parameters **xtol** (*float*) – tolerance in error in the parameters to be minimised.

AND(*mask*)

Logical AND

The new mask is the intersection (AND) of the existing mask and supplied mask

Parameters **mask** (*array of bool or bool*) –

NAND(*mask*)

Logical NAND

The new mask is the inverted intersection of the existing mask and the supplied mask. :param mask: :type mask: array of bool or bool

OR (*mask*)
Logical OR

The new mask is the union (OR) of the existing mask and the supplied mask.

Parameters **mask** (*array of bool or bool*) –

calibrate (**p*, *constraints='dhdt'*, *)*, *weights=None*, *verbose=False*)
Calibrate model

Given one or more model coefficients and specifications of measurements to use, this method calibrates the model, using the self.cost_funtion() method. The interface is flexible so any parameter that is used in the model description can be optimised for. Also the velocity component or combination of components can be set.

Parameters

- **p** (*variable length parameters*) – variable length of parameter names to be optimised.
- **constraints** (*list/tuple of str*) – names of measured velocities against which glider flight is evaluated. These must be present in the dictionary supplied by the set_input_data() method.
- **weights** (*None or array-like*) – weights. If more than one constraint is provided, weights sets their relative importance.
- **verbose** (*bool*) – prints intermediate results during optimising

Returns **rv** – the result of the optimisation routine

Return type dict

Examples

```
>>> # calibrating for mass and drag coefficient (implicitly using depth-rate)␣
↪and printing
>>> # intermediate results (mainly for debugging/progress monitoring)
>>> results = gm.calibrate("mg", "Cd0", verbose=True)
>>> print(results)
    {'mg': 70.00131, 'Cd0':0.145343}
>>>
>>> # Also calibrating the lift coefficient using measured incident water␣
↪velocity
>>> results = gm.calibrate("mg", "Cd0", "ah", constraints=('dhdt', 'U_relative
↪'), weights=(0.5, 0.5), verbose = True)
>>> print(results)
    {'mg': 70.00131, 'Cd0':0.145343, 'ah':3.78787}
```

Notes

The default measurement to evaluate the model against is the depth rate dhdt. If not specified when setting the input data using the set_input_data() method, it is computed automatically. Other velocity components that are to be used to calibrate the model have to be set specifically.

cost_function(*x*, *parameters*, *constraints*, *weights*, *verbose*)
Cost-function used to optimise parameters

This method first sets the parameters which are to be optimised for, and then computes the glider flight. A "cost" is computed from relatively weighted constraints.

Parameters

- **x** (*array*) – values of the parameters to be varied
- **parameters** (*list of str*) – parameter names
- **constraints** (*tuple or list of str*) – names of measured velocities against which glider flight is evaluated. These must be present in the dictionary supplied by the set_input_data() method.
- **weights** (*None or array-like of float*) – weights of constraints. If more than one constraint is provided, weights sets their relative importance.
- **verbose** (*bool*) – print intermediate results during optimising if set True

Returns **mse** – RMS value of exposed measurements (not masked)

Return type float

Different constraints can be applied, and if more than one, their relative contribution is set with weights.

Valid options:

dhdt : the error is computed from the difference between modelled w and observed dhdt w_relative : the error is computed from the difference between modelled w and w_relative (set separately to data dictionary) u_relative : the error is computed from the difference between modelled u and u_relative (set separately to data dictionary) U_relative : the error is computed from the difference between modelled U and U_relative (set separately to data dictionary)

depth : (experimental) the error is computed from the modelled and observed glider depth.

set_input_data(*time*, *pressure*, *pitch*, *buoyancy_change*, *density*, *dhdt=None*, *u_relative=None*, *w_relative=None*, *U_relative=None*, ***kwds*)
Sets the input data time pressure pitch buoyancy_change in-situ density and optionally u_relative and w_relative

Parameters

- **time** (*array*) – time (s)
- **pressure** (*array*) – pressure (Pa)
- **pitch** (*array*) – pitch (rad)
- **buoyancy_change** (*array*) – buoyancy change reported by the glider (cc)
- **ensity** (*array*) – in-situ density (kg m${-3}$)
- **dhdt** (*array, optional*) – depth rate m s^{-1} (if not given it is computed from pressure)
- **u_relative** (*array, optional*) – measured horizontal speed m s^{-1}
- **w_relative** (*array, optional*) – measured vertical speed m s^{-1}
- **U_relative** (*array*) – measured speed through water m s^{-1}

Notes

A mask is automatically created (including all data) when this method is called.

set_mask(*mask*)
Set a mask

Masks those data that should not be used to calibrate.

Parameters **mask** (*array of bool or bool*) –

Notes

If already set ones (after set_input_data(), then mask can be True or False to set all elements in mask.

class gliderflight.gliderflight.**Diagnostics**(*t*, *rho*, *U*, *FB*, *FD*, *FL*)
Bases: tuple

FB
Alias for field number 3

FD
Alias for field number 4

FL
Alias for field number 5

U
Alias for field number 2

rho
Alias for field number 1

t
Alias for field number 0

class gliderflight.gliderflight.**DynamicCalibrate**(*rho0=None*, *k1=0.02*, *k2=0.92*, *dt=None*, *alpha_linear=90*, *alpha_stall=90*, *max_depth_considered_surface=0.5*, *max_CPUs=None*)
Bases: *gliderflight.gliderflight.DynamicGliderModel*, *gliderflight.gliderflight.Calibrate*

Dynamic glider flight model, with calibration interface

class gliderflight.gliderflight.**DynamicGliderModel**(*dt=None*, *rho0=None*, *k1=0.2*, *k2=0.92*, *alpha_linear=90*, *alpha_stall=90*, *max_depth_considered_surface=0.5*, *max_CPUs=None*)
Bases: *gliderflight.gliderflight.ModelParameters*, *gliderflight.gliderflight.GliderModel*

Dynamic glider model implementation

This class inherits from ModelParameters and GliderModel. The physcis are provided by GliderModel. Interacting with ModelParameters is done through methods provided by ModelParameters.

Parameters

- **dt** (*float or None*) – time step (s)

- **rho0** (*float*) – background density (kg m^{-3})
- **k1** (*float*) – added mass fraction in longitudinal direction
- **k2** (*float*) – added mass fraction perpendicular to longitudinal direction
- **alpha_linear** (*float*) – angle (rad) up to which the parameterisation is considered linear
- **alpha_stall** (*float*) – angle (rad) up to which no lift will be generated (stalling angle)
- **max_depth_considered_surface** (*float*) – depth as reported by the pressure sensor which is considered the surface (u=w=0)
- **max_CPUs** (*int*) – maximum number of CPUs to use (clips at system availabel CPUs)

The only method provided by this class that is of interest to the user is solve(). The input to solve is a dictionary with time, pressure, pitch, buoyancy change density.

Methods inherited from ModelParameters can be used to define/set model coefficients, and to copy settings from another model instance.

After solving the model results are available as properties (t, U, wg, w, alpha)

The dynamic model solves the force balances including the intertial forces by numerical integration using a Runge-Kutta scheme. The inertial terms include the added mass terms. The relevant parameters can be set when creating an instance of this class.

Added mass

Added mass terms are specified by the coefficients k1 and k2, which refer to the added mass terms along the principle glider axis (k1) and vertically perpendicular (k2), where k1 and k2 are given as fraction of the glider mass mg.

Examples

>>>dm = DynamicGliderModel(rho0=1024, k1=0.2, k2=0.92, mg=70) >>>dm.define(mg=70) >>>dm.define(Vg=68, Cd0=0.15) >>>dm.solve(dict(time=tctd, pressure=P, pitch=pitch, buoyancy_change=buoyancy_drive, density=density)) >>>print(dm.U)

RK4 (*h*, *M*, *FBg*, *pitch*, *rho*, *at_surface*, *Cd0*, *u*, *w*)

Runge-Kutta integration method

Implementation to solve the model using the classic Runge-Kutta integration method.

Parameters

- **h** (*float*) – time step (s)
- **M** (*matrix (2x)*) – mass (and added mass matrix, inverted)
- **FBg** (*array*) – nett buoyancy force
- **pitch** (*array*) – pitch as recored by glider (rad)
- **rho** (*array*) – in-situ density (kg m^{-3})
- **at_surface** (*array of bool*) – condition whether or not at the surface
- **u** (*array*) – horizontal glider velocity (m s^{-1})
- **w** (*array*) – vertical glider velocity (m s^{-1})
- **Cd0** (*array*) – Lift coefficient per time step

Notes

The results are not returned as such. The parameters u and w are updated in place.

assemble_results(*results*, *intervals*)

compute_inverted_mass_matrix(*pitch*)
Computes the inverse of the mass matrix

not to be called directly

integrate(*data*)
integrate system

not to be called directly

process_fun(*interval*, ***arg_funs*)

solve(*data=None*)
Solve the model

Solves the flight model.

Parameters **data** (*dict or None*) – environment data (see Notes)

Returns **modelresult** – model result (named tuple with arrays of computed results)

Return type *Modelresult*

Notes

The data supplied should contain at least time, pressure, pitch, buoyancy_change and density, as reported by the glider. Depth rate (dhdt) will be added if not already present. Other data are ignored.

The intergration of the model maps the results on the time vector. For this to work successfully it is essential that there are no time duplicates or time reversals. The latter can occur when the system clock is updated with GPS time.

Use the methods `remove_duplicate_time_entries()` and `ensure_monotonicity()`.

Examples

```
>>> dm = DynamciGliderModel(dt=1, k1=0.2, k2=0.98, rho0=1024)
>>> dm.define(mg=70, Vg=68)
>>> data = dict(time=time, pressure=P, pitch=pitch, buoyancy_change=vb,␣
↪density=rho)
>>> dm.solve(data)
>>> plot(dm.U)
```

stall_factor(*alpha*)

class gliderflight.gliderflight.**GliderModel**(*rho0=None*)
Bases: object

Common glider model class

This class, meant to be subclassed, implements the physical glider model description

G = 9.81

RHO0 = 1024

U
incident water velocity (m s^{-1})

alpha
angle of attack (rad)

compute_FB_and_Fg(*pressure*, *rho*, *Vbp*, *mg=None*, *Vg=None*)
Computes the vertical forces FB and Fg

Parameters

- **pressure** (*array-like or float*) – pressure (Pa)
- **rho** (*array-like or float*) – in-situ density (kg m^{-3})
- **Vbp** (*array-like or float*) – volume of buoyancy change (m^{-3})
- **mg** (*array-like, float or None*) – mass of glider (kg). If None (default), then self.mg is used for the computation
- **Vg** (*array-like, float or None*) – Volume of glider (m^{3}). If None (default), then self.Vg is used for the computation

Returns

- **FB** (*Buoyancy force*) – array-like or float
- **Fg** (*Gravity force*) – float

compute_dhdt(*time*, *pressure*)
Compute the depth rate from the pressure

Parameters

- **time** (*array-like*) – time in s
- **pressure** (*array-like*) – pressure (Pa)

Returns depth-rate (m/s)

Return type array-like

Notes

The density used to convert pressure into depth is given by self.RHO0

compute_lift_and_drag(*alpha*, *U*, *rho*, *Cd0=None*)
Compute lift and drag forces

Computes lift and drag forces using parameterised functions

Parameters

- **alpha** (*array-like or float*) – angle of attack
- **U** (*array-like or float*) – incident water velocity
- **rho** (*array-like or float*) – in-situ density
- **Cd0** (*array-like, float or None*) – parasite drag coefficient (-). If None (default), then self.Cd0 is used for the computation

Returns

- **q** (*array-like or float*) – dynamic pressure (Pa)
- **L** (*array-like or float*) – lift force (Pa)

- **D** (*array-like or float*) – drag force (Pa)

convert_pressure_Vbp_to_SI(*m_water_pressure*, *m_de_oil_vol*)
converts units of glider sensor data into SI data

Parameters

- **m_water_pressure** (*array-like or float*) – water pressure in bar
- **m_de_oil_vol** (*array-like or float*) – buoyancy change reported by glider in cc

Returns

- **pressure** (*array-like or float*) – pressure (Pa)
- **Vbp** (*array-like or float*) – volume of bulyancy change (m^3)

ensure_monotonicity(*data*, *T_search_span=600*)
Ensure monotonicity of the data series.

Parameters

- **data** (*dict*) – dictionary with environment data.
- **T_search_span** (*float (600)*) – time span to search back in time for time gaps

Returns dictionary with updated environment data.

Return type dict

Notes

Some times the glider clock gets corrected when it deviates too much from the GPS time. This happens of course at the surface. It can be that time is stepped backwards, which means that the timestamps are not monotonic any more. The strategy we adopt here is, because it happens at the surface, we look if there is a time gap (due to data transmission for example) in the interval 10 minutes prior the time shift. Then we simply move this section of time backwards as well. If this is not possible, we undo the time correction and move all timeseries forward in time.

pitch
pitch angle (rad)

remove_duplicate_time_entries(*data*)

stall_factor(*alpha*, ***kwds*)

t
time (s)

w
vertical water velocity (m s^{-1})

wg
vertical velocity of glider relative to surface (m s^{-1})

exception gliderflight.gliderflight.**ModelParameterError**
Bases: BaseException

class gliderflight.gliderflight.**ModelParameters**(*parameterised_parameters_dict*)
Bases: object

Configuration class for glider model parameters

This class defines the configuration parameters of the glider model. The class is meant to be subclassed from model implementations.

Parameters **parameterised_parameters_dict** (*dict*) – dictionary of parameters that should be computed rather then being set explicitly

Methods defined in this class:

- define(): define or set a parameter
- show_settings(): prints the current settings
- copy_settings(): updates model parameter settings from another model
- undefined_parameters(): returns which parameters have not been set yet.
- cd1_estimate(): estimates the induced drag coefficient
- aw_estimate: estimates the lift coefficient due to the wings using a parameterisation

awEstimate()
Parameterisation for aw

Computes aw using a parameterisation

Returns **aw_param** – parameterised value of aw

Return type float

Notes

If aw is set by using define, the set value takes precedence.

cd1Estimate()
Parameterisation for Cd1

Computes Cd1 using a parameterisation.

Returns **Cd1_param** – parameterised value of Cd1

Return type float

Notes

If Cd1 is set by using define, the set value takes precedence.

copy_settings(*other*)
Copy model parameters

Copy model parameters from a different instance of ModelParameters and apply it to to self.

Parameters **other** (ModelParameters) – an other instance of this class (or subclassing this class)

Examples

```
>>> dynamic_model.copy_settings(steady_state_model)
```

define(**kw*)
Define (set) one or more glider configuration parameters.

Parameters **kw** (*dict*) – keywords with parameter name and values

Examples

```
>>> glidermodel.define(Cd0=0.24)
>>> glidermodel.define(Vg=50e-3, mg=60)
```

get_settings()
Get model settings

Return a dictionary with model coefficient settings

Returns **settings** – a dictionary with the current parameter setting

Return type dict

has_aoa_parameter_changed()
test whether any of the parameters that appear in the angle of attack estimate have changed

Returns **rv** – test result

Return type bool

show_settings()
Prints model parameters

undefined_parameters()
Returns undefined parameters

Checks all model coefficients for having been set. All coeficients set to None are returned.

Returns **list** – list of undefined parametrs

Return type list-comprehension

class gliderflight.gliderflight.**Modelresult**(*t*, *u*, *w*, *U*, *alpha*, *pitch*, *ww*, *depth*)
Bases: tuple

U
Alias for field number 3

alpha
Alias for field number 4

depth
Alias for field number 7

pitch
Alias for field number 5

t
Alias for field number 0

u
Alias for field number 1

w
Alias for field number 2

ww
Alias for field number 6

class gliderflight.gliderflight.**SteadyStateCalibrate**(*rho0=None*)
Bases: *gliderflight.gliderflight.SteadyStateGliderModel*, *gliderflight.gliderflight.Calibrate*

Steady-state glider flight model, with calibration interface

class gliderflight.gliderflight.**SteadyStateGliderModel**(*rho0=None*)

Bases: *gliderflight.gliderflight.ModelParameters*, *gliderflight.gliderflight.GliderModel*

Steady-state implementation

This class inherits from ModelParameters and GliderModel. The physcis are provided by GliderModel. Interacting with ModelParameters is done through methods provided by ModelParameters.

Parameters **rho0** (*float*) – background in-situ density

The only method provided by this class that is of interest to the user is solve(). The input to solve is a dictionary with time, pressure, pitch, buoyancy change density.

Methods inherited from ModelParameters can be used to define/set model coefficients, and to copy settings from another model instance.

After solving the model results are available as properties (t, U, wg, w, alpha)

Examples

```
>>> gm = SteadyStateGliderModel(rho0=1024)
>>> gm.define(mg=70)
>>> gm.define(Vg=68, Cd0=0.15)
>>> gm.solve(dict(time=tctd, pressure=P, pitch=pitch, buoyancy_change=buoyancy_
↪drive, density=density))
>>> print(gm.U)
```

model_fun(*x*, *m_pitch*, *Cd0*)

implicit function of the angle of attack

Parameters

- **m_pitch** (*float or array of floats*) – measured pitch
- **Cd0** (*float*) –
- **is a parameter that might change during a mission, so self. Cd0 is set as an** (*Cd0*) –
- **this function needs to take Cd0 as a float at the appropriate time. It is** (*array,*) –
- **responsibility of the caller function to pass the value of Cd0.** (*the*) –

reset()

Resets angle of attack interpolation function

solve(*data=None*)

Solve the model

Solves the flight model.

Parameters **data** (*dict*) – environment data (see Notes)

Returns **modelresult** – model result (named tuple with arrays of computed results)

Return type *Modelresult*

Notes

The data supplied should contain at least time, pressure, pitch, buoyancy_change and density, as reported by the glider. Depth rate (dhdt) will be added if not already present. Other data are ignored.

Examples

```
>>> gm = SteadyStateGliderModel()
>>> gm.define(mg=70, Vg=68)
>>> data = dict(time=time, pressure=P, pitch=pitch, buoyancy_change=vb,␣
↪density=rho)
>>> gm.solve(data)
>>> plot(gm.U)
```

solve_for_angle_of_attack(*pitch*)

Solves for the angle of attack

Solves angle of attack using an interative method.

Parameters **pitch** (*array-like or float*) – pitch (rad)

Returns **aoa** – angle of attack (rad)

Return type array-like or float

Notes

This method uses an interpolating function. If any parameter on which this calculation depends, changes, the interpolating function is recomputed. Whether any of these parameters is changed, is tracked by the ModelParameters.define() method.

solve_model(*rho*, *FB*, *pitch*, *Fg*)

Solves first for angle of attack and then incident velocity

Not intended to be called directly.

5.1.3 gliderflight.glidertrim module

5.1.4 Module contents

CHAPTER 6

Contact

If you have any suggestions, remarks or feature wishes, you can contact me via

email: <lucas.merckelbach@hzg.de>

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Bibliography

[merckelbach2019] Merckelbach, L., A. Berger, G. Krahmann, M. Dengler, and J. Carpenter, 2019: A dynamic flight model for Slocum gliders and implications for turbulence microstructure measurements. J. Atmos. Oceanic Technol. 36(2), 281-296, doi:10.1175/JTECH-D-18-0168.1

Python Module Index

g

Index

M

N

O

P

R

S

T

U

W