

FluxEngine v4.0.3: Changes and User Guide

May 8, 2022

Contents

1	Overview and major changes	2
1.1	Execution time	2
1.2	Feedback and bug reporting	3
2	Downloading and installing FluxEngine	3
3	Verifying your installation	4
4	Interactive tutorials	4
5	Running FluxEngine v4.0	5
5.1	Using the command line tool	5
5.2	Driving FluxEngine from your own Python scripts	6
6	Input data requirements	7
7	Creating and modifying configuration files	9
7.1	Opening configuration files	9
7.2	Configuration file parameters	9
7.3	k parameterisation specific variables	13
7.4	Tokens in file paths	14
7.5	Data layers in detail	14
7.5.1	Using tokens to reuse selected data	16
7.5.2	Optional data layer attributes	16
7.5.3	Data layer preprocessing	17
7.6	Working with different temporal resolutions	17
7.7	Indexing input data using the temporal dimension	18
7.8	Setting output filenames and directory structure	18
7.9	Grouping output from multiple time points into a single file	19
7.10	Filtering input data using a mask	19

8	Bundled tools	20
8.1	Calculating net fluxes with <code>fe_calc_budgets.py</code>	21
8.2	Using <code>fe_reanalyse_fco2_driver.py</code> to reanalyse data to a consistent temperature and depth	21
8.3	Using <code>fe_text2ncdf.py</code> to create FluxEngine compatible NetCDF files .	22
8.4	Automatically updating old configuration files	23
9	Guide for developers	23
9.1	Adding pre-processing functions	23
9.2	Adding k-parametrisation functors	24
9.2.1	Adding configuration variables	25
9.3	Contributing	26

1 Overview and major changes

The major changes, new to FluxEngine version 4.0 include:

- *Python version* - With support for Python 2.x stopping in January, all FluxEngine code and tools have been updated to run using Python 3.6+.
- *Installation as a Python package* - FluxEngine v4.0 is provided as a Python package which can be installed from PyPi using Pip. This considerably simplified the installation process for Windows, MacOS and Linux, and full supports the use of virtual environments.
- *Command line tools* - FluxEngine comes with a number of command line tools. These have been updated to separate importable library code (which can be used in custom Python scripts) from the driver scripts which are used to run the tools via the command line. The driver scripts are now automatically added to your operating system's environment path, mean you don't need to `cd` into the FluxEngine tools directory to use them.
- *Updated interactive tutorials* - Four interactive Jupyter notebook tutorials were added after the release of FluxEngine v3.0. These have been updated for v4.0. These can be found in the `tutorials` subdirectory.
- *Simplifications to config files* - Configuration files have been further simplified to remove some unnecessary options. In some cases these options were no longer needed, in other cases they could be inferred from the data provided - reducing the potential for user error when creating configuration files.

1.1 Execution time

Simple benchmarking was conducted using an Intel Core i5 2.7GHz processor with 8GB RAM running MacOS El Capitan. A one year (2010) run using the SOCATv4 verification configuration (Nightingale 2000 k parameterisation with process indicator layers off) took

approximately 6 minutes to complete using FluxEngine v3.0. This is compared to over 9 minutes for the same configuration using FluxEngine v2.0. This speed-up can be largely attributed to the removal of non-required input data layers. Run time will therefore differ depending on the options specified by configuration files. For example the Takahashi 2009 verification run took just 4 minutes to complete.

1.2 Feedback and bug reporting

Feedback is greatly appreciated both in terms comments about which aspects of running and using FluxEngine were not intuitive or poorly explained, as well as which features would be useful to you in future releases. These can be e-mailed to Tom Holding at t.m.holding@exeter.ac.uk. Bugs can be reported via our GitHub page by opening an issue: <https://github.com/oceanflux-ghg/FluxEngine/issues> or by e-mailing Tom.

2 Downloading and installing FluxEngine

You will need Python 3.6 (or newer) installed before you can install or use FluxEngine. If you don't already have Python installed we recommend installing Anacondas for this because it has excellent support for Windows, MacOS and Linux. Anacondas is specifically designed to be used by scientists and engineers, and comes with useful package and environment management tools. You can download Anacondas from <https://www.anaconda.com/distribution/> (make sure you choose the version for Python 3.6 or newer).

It is also highly recommended that you create a separate virtual environment for FluxEngine. This prevents dependencies from different projects or versions of Python from interfering with one another. For information on how to create and manage virtual environments with Anacondas, see <https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>.

Once you have Python installed (whether via Anacondas or not) you can install FluxEngine in two ways. For most people, the easiest way is to use Pip - a command line tool for downloading and installing Python modules. The latest stable version of FluxEngine is packaged as a Python module, and can be installed from PyPi with the following command:

```
pip install FluxEngine
```

The very latest version, which may not be as stable as the PyPi version, can be downloaded from our GitHub repository <https://github.com/oceanflux-ghg/FluxEngine/archive/master.zip>, or if you're a Git user you can clone the repository here: <https://github.com/oceanflux-ghg/FluxEngine.git>. The PyPi version will be updated periodically as features are added and tested, or bugs patched. If you download FluxEngine from GitHub you'll need to build the package and install it from your local file system, e.g. by running:

```
python setup.py sdist bdist_wheel
```

```
pip install dist/FluxEngine-4.0.dev0.tar.gz
```

3 Verifying your installation

Two verification scripts come bundled with FluxEngine. These make it easy to verify that everything has installed correctly and that FluxEngine is producing correct output. These scripts compare output generated by your local copy of the FluxEngine with a known reference, and will report any discrepancies.

To verify using Takahashi2009 (T09) and/or SOCATv4 data run the following command/s using command line (aka Terminal or Command Prompt). Make sure you have activated the correct virtual environment if you installed FluxEngine using an environment manager. The executable scripts `fe_verify_takahashi09.py` and `fe_verify_socatv4.py` will have been automatically added to your environment path so there is no need to change directory. On Windows you may need to exit and re-open command prompt before running these commands. The output will be stored in a `verification_output` subdirectory in your current working directory.

```
fe_verify_takahashi09.py
fe_verify_socatv4.py
```

Verification will take 10-20 minutes and if your installation has been successful you will receive a messages saying that the verification is completed successfully after each command. Note that on Windows, if you have .py files associated with a text editor instead of the Python interpreter, calling one of the FluxEngine command line tools will open the script file instead of running it. To fix this, you can either tell Windows to associate .py files with the Python interpreter (see <https://docs.python.org/3/faq/windows.html#how-do-i-make-python-scripts-executable> for details). An alternative work-around is to run the command line tools by explicitly calling the Python interpreter with python prefix added to the command - although you will need to either provide the full file path to the script, or cd (change directory) into the directory which contains the command line tools.

4 Interactive tutorials

FluxEngine comes with interactive Jupyter notebook tutorials which demonstrate the basics of how to setup and run FluxEngine for some simple scenarios. These run in a web browser, and can be started by running `fe_tutorials.py` from the command line (make sure you're in the correct virtual environment first). This command starts a Jupyter server and automatically opens the Jupyter hub page for the tutorials in a web browser. In the web browser you'll the contents of the tutorial's directory, with a sub-directory for each tutorial. Within each tutorial sub-directory there is a Jupyter notebook file (ending in .ipynb). Clicking the Jupyter notebook file will open the interactive tutorial.

Note: On some computers, the web browser may not automatically open. If this happens you can copy and paste the link provided in the command prompt / terminal window into a web browser. The tutorials cover:

- *Tutorial 1* - Introductory topics, including: using Jupyter notebooks, verification, modifying FluxEngine configuration files, adding input data layers, running FluxEngine and plotting output.
- *Tutorial 2* - Working with in situ data, including: using build in tools to re-analyse in situ fCO₂ data to a consistent temperature and depth, using the `fe_text2ncf.py` tool to converting text formatted data into netCDF files for use with FluxEngine, creating/modifying configuration files to use new data, using the `fe_append2insitu.py` tool to append FluxEngine output to the original in situ files.
- *Tutorial 3* - Working with fixed station data, including: a more detailed look at `fe_reanalyse_fco2_driver.py`, utilising input data with a temporal dimension, configuring FluxEngine to perform unit conversions with a pre-processing function, outputting CO₂ flux time series.
- *Tutorial 4* - NO₂ gas fluxes, including: preparing and input data to calculate atmosphere-to-ocean NO₂ fluxes, using pre-processing functions to perform unit conversions and estimate missing parameters, overriding dimension names for individual input data layers, and visualising the effect of surfactant suppression on gas fluxes.

5 Running FluxEngine v4.0

There are two ways to run FluxEngine: using the command line tools or by importing FluxEngine as a module in a custom Python script. The simplest method is to use the command line tools as these provide enough flexibility for most use-cases.

5.1 Using the command line tool

The command line tool for running FluxEngine is called `fe_run.py` and is automatically added to your environment path, so you don't need to be in a particular directory to access it (but you will have to activate the correct virtual environment, if you're using one). To run FluxEngine using this tool, open a terminal window (or Command Prompt) and type `fe_run.py` followed by the path to a configuration file, then any other options:

```
fe_run.py config [-options]
```

where `[-options]` is an option list of options to change how FluxEngine runs. Information on valid options can be listed using `fe_run.py -h` to view the help information. The `-h` flag can be used with any of the command line tools to view the help information.

Some options will require you to specify a value, for example when defining start and stop dates, whereas others will just turn a specific feature on/off.

```
fe_run.py configs/example_config.conf -l -start_date 2000 -end_date 2010
```

The above command will run FluxEngine using a configuration file called `example_config.conf` located in the `configs` subdirectory of the current working directory. All file paths in FluxEngine can be supplied as absolute paths or relative to the current working directory. Configuration files are used to define the flux calculation in detail, including the flux equation to use, input data, gas transfer parameterisation and output file structure. Next, the command uses three options: The `-l` option tells FluxEngine to run without process indicator layers (described later). Turning these off reduces the time taken to run and results in smaller output files. The second and third options, `-start_date` and `-end_date`, tell FluxEngine to calculate fluxes from the year 2000 to the year 2010. More information on using `fe_run.py` can be found in the Jupyter tutorials.

Note: A useful option for testing is `-S1` which will only run FluxEngine for the first time step. This allows you to check the output, and for any error messages, before committing to a longer run.

5.2 Driving FluxEngine from your own Python scripts

Some tools are provided which allow you to write Python scripts to run FluxEngine in custom ways. To do this you should import the `fluxengine` package using `import fluxengine`. The `fluxengine.core.fe_setup_tools` module contains a function called `run_fluxengine`, which allows you to specify custom configuration objects to drive FluxEngine. This module also contains functions for parsing, verifying and modifying configuration files. While these tools are not currently well documented, example use can be found in the `fe_run.py` command line tool, and they allow much more control over how FluxEngine runs. For example it is possible to use a single configuration file and overwrite a subset of parameters to run a suite of similar simulations. This also allows FluxEngine to be incorporated into other projects, for example as one step in a larger model or workflow.

The general workflow for initialising and running FluxEngine using these tools involves the following steps:

1. Parse the config file
2. Verify the config file)
3. Specify one or more time point to run the FluxEngine
4. Create a set of run parameters (these are derived from the verified config file but are specific to the date/time you are using)
5. Run FluxEngine for a specific date/time

6. Check return code

Steps 4-6 will be repeated for each time point you want to run the FluxEngine for. Various functions are available in `fluxengine.core.fe_setup_tools` to help with each of these steps. For an example of how they are used to achieve each of these steps see the `run_fluxengine` function in the same file.

Note that these tools are intended for use by users who are proficient in Python. Please report any bugs (see section 1.2).

6 Input data requirements

Input data are specified in the configuration file and are conceptualised as 'data layers' (see section 7.5 for details on how to specify data layers in the configuration file). The minimum required input data needed to run FluxEngine are:

- Sea surface temperature (sub-skin, skin or both).
- Atmospheric CO₂ (either as Molar fraction in dry air, partial pressure in dry air or concentration).
- CO₂ in the surface water (either partial pressure or concentration).
- Air pressure at sea level.
- Sea surface salinity.
- Any data required by your chosen gas transfer velocity (k) parameterisation (typically wind speed and the second/third moment of wind speed).

There are various other optional inputs, some of which are required only when using specific functionality. Table 1 lists the data layers which are recognised by FluxEngine, whether they are required or optional, and their expected units. Additional input data layers can be added by simply specifying them in the configuration file in the same way you would define any other input data layer (see section 7.5). Doing this will automatically make them available to any custom FluxEngine code. This is useful when additional data is used by custom gas transfer parameters or pre-processing functions.

name	description	units	required?
sstskin	sea surface skin temperature	$^{\circ}K$	if sstfnd not supplied
sstfnd	sea surface foundation temperature	$^{\circ}K$	if sstskin not supplied
pco2_sst	Ocean temperature at the point of CO ₂ measurement. Only used for calculations of CO ₂ flux	$^{\circ}C$	optional
vgas_air	Molar fraction of the flux gas (e.g. CO ₂) in dry air.	$\mu mol\ mol^{-1}$ or <i>ppm</i>	Required if pgas_air and partial pressure (pgas_air) are not supplied.
pgas_air	partial pressure of the flux gas (e.g. CO ₂) in dry air	μatm	Only used in Takahashi verification. gas_air not supplied. Not required if concentration or molar fraction (vgas_air) data are supplied.
pgas_sw	partial pressure of the flux gas (e.g. CO ₂) (aqueous)	$\mu atm.$	Not required if concentration data are supplied.
conca	concentration of the flux gas (e.g. CO ₂) at the interface	$gm^{-3}.$	required if not using fugacity or partial pressure inputs
concw	concentration of the flux gas (e.g. CO ₂) (sub-skin)	$gm^{-3}.$	required if not using fugacity or partial pressure inputs
pressure	air pressure at sea level	<i>mbar</i>	always required
salinity	surface salinity	none	always required
ice	fraction ice coverage	none	optional
pressure	air pressure at sea level	<i>mbar</i>	always required
windu10	wind speed	ms^{-1}	by most gas transfer velocity parameterisations
windu10_moment2	wind speed second moment		by most gas transfer velocity parameterisations
windu10_moment3	wind speed third moment		by some gas transfer velocity parameterisations
sigma0	radar backscatter	<i>dB</i>	by some gas transfer velocity parameterisations
sig_wv_ht	significant wave height	<i>m</i>	by some gas transfer velocity parameterisations
rain	precipitation	$mm\ day^{-1}$	when including precipitation effects (e.g. rain_wet_deposition or bias_sstskin_due_rain)
biology	chlorophyll-a concentration		when using biology process indicator layer
sstgrad	sea surface temperature gradient	$^{\circ}K\ m^{-1}$	when using sst gradients process indicator layer
mask	multipurpose mask used to select specific region/s to compute gas fluxes for	n/a	not required
atlantic_ocean_mask	Atlantic Ocean mask	none	when using Atlantic Ocean region indicator layer
pacific_ocean_mask	Pacific Ocean mask	none	when using Pacific Ocean region indicator layer
southern_ocean_mask	Southern Ocean mask	none	when using Southern Ocean region indicator layer
indian_ocean_mask	Indian Ocean mask	none	when using Indian Ocean region indicator layer
longhurst_mask	Longhurst provinces mask	none	when using Longhurst

7 Creating and modifying configuration files

7.1 Opening configuration files

Configuration files are stored as plain text files with a `.conf` file extension. This means they can be opened in any text editor. If you're using Windows you may find that when you open a configuration file all of the information is stored on a single line, making it difficult to work with. To avoid this you should open the configuration file in a text editor which can understand Unix line endings (e.g. notepad). There is a free lightweight piece of software called *Notepad++* is perfect for this (<https://notepad-plus-plus.org/>). *Sublime Text* is an excellent feature rich but lightweight cross-platform text editor (<https://www.sublimetext.com/>).

The order in which parameters are specified in configuration files doesn't matter with the exception of the first line which must specify the version of FluxEngine that the configuration file is designed for. This takes the form `#?FluxEngineConfigVersion:version`, where `version` is replaced with the version number. For example, for version 4.0 you would use `#?FluxEngineConfigVersion:4.0`. We aim for back compatibility of configuration files, but this is not always possible and the change between version 3.x and 4.0 introduced some minor incompatibilities. A tool is provided to automatically make old configuration files compatible with newer versions of FluxEngine, see section 8.4.

Example annotated configuration files can be found in the `configs` directory of your FluxEngine installation path. Parameters are specified in the config file by name followed by an equals sign (=) and then the value. Note that in contrast to previous versions the equals sign is now required (as of version 3.0), but this means that parameter values can now contain spaces. Both variable names and values are case sensitive, and comments can be added using the hash (#) character. Variables can be specified in any order although it is convenient to group related variables together. An example definition of two parameters is given below:

```
varname1 = 100.0
varname2 = test variable #trailing and preceeding whitespace is ignored
```

7.2 Configuration file parameters

A short summary of each parameter is given below.

flux_calc

Selects the flux equation to use. Valid options are:

- **bulk** - i.e. $F = k\alpha_w(pCO_{2W} - pCO_{2A})$, where F is the air-sea flux, k is the gas transfer velocity, α_w is the solubility of the gas in sea water, pCO_{2W} is the partial pressure of CO_2 in the surface sea water and pCO_{2A} is the partial pressure of CO_2 in the atmosphere.

- **rapid** - As described in Woolf et al. (2016) *Journal of Geophysical Research: Oceans*. $F = k(\alpha_w pCO_{2w} - \alpha_A pCO_{2A})$.
- **equilibrium** - As described in Woolf et al. (2016) *Journal of Geophysical Research: Oceans*.

If you choose to use supply atmospheric and ocean gas inputs as a concentration (rather than fugacity or partial pressure), the choice of **flux_calc** is superseded by the concentration data. For example, if you have supplied both **conca** (interface concentration) and **concw** (sub-skin concentration), and they have been calculated using the same solubility, then the **rapid** calculation is equivalent to the **bulk**. To avoid this **conca** should be calculated using solubility at the skin layer, while **concw** should be calculated using solubility from the sub-skin. For more information, see Woolf et al. (2016) *Journal of Geophysical Research: Oceans*.

temporal_resolution

This is an optional parameter which defines the temporal resolution for which the flux calculation is computed. If the parameter is not defined in the configuration file it defaults to monthly. Different temporal resolutions should be defined using the **hh:mm** format to specify the length of timesteps in days, hours and minutes respectively. A full explanation, with examples, is given in section ??.

sst_gradients, cool_skin_difference

Valid values for **sst_gradients** are **yes** or **no**. If one of **sstskin** or **sstfnd** input data layers are not specified in the configuration file, and **sst_gradients** is set, then the missing SST input is estimated using the following equation:

$$\text{sstskin} = \text{sstfnd} - \text{cool_skin_difference}$$

where **cool_skin_difference** is the difference in temperature between the foundation layer and the skin layer (in Kelvin). The default is 0.17K (see Donlon et al. 2002).

saline_skin_value

Saline skin value is added to salinity. It is an optional entry and will default to 0.0 if not specified.

axes_data_layer, latitude_prod longitude_prod time_prod

Specifies the data layer from which to extract the latitude, longitude and time data from. **axes_data_layer** must be the name of a data layer, e.g. **sstskin**, and all other input data layers will have their dimensions checked for consistency with the named data layer.

pco2_reference_year, pco2_annual_extrapolation

These are optional entries which can be used to specify a reference year from which pCO_2 / fCO_2 can be adjusted. This applies an annual correction according to

$$\text{pco2_increment} = (\text{year} - \text{pco2_reference_year}) * \text{pco2_annual_correction}$$

`datalayername_path, datalayername_prod`

These define each input data layer. A full explanation with worked examples is provided below (section 7.5).

`random_noise_windu10, random_noise_sstskin, random_noise_sstfnd, random_noise_pco2`

These are optional variables which control whether random noise is added for each of their respective data layers. Valid values are *yes* or *no*. Note that currently the magnitude of noise must be modified directly in the source code (this is not advised unless absolutely necessary).

`bias_datalayername, bias_datalayername_value`

Setting `bias_datalayername` will add a constant value (defined by `bias_datalayername_value`) to the named data layer. This is applied after any random noise is added. Currently the following data layer names are supported: `windu10`, `sstskin`, `sstfnd` and `pco2`. Valid options for `bias_datalayername_value` are *yes* or *no* and default to *no* if not supplied. `bias_datalayername_value` must be a valid numeric value and defaults to 0.0 if not supplied.

`bias_k, bias_k_percent, bias_k_value, bias_k_biology_value, bias_k_wind_value`

These variables control the bias applied to *k* (the gas transfer velocity). `bias_k` and `bias_k_percent` but be set to either *yes* or *no* and control whether any bias is applied at all and whether the bias is added as an absolute value or as a percentage of the original value, respectively. Default values for both of these variables are *no*. `bias_k_value` requires a numeric variable (the default is 0.0) and controls the magnitude of the bias (whether as a percentage or absolute value). Bias is only added to *k* if the value of `windu10` at the same point in space is above a threshold specified by `bias_k_wind_value` (the default value is 0.0). Similarly the corresponding value for biology is below `bias_k_biology_value` (the default is 0.0).

`k_parameterisation`

This controls the way that the gas transfer velocity (*k*) is calculated. FluxEngine comes bundled with a number of *k* 'functors' - self-contained Python classes which take data layers as input and write output to one or more data layers (typically the *k* datalayer). For a list of available parameterisations you can run the `fe_run.py` script with the `-list_parameterisations` option. Alternatively you can see the Python implementation of each *k* functor in `rate_parameterisation.py` file located in the `fluxengine.core` directory. These can be referred to by name in configuration files. User defined parameterisations can be added to this file and assigned to `k_parameterisation` by name in the config file. Before writing new *k* functors you should read the guide for developers (section 9), as this describes best practices and provides information on potential pitfalls.

`schmidt_parameterisation`

This sets the Schmidt number parameterisation to use. There are currently two options `schmidt_Wanninkhof1992` (the default) and `schmidt_Wanninkhof2014`. These are based on Wanninkhof's 1992 [?] and updated 2014 [?] Schmidt number parameterisations respectively.

`kb_asymmetry`

This is an optional parameter which controls the relative weighting given to the atmospheric concentration when calculating the bubble component (`kb`) of the gas transfer velocity. This allows the user to scale the relative importance of direct and bubble components when calculating total gas transfer velocity, such that larger values increase the importance of the bubble component. This is only used when `k_parameterisation` is set to `kt_OceanFluxGHG`. If it is not specified it defaults to 1.0 (no asymmetry).

`bias_sstskin_due_rain,` `bias_sstskin_due_rain_value,`
`bias_sstskin_due_rain_intensity,` `bias_sstskin_due_rain_wind`

These control whether and how rain (the `rain` data layer) influences sea surface temperature. `bias_sstskin_due_rain_value` turns this feature on or off (valid values are `yes` and `no`, with the default being `no`). If this feature is turned on, a constant bias (`bias_sstskin_due_rain_value`, default value of 0.0) will be added to sea surface temperature anywhere that `rain` intensity is greater than `bias_sstskin_due_rain_intensity` (default value 0.0) and wind intensity (the `windu10` data layer) is less than `bias_sstskin_due_rain_wind` (default is 0.0).

`rain_wet_deposition`

This option enables wet deposition. Valid values are `yes` or `no`, and the default value is `no`.

`k_rain_linear_ho1997`

This option enables a linear additive gas transfer velocity term due to rain. A description of the method can be found in Ashton *et al.* (2016). Valid values are `yes` or `no`, and the default value is `no`.

`k_rain_nonlinear_h2012`

This option enables a nonlinear additive gas transfer velocity term due to rain. A description of the method can be found in Ashton *et al.* (2016) and Harrison *et al.* (2012). Valid values are `yes` or `no`, and the default value is `no`.

`GAS`

This is an optional parameter which specifies the gas to calculate the air-sea flux of. Valid options are `co2`, `n2o` and `ch4`. If not defined then the default `co2` will be used.

`output_dir`

This specifies the root directory that will be used for writing output to. If the directory doesn't exist it will be created. Any subdirectories which are used to organise the output will be created in this folder.

`output_structure`

This is an optional parameter which defines the way output will be organised into subdirectories. Tokens can be used in this definition of `output_structure`. The default value is `<YYYY>/<MM>`, which will create a directory for each year that FluxEngine runs, and subdirectories for each month of each year. The resulting netCDF files will therefore be organised first by year, then by month. This is the default output directory structure because it is same as that required by the `ofluxghg_net_budget.py` tool (see the section 8) to make running this tool convenient.

`output_file`

This is an optional parameter which defines the names of the output netCDF files produced by FluxEngine. Tokens can be used in the definition (see section 7.4). If it is not defined the default value of `OceanFluxGHG-month<MM>-<mmm>-<YYYY>-v0` is used. If the file already exists it will be overwritten.

`output_temporal_chunking`

This is an optional parameter which tells FluxEngine to combine output into fewer files. The default value of 1 means that each output file will contain one timestep. Setting to, for example, 12 means that one output file will be created for every 12 time steps and FluxEngine output will be placed in a temporal dimension of the output netCDF files. See section 7.9 for examples. The time range covered by each file depends on the temporal resolution (see `temporal_resolution` or section 7.6).

Full meta data, including a list of data layer names recognised by FluxEngine, default values and expected data types can be accessed in the `settings.xml` file in the `fluxengine_src` directory. *Note: It is strongly recommended that you do not modify this file.*

7.3 k parameterisation specific variables

Several `k_parametrisation` options require additional variables to be specified which change the way that the gas transfer velocity is calculated. These must be defined in the configuration file. In particular `k_generic` requires a Schmidt number to be defined `k_generic_sc` (valid values are 600.0 and 660.0) as well as weightings for each order of the generic gas transfer velocity equation, i.e. `k_generic_a0`, `k_generic_a1`, `k_generic_a2` and `k_generic_a3`.

`kt_OceanFluxGHG`, `kt_OceanFluxGHG_kd_wind` and `k_Wanninkhof2013` require that `kb_weighting` and `kd_weighting` be specified. These define the weighting for the bubble and direct components of the gas transfer velocity, as described in Goddijn-Murphy et al., (2015).

Other custom or third-party k parametrizations may require other variables to be specified and you should consult any documentation or guidance specific to the parametrization being used, or examining the initialiser function (`__init__`) of the relevant parametrization functor the `rate_parameterisation.py` file.

7.4 Tokens in file paths

Configuration files need to define a number of file paths. These include the location of various input data layers, the root output directory (`output_dir`), output directory structure (`output_structure`) and output file names (`output_file`). These file paths will often need to change depending on the date, or even time, that the data corresponds to. FluxEngine uses several 'tokens' which allow time information to be substituted into file paths (or file names) to allow these to change depending on the point in time being analysed. Tokens are always prefixed by a less-than sign (<) and suffixed by a greater than sign (>). The following tokens are supported:

- <YYYY>-four digit year, e.g. 2010
- <YY>-two digit year, e.g. 10 for 2010
- <MM> - two digit numerical month, e.g. 01 for January
- <Mmm> - three character abbreviation of the month, e.g. Jan for January
- <MMM> - three character upper-case abbreviation of the month, e.g. JAN for January
- <mmm> - three character lower-case abbreviation of the month, e.g. jan for January
- <DD> - two digit day of the month, e.g. 01 for the 1st of the month. Defaults to 01 when `daily_resolution` is set to `no`
- <DDD> - three digit day of the year, e.g. 123 for the 3rd May (124 if it is a leap year). Defaults to the first of the month when temporal resolution is monthly
- <hh> - two digit hour specification in 24-hour format, e.g. 06 for six AM.
- <mm> - two digit minute specification, e.g. 05 for five minutes past the hour.
- <FEROOT> - used by some internal scripts and tutorials to refer to the root directory to which FluxEngine was installed. This is used, for example, to access data that comes packaged with FluxEngine.

7.5 Data layers in detail

The term 'data layer' is used to describe a geographical dataset and any accompanying metadata. Data layers are used by FluxEngine for inputs, intermediate products and outputs. Configuration files specify all the input data layers which will be needed for the flux calculation, but you only need to specify the input data layers which will be used given the specific options you've selected. For example you do not need to specify biology input files if the 'process indicator layers off' is set using the `-1` flag, and you do not need to specify a sea surface skin temperature data layer if you set `sst_gradients = no` and supply specify data for `sstfnd` in the config file. If you try to run the FluxEngine without a required input you'll get an error message telling you

which input data layer(s) are missing. If you specify data layers which are not needed for the calculation options specified they will simply be added as an extra variable in the netCDF output file(s) and no error messages or warnings will be displayed. This can be useful if you want additional data packaged with the output files for convenience when performing further analysis steps after running FluxEngine. This is used with the `ice` data layer, for example. which isn't used in the main flux calculation, but it is useful to have ice coverage data in the output files so that the net budgets tool can use it.

Data layer paths

To specify an input data layer the configuration file must specify a minimum of two attributes: a path to the netCDF / .nc file, and a prod (variable name within the netCDF file). The path can be absolute or relative. Windows users might experience some problems using absolute paths. If you have problems with this please let me know (tom: t.m.holding@exeter.ac.uk) and I'll try to fix it. Paths are specified in the config file using the data layer name with the `_path` suffix, like so:

```
datalayername_path = path/to/data/filename.nc
```

One important change in this version of FluxEngine is that you should specify the path including a the filename for the netCDF file, rather simply a directory. This provides greater flexibility when working with data from many different sources. To help with this two standard Unix glob patterns can be used to specify patterns of file names: `?` and `*`. These will match any single character/digit, or any number of characters/digits (including no characters), respectively. For example to specify the location of ice coverage data you could use:

```
ice_path = path/to/data/20100101_???-ice*.nc
```

This will match any file with a name that starts with `20100101_` followed by any three characters/digits, then the characters `-ice`, followed by any number of characters/digits and ending in `.nc`. Note that glob patterns cannot be used to match directory names and can only be used to specify the pattern that FluxEngine will use to match the netCDF file itself.

In most cases the file you want to use as input will depend on the point in time that you are analysing. In this case you can use tokens (described in section 7.4) to specify date and time related changes to file or directory names. For example, if your ice coverage data files are prefixed with the year and month they were recorded, and organised into subdirectories for each year, `ice_path` might be defined like this:

```
ice_path = path/to/data/<YYYY>/<YYYY><MM>_???-ice*.nc
```

Here `<YYYY>` will be replaced with the four digit year (e.g. 2010 for 2010) and `<MM>` will be replaced with the two digit representation of the month (e.g. 01 for January).

Data layer products

The second required attribute of a data layer is its product (or 'prod'). This is the name of the variable within the netCDF file. It is specified in the configuration file by using the `_prod` suffix with the data layer name. The minimal specification (in this case for the 'ice' data layer) could therefore look like this:

```
ice_path = path/to/data/<YYYY>/<YYYY><MM>_???-ice*.nc
ice_prod = sea_ice_fraction_mean
```

7.5.1 Using tokens to reuse selected data

Tokens can be used in directory names allowing you to reuse data for selected inputs. For example, you may have data for multiple years of sea surface temperature, but only one year's worth of salinity data (or this may be a multi-year average). In this case you can reuse the salinity data for each year you have of other data. To do this simply specify the file path of the salinity data by hard-coding the year string into the file path, while specifying other data layers using tokens as usual. This could look as follows:

```
salinity_path = path/to/data/2010/2010<MM>_woa-salinity.nc
sstfnd_path = path/to/data/<YYYY>/<YYYY><MM>_OCF-SST-GLO-1M-???-REYNOLDS.nc
```

7.5.2 Optional data layer attributes

There are several optional attributes which can be configured for each data layer. They can be set using the same `datalayername_suffix` notation used for the path and products above. These are:

- `_stddev_prod` - product name of a variable containing standard deviation data for the data layer
- `_count_prod` - product name containing the number of samples used to calculate standard deviation
- `_netCDFName` - the variable name used to label this data layer in the output netCDF file/s
- `_units` - a string description of the units
- `_minBound` - minimum allowed value)
- `_maxBound` - maximum allowed value
- `_standardName` - short standardised description of the variable/data layer
- `_longName` - human readable description of the variable/data layer
- `_temporalChunking` - the number of time points in each file (see section 7.7)

- `_temporalSkipInterval` - sets skip interval between temporal indices
- `_timeDimensionName` - specify the name of the time dimension (default is `time`)

For example to overwrite the `minBound` and `maxBound` attributes for the ice coverage data layer as well as rename it in the output files you can add the following lines to a configuration file:

```
ice_minBound = 0.0
ice_maxBound = 100.0
ice_netCDFName = ice_percent
```

Any value outside of this range will be replaced with missing values.

Default metadata values are stored in `settings.xml` in the `fluxengine` core directory. `settings.xml` is only mentioned as the definitive place to look up default values and shouldn't be modified because this would change the values for all subsequent FluxEngine runs, and can lead to difficult to detect errors in future runs. If you need to change one of these values you should always overwrite it using a configuration file instead (as shown above).

7.5.3 Data layer preprocessing

It is sometimes convenient to apply some pre-processing to a data layer before it is used for any computations. There is an additional data layer attribute which allows the user to specify a list of functions to be applied immediately after the data layer read in. A number of simple preprocessing functions are bundled with FluxEngine (these can be listed by running `ofluxghg_run.py` with the `-list_preprocessing` flag, or by viewing the functions directly in the `data_preprocessing.py` file in the `fluxengine_src` directory). For users comfortable with python, custom pre-processing functions can be added to `data_preprocessing.py`. These will be automatically detected when running FluxEngine available to use in configuration files. Before modifying this file you should familiarise yourself with the guide for developers notes in section 9.

To specify pre-processing functions the `_preprocessing` suffix is used with a list of function names separated by commas. Each function will be applied in the order it appears in this list. For example adding the following line to a config file will first transpose the 2D matrix, then convert from Kelvin to Celsius:

```
sstskin_preprocessing = transpose, kelvin_to_celsius
```

Note that no checks are made to ensure the original values are in Kelvin to begin with, and it is up to the user to ensure that any pre-processing functions are applied appropriately.

7.6 Working with different temporal resolutions

The temporal resolution over which the FluxEngine will undertake the flux computation is, by default, one month. This can be changed by defining a new time step in the

configuration file by setting `temporal_resolution`. The required format is `D hh:mm` for the number of days, hours and minutes between time steps, and it is important that the hour and minute components are exactly two digits (so there should be a preceding 0 if necessary). Four examples are given below, which define a time step of one week, twelve hours, one hour and 30 minutes, and five minutes, respectively:

```
temporal_resolution = 7 00:00 #one week timestep
temporal_resolution = 0 12:00 #twelve hour timestep
temporal_resolution = 0 01:30 #one hour and 30 minutes time step
temporal_resolution = 0 00:05 #five minute time step
```

The temporal resolution should not be higher (smaller time step) than that of the highest resolution input data, otherwise FluxEngine will duplicate some calculations. If temporal resolution is set lower (larger time step) than that of your highest resolution input data then FluxEngine will not use all of this data. Depending on your requirements, this may be what you want, but could indicate that higher resolution inputs should be re-analysed to create a matching data set which has a lower temporal resolution.

When working with higher temporal resolutions than the default monthly resolution, it will be necessary to modify the output filenames and/or the directory structure. Leaving them as the default is likely to result in FluxEngine overwriting some of the previous outputs. See section 7.8 for examples of how to do this.

7.7 Indexing input data using the temporal dimension

Data are sometimes formatted as netCDF files which utilise a temporal dimension in addition to two spatial dimensions (longitude and latitude). Configuration options are provided to allow FluxEngine to utilise input data which use a temporal dimension. The `datalayername_temporalChunking` option can be set for any input data layer and indicates how many time steps are in a single input file. It is therefore important to set this in the context of the temporal resolution over which FluxEngine will be run. For example, if you are using FluxEngine to calculate gas fluxes with a daily resolution and your input wind speed data files provide daily resolution but contain a single file for each week, your configuration file should include something like the following:

```
temporal_resolution = 1 00:00 #daily temporal resolution
windu10_path = path/to/wind_data<DDD>.nc
windu10_prod = windspeed_mean
windu10_temporalChunking = 7 #Each file contains 7 time points
```

7.8 Setting output filenames and directory structure

You can define the output file names by setting `output_file` in the configuration file. For example, when using a temporal resolution of one day you'll need to ensure output file names are unique so they should include the day that the output corresponds to. Below are two possible ways you could define this:

```
output_file = OceanGluxGHG_output_<YYYY>_<DDD>.nc #year and day of the year
output_file = OceanGluxGHG_output_<YYYY>_<MM>_<DD>.nc #year, month and day of
the month
```

Similarly, it might be convenient for FluxEngine to use a different directory hierarchy to organise the output. The default output directory structure is <YYYY>/<MM> to be compatible with the flux budgets tool, but to define a custom output directory structure you can specify `output_structure` in the configuration file. This can be another way to ensure that output files have unique file paths/names. For example, to group output files by year and day, you could use

```
output_structure = <YYYY>/<DDD> #e.g. 1991/019 for 19th Jan 1991
```

or to group output files by year and month, then by day

```
output_structure = <YYYY>_<MM>/<DD> #e.g. 1991_01/19 for 19th Jan 1991
```

7.9 Grouping output from multiple time points into a single file

FluxEngine output for more than one time point can be written to a single file. In this case each netCDF variable will contain a temporal dimension. This can be useful when running FluxEngine at high temporal resolution to prevent generating hundreds or thousands of separate files. To do this, set the `output_temporal_chunking` option in the configuration file, which defines how many time steps to group into each file. For example, if you're running FluxEngine with an hourly resolution, you might want to have a single file per day. This can be achieved by including

```
output_temporal_chunking = 24 #24 timesteps in each output file
```

7.10 Filtering input data using a mask

You can specify a mask by specifying the path and prod for the `mask` data layer. The mask must have the same spatial dimensions as the input data. It specifies for which grid cells the flux calculation should be performed (where the mask is non-zero) and which grid cells should be ignored (where the mask is equal to zero). The usual data/time tokens can be used when specifying the path of a mask, allowing different masks to be used for different time points (e.g. if filtering out grid cells with high wind speed). An example configuration file snippet to define a mask is as follows:

```
mask_path = data/mask/<YYYY><MM>_maskfile.nc
mask_prod = high_winds
```

8 Bundled tools

FluxEngine comes bundled with a number of command line tools. These are designed to be used in conjunction with FluxEngine, and perform closely related tasks such as converting or merging data files, reanalysing fCO₂ data, or calculating monthly/annual net flux budgets. The tools are added to your environment path when FluxEngine is installed so can be accessed via the command line from any directory (remember to activate the correct virtual environment). A description of how to use each tool, and a list of each their command line options, can be viewed by running the tool with the `-h` or `-help` option. A short description of each tool is given below.

`fe_run.py` - Command line tool for running FluxEngine using the settings specified in a configuration file.

`fe_calc_budgets.py` - Calculates integrated net air-sea gas fluxes from FluxEngine output.

`fe_update_config.py` - Updates old FluxEngine configuration files to be valid for the current version. Works with configuration files from FluxEngine v3.0 and newer.

`fe_resample_netcdf.py` - Resamples a 1° by 1° netCDF data to a 5° by 4° grid. This tool is kept for historic reasons as it was used to resample the input data for validating against published Takahashi climatology.

`fe_text2ncdf.py` - Converts in situ data to a netCDF file. See section 8.3 for example usage.

`fe_ncdf2text.py` - Converts netCDF file (e.g. a FluxEngine output file) to a flat text file.

`fe_append2insitu.py` - Appends FluxEngine output to a pre-existing text formatted data file. Intended to allow FluxEngine output to be added as columns for *in situ* data.

`fe_reanalyse_fco2_driver.py` - Generates fCO₂ reanalysed to a consistent temperature and depth. See section 8.2 for details and example usage.

`fe_compare_net_budgets.py` - Simple tool which compare net budgets between two runs. This is used as the basic of some of the checks found in the verification scripts (see section 3). Further tools for performing common verification tasks are available to use in custom Python scripts by importing `fluxengine.tools.lib_verification_tools`.

`fe_verify_socatv4.py` - Runs the verification script using interpolated SOCAT CO₂ data. `fe_verify_takahashi09.py` - Runs the verification script using CO₂ data from Takahashi 2009.

`fe_tutorials.py` - Starts a Jupyter notebook server and opens the interactive tutorials in a web browser.

8.1 Calculating net fluxes with `fe_calc_budgets.py`

This tool will calculate monthly and annual the net flux budgets from monthly FluxEngine output. FluxEngine output is used as input to the tool and must adhere to a `<YYYY>/<MM>` output directory structure (this is the default, see section 7.8 for details). Fluxes can be calculated regionally by setting the region(s) you'd like to calculate fluxes for. After running the tool a single text file will be generated for each region. This file contains a monthly breakdown of the estimated net, missing and gross downward (into the ocean) and upward (into the atmosphere) flux. These regions must correspond to the values of regions in the region mask file. Usage of the tool is as follows:

```
fe_calc_budgets.py -d FluxEngine/output -v -lf land_mask_file.nc -mf
  region_mask_file.nc -o your/output/directory
```

The `-d` or `--dir` option specifies the directory to the FluxEngine output which the tool will use to calculate the net flux. The `-v` option increases verbosity (prints extra information for the user). The `-lf` or `--landfile` options specify a netCDF file with a variable describing the proportion land in each grid cell. An additional option `--landdataset` (or `-ld`) specifies the variables/product name in the land netCDF file, but defaults to `land_proportion` if not set. Similarly `-mf` or `--maskfile` defines the region mask netCDF file while `--maskdatasets` or `-md` specifies the variable names of each region's mask within the netCDF file. Finally, the `-o` or `--outroot` option defines the root directory to which `ofluxghg-flux-budgets.py` will write output files. Note that the FluxEngine output, land file and region mask must all have the same spatial dimensions (grid size). Region names can be set using `-r` or `--regions`, and there must be one for each mask dataset used (defaults to 'global', e.g. `--maskdatasets` is not set).

Examples of how land and mask files should be formatted can be seen by examining the land and mask files used in the verification scripts. These are located `<FEROOT>/data/onedeg_land.nc` and `<FEROOT>/data/World_Seas-final-complete_IGA.nc`, respectively.

8.2 Using `fe_reanalyse_fco2_driver.py` to reanalyse data to a consistent temperature and depth

`fe_reanalyse_fco2_driver.py` is an external tool which comes bundled with FluxEngine and which implements the method described by Goddijn-Murthy (2015) to reanalyse fCO_2 / pCO_2 , sampled in situ from different depths and/or with different instantaneous temperatures, to a consistent temperature field. A driver script (`fe_reanalyse_fco2_driver.py`, in the `tools` directory) is provided to facilitate using this script with *in situ* data via the command line. This tool has many options, which can be viewed by running it with the `-h` command:

```
fe_reanalyse_fco2_driver.py -h
```

A simple example command to generate a netCDF file containing fCO_2 values from a tab delimited text file is as follows:

```
fe_reanalyse_fco2_driver.py -input_dir path/to/data -input_files datafile.tsv
-sst_dir path/to/ReynoldsSST/ -sst_tail 01_OCF-SST-GLO-1M-100-REYNOLDS.nc
-output_dir output/path -socatversion 6 -usereynolds -startyr 2008 -endyr
2015
```

The `-socatversion 6` option tells the tool to expect column headings to conform to those used by SOCATv6. If your data header does not conform to a SOCAT naming convention you can specify each column names for each of the required variables (see the available options using the `-h` option for details).

To produce output as text files instead of NetCDF files, use the `-asciioutput` option.

Example commands utilising different options can be found in the comments at the top of `fe_reanalyse_fco2_driver.py`.

8.3 Using `fe_text2ncdf.py` to create FluxEngine compatible NetCDF files

FluxEngine requires all input data to be supplied in NetCDF files. While FluxEngine will accept correctly formatted NetCDF files that have been created by any method, the `fe_text2ncdf.py` tool is provided as a convenient and flexible tool to convert flat text formatted data into NetCDF files. The tool can run from the command line, or imported as a Python module (`import fluxengine.tools.lib_text2ncdf`). Basic operation using the commandline is as follows:

```
fe_text2ncsf.py inFiles inputfile1.tsv inputfile2.tsv --startTime 2010-01-01
--endTime 2010-12-31 --ncOutPath output/path/file.nc --delim '\t'
--latProd latitude --lonProd longitude --latResolution 1 --lonResolution 1
--dateIndex 0 --temporalResolution '0 12:00' --colNames 1 2 5 'SST [C]'
'windspeed [ms-1]' --parse_units --limits -90 90 -180 180
```

This will convert two files (`inputfile1.tsv` and `inputfile1.tsv`) into NetCDF files for the year 2010. Output files will be saved to `output/path`. The delimiter separating values in the input files is defined as a tab, and the latitude and longitude column names in the input files are `latitude` and `longitude` respectively. Output files will consist of a 1° by 1° grid, with means, counts and standard deviations calculated for this grid size. `--dateIndex` specifies the column index containing the date/time. The `--temporalResolution` defines the time step used (by default a separate output file will be created for each time point). The `colNames` option allows you to specify column names (or indices, or a mixture of names and indices) to be converted to NetCDF (indexes start from 0). This should not include longitude, latitude of data/time columns, as these are specified separately by `--lonProd`, `--latProd` and `--dateIndex`, respectively. In this case column numbers 1, 2 and 5, as well as columns with the names `SST [C]` and `windspeed [mc-1]`, will be converted. The `--parse_units` option tells the tool to automatically interpret any text in the header which is contained between square brackets (`[` and `]`) as the units, which will be added as metadata in the output NetCDF.

As with other tools, a full description of the various options can be found by running the tool using the `-h` option. This description will always contain the most up to date

description of the options and usage.

8.4 Automatically updating old configuration files

Configuration files for FluxEngine version 4.0 are mostly compatible with version 3.x, however some changes in variable names will prevent old configuration files from running. A simple command line utility is provided (`fe_update_config.py`) to automatically update old configuration files to the compatible with the current version of FluxEngine. This utility will work for any configuration file for FluxEngine 3.0 or newer. Usage is simple:

```
fe_update_config.py path/to/old_config.conf path/to/updated_config.conf
```

If you want to ensure a file is written to the updated configuration file path, regardless of whether the old configuration file required modifying (e.g. as part of an automated workflow) you can add `-alwayswrite` to ensure a copy will be written to your output path.

If version information is missing from an old configuration file, you can manually add it, or use the `-oldversion` flag to tell the tool which version to expect. See the tool's help (`-h`) information for more details.

9 Guide for developers

In FluxEngine v3.0 we have made changes to make it easier for people to modify and extend the functionality of FluxEngine. We have implemented a more consistent structure to the code and added the ability to easily extend certain aspects of FluxEngine's functionality without modifying the core code. The following sections provide some background information on how to do this. This is intended for users who are comfortable programming in Python.

9.1 Adding pre-processing functions

Pre-processing functions are defined in `fluxengine\core\data_preprocessing.py`. When parsing `*_preprocessing` data layer properties in configuration files, FluxEngine searches the function names defined in this file, and so any function which is added will be immediately available for use as a pre-processing function. However, there are certain requirements for the function to operate harmoniously with FluxEngine. These are listed below:

- Pre-processing functions must have a single argument, and this must be the `DataLayer` instance which corresponds to the input data layer which is being transformed. Detailed of the `DataLayer` class can be found in `DataLayer.py`.
- `DataLayers` should be modified in place. Returned values are ignored.

- Pre-processing functions should only modify the (1D) `fdata` attribute of the `DataLayer` instance. The exception to this is if it is convenient to modify 2D view of this (the `data` attribute), in which case you must call `datalayer.calculate_fdata()` afterwards. This is because, while in most cases `fdata` is a view of `data`, in some cases `fdata` may be a copy of `data` and hence any changes to `data` will not be reflected in `fdata`. `fdata` is used for all calculations, so it is important that any changes are copied to this attribute.
- If a pre-processing function modifies the dimensions or any of the meta data associated with a *DataLayer* it must also manually update the relevant attributes as these will not be automatically reflected.

9.2 Adding k-parametrisation functors

The gas transfer velocity calculation is fully customisable and it is possible to write custom 'functor' classes to add to the built-in parameterisations available. In older version of FluxEngine, this was achieved by adding the parameterisations to the `fluxengine\core\rate_parameterisation.py` file, thereby making them globally available to any project. As of FluxEngine v4.0.3, the new recommended way to add parameterisations is to implement them in a separate file which is stored with your other project files / config files etc. You can then pass the path to this file as an argument when running FluxEngine and the contents of the file will be executed, and any parameterisations will be available for use in configuration files. The advantages to this are 1) Parameterisations can be stored with other project-specific files, such as config files, and therefore moved around with them. 2) Custom parameterisations aren't lost when updating FluxEngine. And 3) parameterisations can be easily shared, e-mails or attached as supplementary material to manuscripts.

You can pass the file path containing custom gas transfer velocity parameterisations to FluxEngine in two ways. Firstly, using the command line `fe_run.py` tool, you can supply the path using `-custom_gas_transfer_parameterisation` (`-gtvp` for short) followed by a string containing the relative or absolute file path. For example:

```
fe_run.py test.conf -l -custom_gas_transfer_parameterisation
"a_directory/my_custom_gtv.py"
```

The second method can be used when driving FluxEngine directly from Python. Simply provide the `customGTVPath` argument to the `run_fluxengine` function:

```
import fluxengine.core.fe_setup_tools as setup;
setup.run_fluxengine("test.conf", 2010, 2010,
    customGTVPath="a_directory/my_gtv.py");
```

To implement a new parameterisation you must write a 'functor' class (a class which implements the `__call__` member function), and the class must be derived from the `KCalculationBase` class (also defined in `rate_parameters.py`) and implement four functions:

- `__init__` - Initialises the class. This must, at a minimum, set `self.name`. You can add any arguments which the class needs to initialise to the function signature and provided they are added as variables with the same name/s in the configuration file they will be automatically passed to the functor during initialisation (see the notes on adding configuration variables below).
- `input_names` - This should return a list of `DataLayer` names (strings) which are required as inputs to the gas transfer velocity calculation.
- `output_names` - This should return a list of `DataLayer` names which are modified or written to. These can be existing or new `DataLayers`. Any non-existing `DataLayers` will be created in the correct dimensions (but filled with missing values) by FluxEngine prior to performing the gas transfer velocity calculation.
- `__call__` - This performs the gas transfer velocity calculation, and will contain all the implementation details for your specific case. In addition to `self`, an argument called `data` is passed to this function which contains a dictionary of each `DataLayer` available to FluxEngine. These can be assessed by using the `DataLayer` name as a key. Note that you should not create new `DataLayer` instances, add entries to this dictionary or change `DataLayers` which are not listed by name in the list returned by `output_names`.

It is best practice to modify the 1D 'flat' data attribute of output `DataLayers` (`DataLayer.fdata`). If the 2D `DataLayer.data` attribute is modified you should update the `fdata` attribute by calling `DataLayer.calculate_fdata` for the `DataLayers` which have changed. This is because, while in most cases `fdata` is a view of `data` to avoid unnecessary duplication, this is not guaranteed on all systems.

The final gas transfer velocity output should usually be written to the `k` data layer, as this is what will be used by FluxEngine to calculate air-sea gas flux. Additionally, it can be a good idea to set the `DataLayer.long_name` and `DataLayer.short_name` attributes of `k` to provide a description of the parameterisation used because this will be copied to the output netCDF files.

An example implementation, as well as all the pre-bundled gas transfer velocity functors, can be found in `fluxengine\core\rate_parameterisation.py`.

9.2.1 Adding configuration variables

You can add variables to the configuration file and these will be immediately available in the flux engine code (encapsulated in the `runParams` 'namespace'). For example, if you add

```
my_new_var = 100.0
```

to the configuration file. This can be referenced in the FluxEngine code using `runParams.my_new_var`. This is utilised by different `k`-parameterisation functors which

require additional variables to initialise correctly (see the definition of `k_generic` in `rate_parameterisation.py` for an example).

Additional configuration variables are interpreted as a float if they're formatted as a valid float, otherwise they're interpreted as a string.

To avoid naming conflicts (since all config variables are imported to the same namespace) it is good practice to add a prefix to the name of any custom configuration variables. Typically this should be the name of the k-parameterisation functor it is associated with. For example, the `k_generic` functor requires 5 additional variables each of which begin with `k_generic` (e.g. `k_generic_sc`).

9.3 Contributing

If you're a git user you can fork the FluxEngine repository at <https://github.com/oceanflux-ghg/FluxEngine> and if you develop extensions or functionality which might be useful to the wider community, or spot and fix any bugs, you can share them by sending us a pull request. Alternately, if you don't know what any of that means but you've developed an extension or fixed a bug which you think will be useful to others, you can e-mail me at t.m.holding@exeter.ac.uk.