FluxEngine v3.0: Changes and User Guide

June 18, 2018

Contents

1 Overview and major changes

The major changes, new to FluxEngine version 3.0 include:

- *Standalone execution* - Various changes which make it considerably easier to run locally (i.e. without access to a remote server). A big part of this is that FluxEngine no longer requires input files for data layers which will not be used given a the configuration options specified. For example, you no longer need to specify the location of sea surface skin temperature data, if you're using foundation temperature. This means you do not need access to data which might be irrelevant to your particular use case in order to run FluxEngine.

- *Greater flexibility* - Use of Unix-like glob pattern matching to specify input data means file names no longer need to conform to particular formats. This makes it easier incorporate new and user supplied sources of data. A modular and extendible system for data pre-processing has been added to make it easier to use data from a wider range of sources.

- *Substantial restructuring* - The source code has been substantially restructured. While this is an ongoing process, having a consistent method for reading, processing and outputting data layers reduces the scope for bugs, provides a simpler conceptual model to understanding and extending FluxEngine functionality. For example, additional data pre-processing and gas transfer velocity calculations can be added and tested without modifying the core FluxEngine code.

- *Python workflow* - Python, and some Python modules are now the only dependencies needed by FluxEngine. The Perl driver script has been transcribed into Python and updated. FluxEngine can run from the command-line or imported as a Python module providing a simple API interface for more advanced use-cases.

- *User-friendly installation and verification* - Installation scripts will automatically install missing dependencies for MacOS and Linux based systems. Installation instructions are provided for Windows users. Verification scripts are provided to ensure FluxEngine is operating correctly, and can be used to verify modifications.

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 suggestions about useful potential features to include in the next release. 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 (see above).

2 Downloading FluxEngine v3.0

FluxEngine can now be ran on a standalone computer so you don't need access to a remote host. The latest version can be downloaded from https://github.com/oceanflux-ghg/FluxEngine/archive/master.zip, or if you use Git you can clone the FluxEngine repository here: https://github.com/oceanflux-ghg/FluxEngine.git

After downloading unzip the files into a location of your choice. This should leave you with the directory hierarchy shown below:

```
FluxEngine (or FluxEngine-master) - referred to as the root folder
├── configs - contains verification and example configuration files
├── data - input data required to run verifications and reference
│   output
└── fluxengine_src - FluxEngine source code
    └── tools - additional tools which can be run separately to
        FluxEngine
```

3 Installing dependencies

One of the changes from v2.0 is that FluxEngine can be run entirely using Python 2.7. That being said there are several modules which are not part of the Python 2.7 standard library. These are `netCDF4`, `numpy` and `scipy.integrate`. If you don't already have these installed, you'll need to install them before running FluxEngine. They can be installed manually (the recommended way to do this is using `pip` which usually comes bundled with Python 2.7) or they can be installed using the dependency installation scripts which can be found in the root directory for MacOS and Linux.

3.1 MacOS

An installer script is available for MacOS which will automatically install dependencies. While it's provided with no guarantees, the script shouldn't overwrite anything without asking you first. To run it, open Terminal, navigate to the FluxEngine root folder and run

```
./install_dependencies_macos.sh
```

If you see an error about not having permission to execute, chances are you'll need to add this permission before running the install script. To do this run the following:

```
./sudo chmod +x install_dependencies_macos.sh
```

3.2 Linux

An installer script is available for Debian based Linux systems and will automatically install the dependencies needed to run FluxEngine. It has only been tested on Ubuntu 16.4, and is provided with no guarantees. To run it, open Terminal, navigate to the FluxEngine root folder and run

```
./install_dependencies_ubuntu.sh
```

If you see an error about not having permission to execute, chances are you'll need to add this permission before running the install script. To do this run the following:

```
./sudo chmod +x install_dependencies_ubuntu.sh
```

3.3 Windows

An installer script isn't provided for Windows but the following steps should enable you to install the dependencies required to run FluxEngine.

Step 1) Since Python 2.7 doesn't come with Windows you may not have it installed. If you already have Python 2.7 go to step 2, if not then go to https://www.python.org/downloads/ and download the installer. At the time of writing the most recent version of Python 2.7 was 2.7.14. Do not download version 3.x, FluxEngine will not run on it. Once downloaded run the file and you'll be guided through the installation process. Make a note of your installation directory (it will probably be something like `C:\Python27\` or `Users\<your username>\AppData\Local\Programs\Python\Python27`).

Step 2) Add Python and the Python `Scripts` folder to your `PATH` environment variable. This will allow you to run python scripts without having to navigate to specify the whole Python installation directory each time. To do this in Windows 10, follow these instructions: `https://www.youtube.com/watch?v=uXqTw5eO0Mw` using the installation directory from the previous step.

Step 3 Use `pip` to install three Python modules: `netCDF4`, `numpy` and `scipy.integrate`. Pip is a package manager, which comes with Python and allows you to add/remove additional libraries. First, open a new command prompt window (or close and reopen an existing one - this is important or your `PATH` environment variable won't have updated). Type the following commands:

```
pip install --upgrade pip
pip install numpy
pip install netCDF4
pip install scipy
```

You should now be ready to run FluxEngine. More detailed information on installing and configuring Python 2.7 on Windows can be found here: `https://docs.python.org/2.7/using/windows.html`

4 Verifying

Two verification scripts come bundled with the 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 discrepencies.

To verify using Takahashi2009 (T09) and/or SOCATv4 data run the following command/s from FluxEngine's root directory, respectively:

```
python verify_takahashi09.py
python verify_socatv4_sst_salinity_gradients_N00.py
```

Alternatively, you can write python scripts to verify against your own reference data sets using the functions provided in the `validation_tools.py` file in the `tools` directory.

5 Running FluxEngine v3.0

In previous versions FluxEngine was ran using a bash script which invoked a Perl script to parse a config file and run the core Python FluxEngine code. This has been considerably simplified in version 3.0 so that everything is handled by Python. This has the advantage of reducing dependencies and moving toward a more API-like interface for executing FluxEngine. An execution script (`ofluxghg_run.py`, found in the root directory) is still provided, which functions similarly to the previous Perl script (described below). To run the FluxEngine for

5.1 Using the command line script

`ofluxghg_run.py` is found in the FluxEngine root directory can be be used as a standalone tool to run the FluxEngine, for example

```
python ofluxghg_run.py configs/example_config.conf -l
```

will run the flux engine using the options specified in a configuration file called `example_config.conf` located in the `configs` subdirectory of the current working directory. This assumes `ofluxghg_run.py` is being executed from the FluxEngine root directory, but it can be executed from anywhere, provided that 1) the path of the config file is reachable (absolute file paths can be used), 2) `ofluxghg_run.py` can find the fluxengine_src folder (either it has been added to your Python paths, moved somewhere which is already in your Python path, or it is a subdirectory of the current working directory) and 3) the input data specified in your config file is reachable from the current working directory (these can also be absolute paths). The `-l` option tells FluxEngine to run without process indicator layers, this reduces the time taken to run and results in smaller output files.

The general syntax for running `ofluxghg_run` is

```
python ofluxghg_run.py config [-options]
```

where `[-options]` is an option list of options to change how FluxEngine runs. A list of valid options can be listed by using the `-h` flag, e.g. `python ofluxghg_run.py -h`. Some options require you to specify a value, for example to set the start and end years you can use

```
python ofluxghg_run.py configs/example_config.conf -year_date 2000 -year_end
    date -l
```

to run FluxEngine from the start of the year 2000 to end of the year 2010. Other date formats are supported, for example ..... Note: A useful option for testing is `-S1` which will only run FluxEngine for a single month. This allows you to check the output and 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_src.fe_setup_tools` module and use `run_fluxengine` function. An example of how to use this function can be found in `ofluxghg_run.py`. For more control over how FluxEngine is driven, e.g. if you need to use a single configuration file and overwrite a subset of parameters to run a suite of similar simulations, see the tools and functions provided in `fe_setup_tools` in the `fluxengine_src` directory. Note that these tools are intended for use by users who are proficient in Python. They haven't undergone intensive testing and likely to contain bugs (especially in Windows environments). Bugs can be reported as described in section 1.2.

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 `fe_setup_tools.py` 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.

6 Creating and modifying configuration files

The format of configuration files has changed from previous versions. Example annotated configuration files can be found in the `configs` directory. Variables are specified in the config file by name, followed by an equals sign (`=`) and then the value. In contrast to previous versions the equals sign is now required, but this means that whitespace can be used in variable values. 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 variables is given below:

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

6.1 Configuration file specification

`src_home`
The file path to the `fluxengine_src` directory. Note that this is no longer used in version 3.0, but is reserved and may be used in the future to distinguish between different versions of FluxEngine which may be installed on the same computer.

`flux_calc`
The flux calculation to perform. Valid options are:

- `bulk` - i.e. $F = k\alpha_W(pCO_{2_W} - pCO_{2_A})$, 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_{2_W} is the partial pressure of CO_2 in the surface sea water and pCO_{2_A} is the partial pressure of CO_2 in the atmosphere.
- `rapid` - As described in Woolf et al. (2012) *Journal of Geophysical Research: Oceans*. $F = k(\alpha_W pCO_{2_W} - \alpha_A pCO_{2_A})$.

- `equilibrium` - As described in Woolf et al. (2012) *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 a `D hh:mm` format to specify the length of timesteps in days, hours and minutes respectively. A full explanation, with examples, is given in section **??**.

`use_sstskin`, `use_sstfnd`, `sst_gradients`, `cool_skin_difference`
Determines the source/s of sea surface temperature to use (skin or foundation or both). Valid values for `use_sstskin`, `use_sstfnd` and `sst_gradients` are `yes` or `no`. At least one of `use_sstskin` or `use_sstfnd` must be set to `yes`. If only one is specified and `sst_gradients` is set then the other is estimated using the following equation:

```
sstskin = sstfnd - 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 for. This applies an annual correction according to

```
pco2_increment = (year - pco2_reference_year) * pco2_annual_correction
```

`datalayername_path`, `datalayername_prod`
These define each input data layer. A full explanation with worked examples is provided below (section 6.4).

`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 as input data layers, and write output to one or more data layers (typically the k datalayer). For a list of available parameterisations you can run the ofluxghg_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_src 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 writting new k functors you should read the guide for developers (section 8), as this describes best practices and provides information on potential pitfalls.

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`, `o2`, `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 7), making 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 6.3). 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.

Full meta data, including a list of data layer names recognised by FluxEngine, de-

fault 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.*

6.2 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 parametrisations may require other variables to be specified and you should consult any documentation or guidance specific to the parametrisation being used, or examining the initialiser function (`__init__`) of the relevant parametrisation functor the `rate_parameterisation.py` file.

6.3 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.

The only path you cannot use tokens in is the `src_home` entry.

6.4 Data layers

The term 'data layer' is used to describe a 2D dataset, and any accompanying meta-data, which is used by FluxEngine either as input, an intermediate product or output. Configuration files must specify all the input data layers which will be needed, but you only need to specify the input data layers which will be used given the specific options youve selected. For example you do not need to you do not need to specify biology input files if the 'process indicator layers off' is set using the `-l` flag, and you do not need to specifiy a sea surface skin temperature data layer if `use_sstskin = no` is set in the config file. If you try to run the FluxEngine without the required inputs youll get an error message telling you which input data layer was missing. If you specify data layers which are not needed for the calculation options specified they will simply be added to the output data layer.

Data layer paths

To specify an input data layer the configuration file must specify a minimum of two attributed: 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 Ill try to fix it. Path 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 6.3) 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 looks like this:

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

6.5 Using tokens to match data sampled at different temporal resolutions

Since these tokens can be used in directory names it can be useful if, for example, you have data for multiple years for one data layer, but only have data covering one year for another. In this case you would specify the file path of the first data layer using the appropriate year token, and the second by hard-coding the year string into the file path. Supposing we only has sea surface foundation temperature for the year 2010, but ice coverage data for each year we're interested in, this might look as follows:

```
ice_path = path/to/data/<YYYY>/<YYYY><MM>_???-ice*.nc
sstfnd_path = path/to/data/2010/2010<MM>_OCF-SST-GLO-1M-???-REYNOLDS.nc
```

6.6 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

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_src` directory, but shouldn't usually be modified because this will change the behaviour for all FluxEngine runs. These values can always be overwritten in the configuration file as described above.

6.7 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 8.

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 Kelbin to begin with, and it is up to the user to ensure that any pre-processing functions are applied appropriately.

6.8 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 timestep in the configuration file by setting `temporal_resolution`. The required format is `D hh:mm` for the number of days, hours and minutes between timesteps, 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. You can define the output file names be setting `output_file` in the configuration file. For example, when using a temporal resolution of one day to ensure output file names are unique they should include the day which the output was generated for. 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>
```

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

```
output_structure = <YYYY>_<MM>/<DD>
```

This is a flexible system which should be able to suit most use-cases.

7 Bundled tools

The `tools` subdirectory in the `flexengine_src` directory contains various tools designed to be used in conjunction with FluxEngine. These are discussed below.

`resample_netcdf.py` - Resamples a 1° by 1° netCDF data to a 5° by 4° grid.

`text2ncdf.py` - Converts in situ data to a netCDF file.

`ofluxghg_flux_budgets.py` - Calculates integrated net fluxes.

`compare_net_budgets.py` - Simple functions which compare net budgets between two runs. Examples of their use can be found in the verification scripts (see section 4). To use these you must first import them into a python project using `import fluxengine_src.tools.compare_net_budgets`.

`validation_tools.py` - This file contains several functions to perform common verification tasks, such as comparing the netCDF outputs of two FluxEngine runs, or calculating and comparing the global flux budgets between a new run and a reference data set. To access these functions they must be imported into a python project (e.g. by using `import fluxengine_src.tools.validation_tools`). The function which is more likely to be useful is `validation_run`. A description of how to use this function is provided in `validation_tools.py`.

8 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 extent 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.

8.1 Adding pre-processing functions

Pre-processing functions are defined in *data_preprocessing.py* in the *fluxengine_src* directory. When parsing `_preprocessing` entries 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`.
- `DataLayer`s 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 cased `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.

8.2 Adding k-parametrisation functors

The gas transfer velocity calculation is fully customisable by adding a 'functor' class to the `rate_parameterisation.py` file. Functors are classes which are callable. This class must be derived from the `KCalculationBase` class 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 `DataLayer`s. Any non-existing `DataLayer`s 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 `DataLayer`s which are not listed by name in the list returned by `output_names`.

It is best practice to modify the 1D `DataLayer.fdata` attribute of output `DataLayers`. 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 `rate_parameterisation.py` in the `fluxengine_src` directory.

8.2.1 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. You can specify the type of input which is required, or add constraints and/or metadata associated with a configuration variables by adding an entry to `settings.xml` (found in in the `fluxengine_src` directory). *Modifying entries for pre-existing variables is not recommended.*

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`).

8.3 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 pull requests. 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`.