MESAS

Release alpha

Ciaran J. Harman

Apr 19, 2023

GETTING STARTED

Multiresolution Estimation of StorAge Selection functions

CHAPTER

ONE

WHAT ARE STORAGE SELECTION FUNCTIONS?

StorAge Selection is a theoretical framework for modeling transport through control volumes. It is appropriate if you are interested a system that can be treated as a single control volume (or a collection of such), and wish to make minimal assumptions about the internal organization of the transport. SAS assumes that the material leaving a system is some combination of the material that entered at earlier times. This can be useful for constructing simple models of very complicated flow systems, and for inferring the emergent transport properties of a system from tracer data.

SAS is a very general framework – at heart, it is composed of two parts: a way of keeping track of changes in the age structure of the 'population' of fluid parcels in a system, and a rule (or rules) that determines how that age structure changes. By using tracers to investigate what rule best mimics the behaviour of a given system, we can learn important information about that system – most directly, something about the volume of tracer-transporting fluid that is actively 'turning over'. SAS is a generalization of commonly-used transit time theory to the case where fluxes vary in time, and there may be more than one outflow (e.g. discharge and ET).

For more information see the free HydroLearn course: Tracers and transit times in time-variable hydrologic systems: A gentle introduction to the StorAge Selection (SAS) approach

CHAPTER

TWO

WHAT IS MESAS.PY?

The package will eventually include a number of component tools for building SAS models and using them to analyze tracer data.

Currently the main repository codebase includes three submodules:

mesas.sas
: The main submodule for building and running SAS models

mesas.utils
: Tools for manipulating SAS model input data and results

mesas.utils.vis
: Visualization of SAS model results

I hope to expand these in the future, and welcome contributors through the GitHub repository

CHAPTER

THREE

SUPPORT, BUGS, SUGGESTIONS, ETC.

If you find a bug please open an issue on GitHub here: https://github.com/charman2/mesas/issues

You can also use the issue tracer to make suggestions to improve the code.

If you need further help using the code or interpreting the results, please get in touch: charman1 at jhu dot edu

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 Installation

The current version of mesas.py can be installed using Conda with:

```
conda install -c conda-forge mesas
```

This will install any additional dependencies at the same time.

Alternatively, the code can be obtained from GitHub: https://github.com/charman2/mesas. Note that a fortran compiler is required to build from source (but is not required to install through Conda).

Clone the git repo and open a command prompt in the mesas directory (where the setup.py file sits). Make and install with:

```
python setup.py install
```

4.1.1 Build from source

Building MESAS from source is a two step process: (1) compile the *cdflib90* Fortran source code, and (2) install the *mesas* python package.

Compile *cdflib90*

To compile the *cdflib90* source code (located under `mesas/sas/cdflib90/`) you will need a Fortran compiler and *cmake*. Both of which can be installed through *conda*,

```
$ conda install fortran-compiler cmake -c conda-forge
```

To compile *cdflib90*,

- create a build directory into which library and module files will be placed
- run *cmake* to setup the build folder
- run *cmake* to compile the code

```
$ mkdir mesas/sas/cdflib90/_build
$ cmake -S mesas/sas/cdflib90 -B mesas/sas/cdflib90/_build
$ cmake --build mesas/sas/cdflib90/_build
```

Note: The location of the build folder (`mesas/sas/cdflib90/_build/`) is important. The location of this folder MUST match the value that is in *setup.py*.

Build and install *mesas*

With *cdflib90* built, *mesas* can be *pip*-installed in the usual way,

```
$ pip install -e .
```

4.2 Quickstart

To run a mesas.py model, you must supply four things.

The first is a timeseries of the system inputs and outputs, and of any time-varying parameters, stored in either

- a Pandas dataframe
- a `.csv` file

The other three are sets of parameters and options to configure the model, which can be supplied either as text files (in the human-readable JSON format) or as python dictionaries (`dict` objects). The three sets are

- A specification of the SAS function
- A specification of any solute solute_parameters
- Any optional parameters and settings

If you are unfamiliar with `pandas` and python `dict` objects (or even if you are) it is probably easiest to start by using `.csv` and `.json` input files.

4.2.1 Input timeseries

The `.csv` of timeseries data should contain (at minimum) the timeseries of fluid input, output, and any tracer input timeseries. In addition, many SAS function and solute parameters can be time-varying (rather than fixed values). When they vary in time, they must be provided as a column in the dataframe.

The code below creates a `.csv` file with some artificial data. We will construct it to be 100 timesteps of steady flow at rate `Q_steady` through storage volume `Storage_vol`. The inputs will be labeled with tracer at concentration `C_tracer_input` for 10% of the total duration near the start of the timeseries. Let's start by creating some variables to hold this information:

```
timeseries_duration = 1.
timeseries_length = 100
dt = timeseries_duration/timeseries_length

pulse_start = 0.05
pulse_end = 0.15

C_tracer_input = 0.5
Q_steady = 1.
Storage_vol = 0.1
```

Now we will create the dataframe in pandas:

```
import pandas as pd
import numpy as np
data_df = pd.DataFrame(index=np.arange(timeseries_length) * dt)
data_df['Q out [vol/time]'] = Q_steady
data_df['J in [vol/time]'] = Q_steady
data_df['C [conc]'] = 0
data_df.loc[pulse_start:pulse_end, 'C [conc]'] = C_tracer_input
data_df.to_csv('data.csv')
```

This creates a file `data.csv` that contains the input timeseries.

4.2.2 Configuration information

Next, let's give the specifications of the SAS function, solute parameters, and options in a file `config.json`. The part specifying the SAS function looks like this:

```
"sas_specs": {
    "Q out [vol/time]": {
        "my first SAS func!": {
            "ST": [0, "Storage_vol"]
            }
        }
    }
```

This specification says the following:

- There is only one flux out, and it can be found under the column `"Q out [vol/time]"` in the dataframe.
- There is only one SAS function associated with this flux, and it is called `"my first SAS func!"`.
- The SAS function is specified as a piecewise linear function with one linear segment from $P = 0$ at $ST = 0$ to $P = 1$ at $ST =$ `Storage_vol`.

The only solute information we need to give is the name of the column containing the input timeseries. The part of the `config.json` file providing this information looks like this:

```
"solute_parameters": {
    "C [conc]": {}
    }
```

The dictionary associated with our solute is empty `{}`, so the default parameters will be used.

The part providing additional options looks like this:

```
"options":{
    "dt": 0.01,
    "influx": "J in [vol/time]"
    }
```

This specifies

- The timestep of the model `dt` (which depends on the units of the inflow and outflow)
- The name of the column in the dataframe that contains the inflow rate, given by the keyword argument `influx`

The complete `config.json` file should look like this:

```
{

"sas_specs": {
    "Q out [vol/time]": {
        "my first SAS func!": {
            "ST": [0, "Storage_vol"]
            }
        }
    },

"solute_parameters": {
    "C [conc]": {}
    },

"options":{
    "dt": 0.01,
    "influx": "J in [vol/time]"
    }

}
```

4.2.3 Running mesas.py

Now we are ready to import mesas.py, create the model, and run it:

```
from mesas.sas.model import Model
model = Model(data_df='data.csv', config='config.json')
model.run()
model.data_df.to_csv('data_with_results.csv')
```

Assuming the model runs without incident the predicted discharge concentration has appeared as a new column in the file `data_with_results.csv`. The columns generated by the model will have the form `'<solute column name> --> <flux column name>'`.

4.2.4 Plot the results

The results can be accessed within python as the `pandas` dataframe `model.data_df`

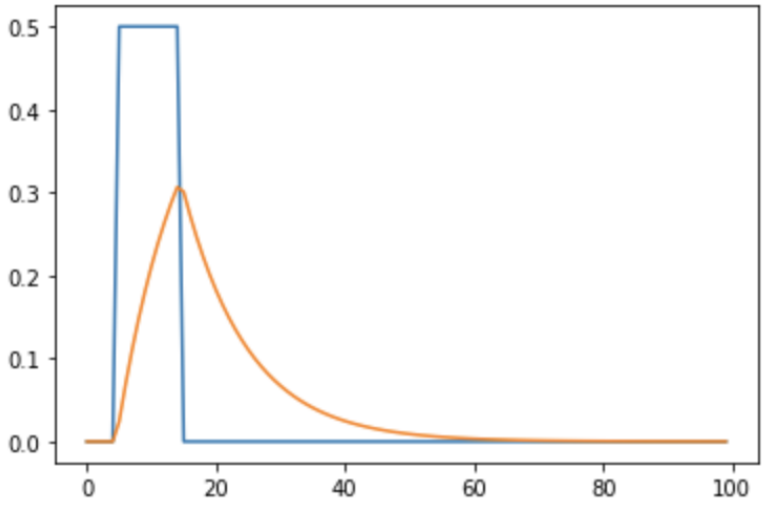
We can use matplotlib to plot individual columns of the dataframe like this:

```
import matplotlib.pyplot as plt
plt.plot(model.data_df.index, model.ata_df['C [conc]'])
plt.plot(model.data_df.index, model.data_df['C [conc] --> Q out [vol/time]'])
```

Which should give this:

[12]:
```
import matplotlib.pyplot as plt
plt.plot(data_df.index, data_df['C [conc]'])
plt.plot(data_df.index, data_df['C [conc] --> Q out [vol/time]'])
```

[12]:
```
[<matplotlib.lines.Line2D at 0x1a1e348c50>]
```



4.3 Timeseries Inputs and Outputs

A single Pandas dataframe `data_df` is used to store all the input timeseries needed by the model. These data can be provided to the model as a path to a `.csv` file that will be automatically imported and stored as a dataframe.

The dataframe or `.csv` should be constructed before creating the model object, and then included in the model initialization. The following are equivalent:

```
import pandas as pd
from mesas.sas import Model

# read input timeseries from a .csv file into a dataframe
my_dataframe = pd.read_csv('my_input_timeseries.csv', ...)

# create model
my_model = Model(data_df=my_dataframe, ...)
```

and

```
from mesas.sas import Model

# create model
my_model = Model(data_df="path/to/my_data.csv", ...)
```

Model outputs are appended to the copy of the dataframe stored in `my_model`. To access the model version call the property directly:

```
t = my_model.data_df.index
y = my_model.data_df['some column name']
plt.plot(t,y)
```

This version can also be modified in-situ, though the model must be re-run to generate corresponding results:

```
# Generate results with current inputs
my_model.run()
# Modify an input
my_model.data_df['input variable'] = new_version_of_input_variable
# Re-run model to update results
my_model.run()
```

4.3.1 Input and output fluxes

To run the model the timeseries dataframe must contain one input flux timeseries, and at least one output flux. These can be specified in any units, but should be consistent with one another, and the flux values multiplied by the value in `my_model.options['dt']` (see *Optional parameters*) should equal the total input and output mass of fluid (i.e. water) in each timestep. The timesteps as assumed to be all of equal interval, and equal length, and should contain no `NaN` values.

A simple steady flow model can be constructed by creating timeseries with constant values:

```
import pandas as pd
import numpy as np

N = 1000 # number of timesteps to run
J = 2.5 # inflow rate
Q = J # steady flow so input=output

my_dataframe = pd.DataFrame(index = np.arange(N))

my_dataframe['J'] = J
my_dataframe['Q'] = Q

...

my_model = Model(data_df=my_dataframe, ...)

my_model.run()
```

The name of the column that contains the inputs is `'J'` by default, but can be modified by changing the `'influx'` option (see *Optional parameters*). The name of the column that contains each flux is specified in the `sas_specs` input dictionary (see *Specifying SAS functions*)

4.3.2 Other input timeseries

The timeseries dataframe also stores timeseries used in the specification of SAS functions (see *Specifying SAS functions*) and solutes (see *Specifying solute parameters*). The column names specified in the `sas_specs` and `solute_parameters` inputs must exactly match a column in the `data_df` dataframe.

Here is a minimal example with steady inflow, time-variable discharge according to a linear storage-discharge relationship, uniform sampling, and a pulse of tracer input at a timestep some short time after the storage begins to fill. Note that the total storage `S` is stored in a column of the dataframe named `'S'`, which is used in the specification of the uniform SAS function in `my_sas_spec`. Similarly, the concentration timeseries is stored in a column of the dataframe named `'Cin'`, which corresponds to a top-level key in `my_solute_parameters`.

```
import pandas as pd
import numpy as np
from mesas.sas.model import Model

N = 1000 # number of timesteps to run
t = np.arange(N)

J = 2.5 # inflow rate
k = 1/20 # hydraulic response rate
Q = J * (1 - np.exp(-k * t))
S = Q / k
S[0] = S[1]/1000

Cin = np.zeros(N)
Cin[10] = 100./J

my_dataframe = pd.DataFrame(index = t, data={'J':J, 'Q':Q, 'S':S, 'Cin':Cin})

my_sas_specs = {
    'Q':{
        'a uniform distribution over total storage':{
            'ST': [0, 'S']
            }
        }
    }

my_solute_parameters = {
    'Cin': {}
    }

config = {
    'sas_specs': my_sas_specs,
    'solute_parameters': my_solute_parameters
    }

my_model = Model(data_df=my_dataframe, config=config)

my_model.run()
```

4.3.3 Output timeseries

If a timeseries of solute input concentrations is provided and its name appears as a top-level key in the `solute_parameters` dictionary, timeseries of output concentrations will be generated for each output flux specified in the `sas_specs`.

The predicted outflow concentration timeseries will appear as a new column in the dataframe with the name `'<solute column name> --> <flux column name>'`. For example, the outflow concentrations in the simple model given above will appear in the column `'Cin --> Q'`.

To save the model outputs as a csv use:

```
my_model.data_df.to_csv('outputs.csv')
```

4.4 Configuring the model

To run *mesas.py* the user must provide some information about how the model is to be configured. First, at least one SAS function must be specified. In addition you may wish to provide information about solutes to be transported through the system, and/or modify the default model options.

The recommended way of providing configuration data is in a text file formatted using the JSON standard. Typically this file is called something like `config.json`. The contents look very much like a python `dict`, with some differences. A minimal example of a configuration file is a text file containing the following text:

```
{
"sas_specs":{
    "Flux out":{
        "SAS fun":{
            "ST": [0, 100]
            }
        }
    }
}
```

This specifies a single uniform SAS function between 0 and 100 associated with the flux named `"Flux out"`.

mesas.py looks for three top-level entries in the file:

- `"sas_specs"`: spcification of the SAS functions (see *Specifying SAS functions*)
- `"solute_parameters"`: information about solutes to be transported through the system (see *Specifying solute parameters*)
- `"options"`: optional arguments for customizing the model run (see *Optional parameters*)

For more information click the links above. Here is an example of the contents of a `config.json` file that uses all three of these top-level fields.

```
{
"sas_specs":{
    "Flux out":{
        "SAS fun":{
            "ST": [0, 100]
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }

"solute_parameters":{
    "solute A":{
        "C_old": 0.5
        }
    }

"options":{
    "dt": 3600
    }
}
```

4.5 Specifying SAS functions

SAS functions are specified using a nested dictionary-like structure, stored as either python code or in a `config.json` file (see *Configuring the model*). A simple JSON specification looks like this:

```
{
"sas_specs":{
    "Flux out":{
        "SAS fun":{
            "ST": [0, 100]
            }
        }
    },

"...": "..."

}
```

The `"...": "..."` in the example above should not be included in a real file. There are used here to show where additional information *Specifying solute parameters* and setting *Optional parameters* may be included.

The equivalent python dictionary looks like this:

```
my_sas_spec = {
    "Flux out":{
        "SAS fun":{
            "ST": [0, 100]
            }
        }
    }
```

The levels of the `sas_specs` nested dictionary / JSON entry are as follows:

Level 1: One `key:value` pair per outflow flux (e.g. discharge, ET, groudwater discharge, etc.)
Each key is a string that corresponds to a column in the input timeseries dataset `data_df` that contains the corresponding outflow flux rates. To specify multiple outflows, just add additional `key:value` pairs to this level of the dictionary.

Level 2: One `key:value` pair per component SAS function to be combined in a time-variable weighted sum
If there is more than one key in a dict at this level, the model will assume that the dictionary values are specifica-

tions for SAS functions, and the keys correspond to columns in `data_df` with the time-varying weight for that component. This can be a useful way to specify time-varying SAS functions, but the individual SAS functions can themselves be time-varying.

Level 3: `key:value` pairs giving properties of the SAS function
Where properties are given as strings rather than numbers it will be assumed (with a few exceptions) that the associated values are time-varying and are given by the corresponding column in `data_df`.

There are four ways to specify a SAS function in the level 3 dict:

- As an exact gamma or beta distribution
- Using any distribution from `scipy.stats` (which will be converted into a piecewise linear approximation in the model)
- As a piecewise linear CDF
- As a B-Spline PDF (not yet implemented)

4.5.1 Using a gamma or beta distribution

The gamma and beta distributions are often used to model SAS functions due to their flexibility. An important difference between them is that the beta distribution only has non-zero probability over a finite interval, while the gamma distribution is defined for infinitely large values. The beta distribution is useful when the total storage volume can be reliably inferred from tracer data. The gamma may be preferable where a large portion of the storage volume turns over very slowly, and so its volume is difficult to constrain from the tracer data.

The PDF of a beta distribution is given by:

$$f(x) = x^{\alpha-1}(1-x)^{\beta-1}N(\alpha,\beta)$$

where α and β are parameters and $N(\alpha,\beta)$ normalizes the distribution to have unit area. The gamma distribution is given by:

$$f(x) = x^{\alpha-1}e^{-x}N(\alpha)$$

which has only one shape parameter, α, and again $N(\alpha,\beta)$ normalizes the distribution. These distributions can be used flexibly by converting the input values of S_T into a normalized form x:

$$x(T,t) = \frac{S_T(T,t) - S_{\mathrm{loc}}(t)}{S_{\mathrm{scale}}(t)}$$

where

- $S_{\mathrm{loc}}(t)$ or `"loc"` : the location parameter, which shifts the distribution to the right for values >0 and to the left for values <0 (default is 0)
- $S_{\mathrm{scale}}(t)$ or `"scale"` : the scale parameter (default is 1)

The desired distribution is specified using the key `"func"`, and the associated parameters using the keyword `"args"`, as illustrated in the example below. All parameters can be made time-varying by setting them to a string corresponding to a column in `data_df`.

Example

Here is an examples of a SAS specification for two fluxes, "Discharge" and "ET", whose SAS function will be modeled as a gamma and beta distribution respectively. The scale parameter of the discharge gamma distribution is set to "S0" indicating that its values can be found in that column of the data_df dataframe.

```
{
"sas_specss":{
    "Discharge": {
        "Discharge SAS fun": {
            "func": "gamma",
            "args": {
                "a": 0.62,
                "scale": "S0",
                "loc": 0.
                }
            }
        },
    "ET": {
        "ET SAS fun": {
            "func": "beta",
            "args": {
                "a": 2.31,
                "b": 0.627,
                "scale": 1402,
                "loc": 248
                }
            }
        }
    }

"...": "..."

}
```

In this case the model will look for columns in data_df called "Discharge" and "ET", and assume the values in these columns are timeseries of outflows from the control volume. Note that the values in these columns must be in the same units.

The "Discharge" flux has a single component SAS function named "Discharge SAS fun". Since there is only one component SAS function for the "Discharge" flux there does not need to be a column in the dataframe called "Discharge SAS fun". We specify the SAS function as a gamma distribution with the key:value pair "scipy.stats": gamma. The distribution properties are set in the dictionary labeled "args". The gamma distribution with shape parameter "a" which is here set to 0.62.

The "ET" flux has a SAS function named "ET SAS fun". This is specified to be a beta distribution, which has two shape parameters: "a" and "b". As before, these are set in the "args" dictionary, along with the scale and shape parameters.

4.5.2 Using parameterized distributions from scipy.stats

`Scipy.stats` provides a large library of probability distributions that can be used to specify a SAS function. Note that only continuous distributions with non-negative support are valid SAS functions (though the support need not be finite).

The continuous distribution is converted into a piecewise linear approximation, which is then passed into the core number-crunching part of the code. This is done because evaluating the native `scipy.stats` functions was found to be too computationally expensive.

To use them, the distributions are specified using the same format as above, but with the additional key `"use": "scipy.stats"`. The Level 3 dictionary in this case should therefore have four key:value pairs

- `"func"` : <a string giving the name of a `scipy.stats` distribution>
- `"use"` : `"scipy.stats"`
- `"args"` : <a dict of parameters to be passed to the distribution
- `"nsegment"` : <an integer giving the number of segments to use in the piecewise linear approximation (optional, default is 25)>

The dict associated with `"args"` specifies parameters for the associated distribution. These can be given as a number, or as a string that refers to a column in `data_df`.

Each function in `scipy.stats` requires at least two parameters:

- `"loc"` : the location parameter, which shifts the distribution to the right for values >0 and to the left for values <0 (default is 0)
- `"scale"` : the scale parameter (default is 1)

These two parameters are used to convert the input values of S_T into a normalized form x:

$$x(T,t) = \frac{S_T(T,t) - S_{\mathrm{loc}}(t)}{S_{\mathrm{scale}}(t)}$$

Additional parameters are needed for a subset of functions (see the `scipy.stats` documentation). For example, the gamma distribution requires a shape parameter `"a"`, and the beta distribution requires two parameters `"a"` and `"b"`.

Example

Here is an examples of a SAS specification for two fluxes, `"Discharge"` and `"ET"`, whose SAS function will be modeled as a gamma and beta distribution respectively. These will be converted into piecewise linear approximations with 50 segments.

```
{
"sas_specss": {
    "Discharge": {
        "Discharge SAS fun": {
            "func": "gamma",
            "use": "scipy.stats",
            "args": {
                "a": 0.62,
                "scale": 5724.,
                "loc": 0.
                },
            "nsegment": 50
            }
```

(continues on next page)

(continued from previous page)

```
        },
    "ET": {
        "ET SAS fun": {
            "func": "beta",
            "use": "scipy.stats",
            "args": {
                "a": 2.31,
                "b": 0.627,
                "scale": 1402,
                "loc": 248
                },
            "nsegment": 50
            }
        }
    }

"...": "..."

}
```

4.5.3 As a piecewise linear CDF

A SAS function can be specified by supplying the breakpoints of a piecewise linear cumulative distribution (i.e. a piecewise constant PDF).

At minimum, the values of S_T (corresponding to breakpoints in the piecewise linear approximation) must be supplied. These are given by the `"ST"` key, which must be associated with a list of strictly-increasing non-negative values. Non-increasing or negative values in this list will result in an error. The first value does not need to be zero. The values can be given as a fixed number, or as a string referring to a column in `data_df`.

Values of the associated cumulative probability can optionally be supplied with the key `"P"`, which must be associated with a list of strictly-increasing numbers between 0 and 1 of the same length as the list in `"ST"`. The first entry must be `` `0 `` and the last must be `1`. Again, the values can be given as a fixed number, or as a string referring to a column in `data_df`. If `"P"` is not supplied it will be assumed that each increment of `"ST"` represents an equal increment of probability.

Example

Here is an example, where storage is given in units of millimeters:

```
{
"sas_specss": {
    "Discharge": {
        "Discharge SAS fun": {
            "ST": [0, 553, "Total Storage"]
            "P" : [ 0, 0.8, 1.]
            }
        },
    "ET": {
        "ET SAS fun": {
            "ST": [50, 250, 800]
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }

"...": "..."

}
```

This specifies that for `"Discharge"` 80% of the discharge should be uniformly selected from the youngest 553 mm, and the remaining 20% from between 553 mm and the (presumably time-varying) value given in `data_df["Total Storage"]`.

For "ET", only the "ST" values are provided, so mesas.py will assume the "P" values are uniformly spaced from 0 to 1. Here no ET will be drawn from the youngest 50 mm of storage, 50% will be drawn from between 50 and 250 mm, and 50% will be drawn from between 250 mm and 800 mm.

4.6 Specifying solute parameters

Solute parameters are also given using a nested set of python dictionaries or a JSON entry in a `config.json` file. These are usually set when the model object is created. See below for how to change solute parameters after model creation.

Keys of the highest level of the dictionary are assumed to be the `data_df` column names for the input concentrations of each solute. The value associated with each key should be a (possibly empty) dictionary of parameters. For example:

```
{

"...": "...",

"solute_parameters": {
    "C1 [per mil]": {},
    "C2 [stone/bushel]": {}
    }
}
```

The `"...": "..."` in the example above should not be included in a real file. There are used here to show where additional information *Specifying SAS functions* and setting *Optional parameters* may be included.

The equivalent python dictionary is:

```
my_solute_parameters = {
    "C1 [per mil]": {},
    "C2 [stone/bushel]": {}
    }
```

In this case *mesas.py* would expect to find columns `"C1 [per mil]"` and `"C2 [stone/bushel]"` in the *timeseries input data* that contain the input concentrations of two solutes. Since an empty dict `{}` is passed in each case, default parameters (representing a conservative ideal tracer initially absent from the system) will be used.

The parameter dictionary may specify any of the following default `key:value` pairs. If a `key:value` pair does not appear in the dictionary, the default value will be used.

`mT_init` (array-like, default = [0.0, 0.0, ...])
Initial age-ranked mass in the system. This is useful if the system is initialized by some sort of spin-up. Each entry

is age-ranked mass in an age interval of duration Δt. If `mT_init` is specified, `sT_init` must also be specified in *Optional parameters*, and be of the same length. The element-wise ratio `mT_init/sT_init` gives the age-ranked concentration `CS` of the water in storage at time zero. Note that if `sT_init` is specified but `mT_init` is not, the concentrations associated with each non-zero value of `sT_init` will be zero.

`C_old` (float, default = 0.0)
Old water concentration. This will be the concentration of all water released of unknown age. If `sT_init` is not specified, this will be all water in storage at `t=0`. If `sT_init` is specified, it will be all water older than the last non-zero entry in `sT_init`.

`k1` (float or string, default = 0.0)
First-order reaction rate constant. The rate of production/consumption of solute s is modeled as:

$$\dot{m}_R^s(T,t) = k_1^s(t)(C_{eq}^s(t)s_T(t,T) - m_T(t,T))$$

where $m_R^s(T,t)$ is the rate of change of mass of solute s in storage. Note that the reaction rate has units of [1/time], and should be in the same time units as the fluxes. The reaction rate can be made time-varying by associating `k1` with a string referring to a column of reaction rates in `data_df`.

`C_eq` (float or string, default = 0.0)
Equilibrium concentration. See above for role in modeling first order reactions. Note the equilibrium concentration can be made time-varying by associating `C_eq` with a string referring to a column of equilibrium concebntrations in `data_df`.

`alpha` (dict, default = 1.0)
A dict giving partitioning coefficients for each outflow. The rate of export $\dot{m}_q^s(T,t)$ of solute s through outflow q is:

$$\dot{m}_q^s(T,t) = \alpha_q^s(t)\frac{m_T^s(T,t)}{s_T(t,T)}p_q(T,t)Q_q(t)$$

Thus if $\alpha_q^s = 1$ the outflow concentration of water of a given age will equal that in storage. If $\alpha_q^s = 0$, the solute will not be exported with outflow q. Values of $0 < \alpha_q^s < 1$ will result in partial exclusion of the solute from the outflow, and $\alpha_q^s > 1$ will result in preferential removal via outflow outflow q.

The keys in the `alpha` dict must match keys in top level keys of `sas_specs`. Each key may be associated with a number or a string referring to a column of partitioning coefficients in `data_df`. {"Q": 1., ...} # Partitioning coefficient for flux "Q"

`observations` (dict, default = None)
This dict provides the name of columns in *data_df* that contain observations that may be used to calibrate/validate the model"s predictions of outflow concentrations. Keys are outflow fluxes named in top level keys of `sas_specs`, e.g. `"observations":{"Q": "obs C in Q", ...}`.

4.6.1 Modifying parameters

There are two equivalent ways to modify the parameters of an existing model.

Assigning a dict
The model property `<my_model>.solute_parameters` can be assigned a dict of valid key-value pairs. This will overwrite existing parameters for all the properties in the dict, but leave the remainder unchanged.

To remove all solute parameters (so no solute transport will be modelled) set `<my_model>.solute_parameters=None`. Default parameters can then be set by assigning an empty dict to each solute `<my_model>.solute_parameters={"C1":{}, ...}`

Using the `<my_model>.set_solute_parameters()` function
Individual properties of a solute can be set using this convenience function. Individual parameters are set as keyword arguments, like this:

```
<my_model>.set_solute_parameters("C1", C_old=22.5)
```

This would set the `C_old` property associated with solute `"C1"` to `22.5`.

4.7 Optional parameters

A number of optional parameters can be set. These can be specified in the `config.json` file under the `"options"` key:

```
{

"...": "...",

"options":{
    "dt":{
        "dt": 3600,
        "n_substeps": 5
        }
    }
}
```

Options can be set as keywords when the instance is created:

```
my_model = Model(..., option1='value', option2='another')
```

or by assigning a `dict` of valid key-value pairs:

```
my_model.options = {option1='value', option2='another'}
```

In the latter case, only the options specifically referenced will be changed. The others will retain their previous values (i.e. the defaults unless they have been previously changed).

Default options are

`dt`: (scalar, default=1.0)
Timestep in appropriate units, such that `dt` multiplied by any of the fluxes in `data_df` gives the total volume of flux over the timestep

`verbose`: (bool, default=False)
Print information about the calculation progress

`debug`: (bool, default=False)
Print information useful for debugging. Warning: do not use.

`warning`: (bool, default=True)
Turn off and on warnings about calculation issues

`n_substeps`: (int, default=1)
Number of substeps in the calculation. Each timestep can be subdivided to increase the numerical accuracy of the solution and address some numerical issues, at the cost of longer run times. Note that the substep calculations are not retained in the output. The substeps are aggregated back to the full timestep first

`max_age`: (int, default=len(data_df))
The maximum age that will be calculated. This controls the number of rows in the output matricies. Set to a smaller value than the default to reduce calculation time (at the cost of replacing calculated concentrations of old water with `C_old`)

sT_init: (1-D numpy.array, default=np.zeros(len(data_df)))
Initial distribution of age-ranked storage (in density form). Useful for starting a run using output from another model run, e.g. for spin up. If the length of this array is less than the length of `data_df`, then `max_age` will be set to `len(sT_init)`

influx: (str, default=``J``)
The name of the column in `data_df` containing the inflow rate

record_state: (str, default=False)
Record the state variables at some or all timesteps. Default value of `False` will still record the final state of the model. Note that setting this to `True` can greatly increase the memory requirements for large datasets. Can also be set to a string representing a column in `data_df` of booleans.

4.8 Extracting results

Once the model has been run, timeseries of solute outputs can be found in `<Model object>.data_df`. To save these as a `.csv` use:

```
my_model.data_df.to_csv('my_model_outputs.csv')
```

where `my_model` is a Model object.

Alternatively, raw results in the form of output arrays can be accessed through the `<Model object>.results` property, which returns a dict with the following keys:

sT
[m x n+1 numpy float64 2D array] Array of instantaneous age-ranked storage for n+1 times, m ages. First column is initial condition (given by the `sT_init` option if provided, or all zeros otherwise)

pQ
[m x n x q numpy float64 2D array] Array of timestep-averaged time-varying backward transit time distributions over m ages, at n times, for q fluxes.

WaterBalance
[m x n numpy float64 2D array] Should always be within tolerances of zero, unless something is very wrong.

C_Q
[n x q x s float64 ndarray] If input concetrations are provided (see *Specifying solute parameters*), this gives the timeseries of timestep-averaged outflow concentration

mT
[m x n+1 x s float64 ndarray] Array of instantaneous age-ranked solute mass over m ages, at n times, for s solutes. First column is initial condition

mQ
[m x n x q x s float64 ndarray] Array of timestep-averaged age-ranked solute mass flux over m ages, at n times, for q fluxes and s solutes.

mR
[m x n x s float64 ndarray] Array of timestep-averaged age-ranked solute reaction flux over m ages, at n times, for s solutes.

SoluteBalance
[m x n x s float64 ndarray] Should always be within tolerances of zero, unless something is very wrong.

The order that the fluxes `q` and solutes `s` appear in these arrays is given by the properties `my_model.fluxorder` and `my_model.solorder`. These provide lists of the column names in the order they are given in `results`.

4.9 Visualizing outputs

Under construction…

4.10 SAS Functions

Functions will be documented here. Some of this is out of date.

4.10.1 mesas.sas.functions module

4.11 SAS Specification Class

A SAS specification is an object that can be queried to get the sas function at each timestep. This allows us to specify a time-varying sas function in a couple of different ways.

The `Fixed` spec is the simplest, producing a time-invariant function

The `StateSwitch` spec switches between several sas functions, depending on the state of the system at a given timestep

The `Weighted` spec produces a weighted average of several sas functions. The weights can vary in time

4.11.1 mesas.sas.specs module

4.12 SAS Model class

Model will be documented here.

4.12.1 mesas.sas.model module

4.13 Indices and tables

- genindex
- modindex
- search