Suberbox manual and developer guidelines

[Work under construction, updated as the work progresses]

Table of Contents

1.1 Precision

All reals are double precision, use `real(dp)`; dp is declared in `second_precision` and should be used globally. Always `USE SECOND_PRECISION` (the module comes later from KPP and will always be present).

1.2 Derived types

1.2.1.1 type timetype (in constants.f90)

For all time-related variables, a single variable of `TYPE(timetype)` is used. The type definition is currently in `constants.f90`. It currently holds the following information, and more will be added:

Component name	Component type	Default value	automatically updated?	Description
SIM_TIME_H	real(dp)	24d0	no	Total simulation time in hours; input from user
SIM_TIME_S	real(dp)	86400d0	only upon declaration	Total simulation time in hours; calculated from above
dt	real(dp)	10.0d0	no	Time step of main model in seconds
sec	real(dp)	0	no	Current time in seconds
min	real(dp)	0	yes	model time in minutes (and decimal fractions)
hrs	real(dp)	0	yes	model time in hours (and decimal fractions)
day	real(dp)	0	yes	model time in days (and decimal fractions)
dt_chem	real(dp)	10.0d0	no	Time step of chemical model in seconds
dt_aero	real(dp)	10.0d0	no	Time step of aerosol model in seconds
ind_netcdf	integer	1	yes	index of netcdf-file row (in effect the row number)
JD	integer	0	no	Julian day of year at the start of the run
PRINT_INTERVAL	integer	15	no	interval of screen output in model minutes
FSAVE_INTERVAL	integer	5	no	interval for saving all values in model minutes
hms	character(8)	"00:00:00"	yes	Character string for showing time (e.g. error messages)
printnow	logical	.true.	yes	TRUE if time to print screen output (automatic)
savenow	logical	.true.	yes	TRUE if time to saving values
PRINTACDC	logical	.false.	no	TRUE if cluster/monomer fractions are printed

Function `ADD(time, sec)`, also in `constants.f90`, will forward the `time%sec` by `sec` seconds. Alternatively, `ADD(time)` will forward the `time%sec` by `time%dt`. Also, operator + ("plus") is assigned to `timetype`, so that `time+sec` will have same effect than `ADD(time, sec)`.

Every module working with time should `USE CONSTANTS`.

1.2.1.2 type parametered_input (in constants.f90)

The purpose of this type is to modify the value of any input. It wraps the necessary parameters for the function `NORMALD`, and additionally it has two operators assigned to it, + for adding (or subtracting) a

constant in `parametered_input%min_c`, and * for multiplying the value by `parametered_input%amplitude`. When `parametered_input` is used with + or *, only the respective variables are used, and none of the others have any influence.

Effect of different operators and `parametered_input`:

`real(dp) + parametered_input` returns `real(dp) + parametered_input%min_c`

`real(dp) * parametered_input` returns `real(dp) * parametered_input%amplitude`

The `parametered_input` type holds the following variables:

	Component name	Component type	Default value	Description
Pure Gaussian	`min_c`	real(dp)	0d0	Minimum value for the function; background level. Also used as offset if parametered_input is used with operator +
	`max_c`	real(dp)	1d5	Maximum of the peak.
	`width`	real(dp)	1d0	Width of the bell curve; sigma of the gaussian function
	`peaktime`	real(dp)	12d0	Time of the peak in hours; exact if modulation is not used, otherwise approximate.
Modulation function	`omega`	real(dp)	0d0	Angular frequency of the modulation
	`phase`	real(dp)	0d0	Phase shift of the modulation
	`amplitude`	real(dp)	1d0	Amplitude of the modulation function. Also used as multiplication factor if parametered_input is used with operator *
Common	`LOGSCALE`	LOGICAL	.true.	use linear or logical scale for values

The functions that are assigned to + and * operators are `PLUS(c, MODS)` and `MULTIPLICATION(c, MODS)`, respectively, found in constants.f90.

For consecutive operation `real * parametered_input + parametered_input` can be used (but only in this order), also for convenience an operator .mod. has been assigned to such operation, so `real .mod. parametered_input` has the same effect; first multiplying with `amplitude`, then adding a constant `min_c`.

```
real(dp) :: a = 4.0d0
type(parametered_input) :: b
b%amplitude = 3d0
b%min_c = 4.2d0
a = a .mod. b
print*, a
```

```
>>> 16.199999999999999
```

`parametered_input` (or an element of a vector of them) can be used to modify any variable in the program, including multiplying values, or setting them to zero or some other constant, using just one variable. Naturally, if `parametered_input%min_c == 0` and `parametered_input%amplitude == 1`, such operation keeps the original value intact. These are the default values of a declared `parametered_input` variable.

1.2.1.3 Other derived types (in constants.f90)

These include the types that are directly imported from old UHMA, and are most certainly going to see many changes.

1.3 Useful functions

All the functions under chapter 1.3 are in auxillaries.f90

1.3.1.1 real(dp) function NORMALD(TIME, MODS)

time: current `type(timetype)`
MODS: `type(parametered_input)`

The function is useful if one wants to create artificial concentrations, rates or whatever value is needed, which has a peaked form. It creates a Gaussian bell curve, and also optionally modifies it with sine function (see code for details). `NORMALD` uses `MODS` to return a value at `TIME`. Assumed time unit is hours, so basically it is possible to create a function just by estimating the correct numbers. In practice, it is best to use the helper program ParameterTweaker.py to create parameters for artificial functions. Also see above for description of `parametered_input`.

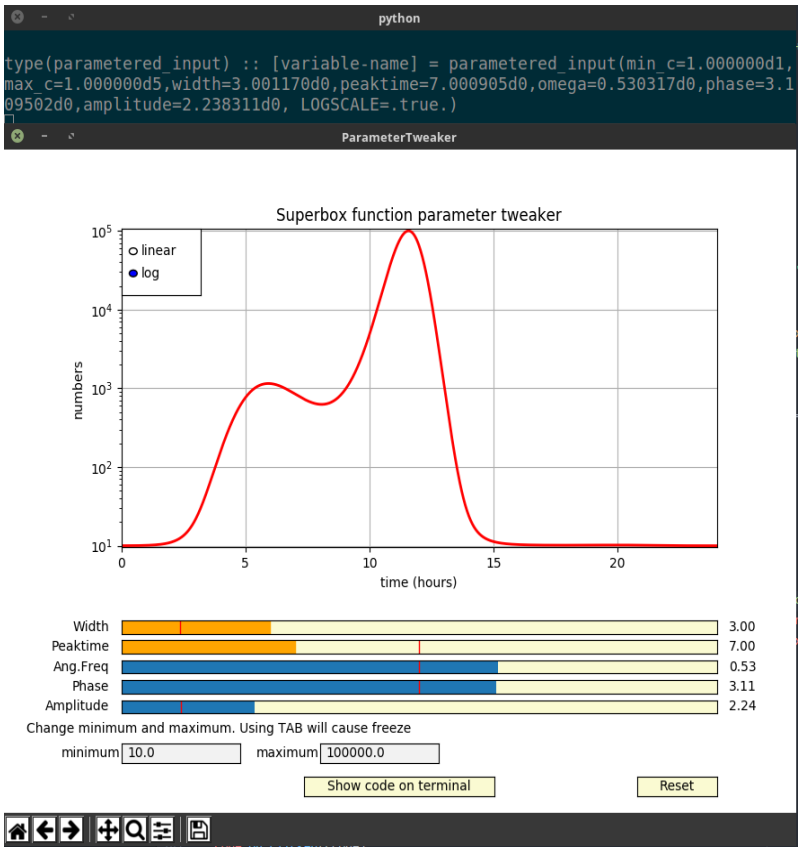

Figure 1: Screenshot of ParameterTweaker.py in action. A function is created here using both Gaussian function and sine function. The parameters are visible in the widget, but clicking on the Show code on terminal will output the Fortran code snippet that is needed to define new variable of `parametered_input`*, which will produce identical output in Suberbox when used in* `NORMALD()` *function. The code snippet is the full variable definition line, user needs to change the name in* `[variable-name]`*.*

1.3.1.2 real(dp) function PERIODICAL(time, MODS)

time: `type(timetype)`
MODS: `type(parametered_input)`

A simple sine function that uses `timetype` and `parametered_input`. The function returns a value from a sine curve that has minimum in `MODS%min_c`, maximum at `MODS%max_c`, has exactly `MODS%omega` periods (e.g. `MODS%omega == 1.5d0` means one and half period) and is shifted forward by `MODS%phase` hours. Might or might not be useful for simulating temperature, humidity and other periodic values.

1.3.1.3 real(dp) function INTERP(time, conctime, conc [, row, unit])

time: `type(timetype)`
conctime: `real(dp), dimension(n)`
conc: `real(dp), dimension(n)`
row: `integer`
unit: `character(len=3)`

Function that linearly interpolates any value at time using a vector of timepoints and respective concentration time series. If `row` is provided, function will use that row as starting point for interpolation, or search for correct row if this fails (and issues a warning in that case).

By default, `INTERP()` assumes that `conctime` is in days (decimal), and uses `time%day` to search for the correct point of interpolation. If `conctime` is in some other units, optional value `unit` must be sent. Possible options for `unit` are: `'sec', 'min', 'hrs', 'day'`. Later we might want to define a global default time unit but the possibility to define unit for each case provides additional flexibility.

The `conctime` does not have to be spaced equally, but the dimensions must match the `conc` vector. It must not contain NaNs or infinities. Also, in order to successfully run the whole simulation, the `conctime` must include the start and end time of the simulation.

1.3.1.4 integer function ROWCOUNT(file_id)

file_id: `integer`

Given the UNIT (`file_id`) that points to opened file, calculates the rows in file (including empty lines), returns INTEGER, number of rows. `ROWCOUNT()` and `COLCOUNT()` rewind to the beginning of the file and return the pointer back where it was when the function was called, so it is (or should be) safe to use it at any time, even while cycling through a file.

1.3.1.5 integer function COLCOUNT(file_id [, separator])

file_id: `integer`

Given the opened `file_id`, calculates the columns in file (including empty lines), returns INTEGER, number of columns. By default, assumes columns are space-separated, and multiple spaces are counted as one. If optional character `separator` is send, it will be used as separator, in which case consecutive separators are counted as different columns. `COLCOUNT()` and `ROWCOUNT()` rewind to the beginning of the file and return the pointer back where it was when the unit was sent in the function, so it is (or should be) safe to use it at any time, even while cycling through the file.

1.3.2 Other functions

1.3.2.1 real(dp) function C_AIR_cc(T, P)

T: `real(dp)`
P: `real(dp)`

Given temperature `T` (Kelvins) and pressure `P` (Pascals) returns air concentration in cubic centimetre [#/cm^3] using ideal gas law PV = NkbT.

1.3.2.2 real(dp) function C_AIR_m3(T, P)

T: `real(dp)`
P: `real(dp)`

Given temperature `T` (Kelvins) and pressure `P` (Pascals) returns air concentration in cubic meter [#/m^3] using ideal gas law PV = NkbT.

1.3.2.3 real(dp) function hrs_to_s(h) and sec_to_d(s)

h: `real(dp)`
s: `real(dp)`

These take in hours and seconds and return seconds and hours. The purpose of such trivial functions is to lessen the probability of errors in conversion. These functions have already become obsolete because we are using `type(timetype)`.

1.3.2.4 boolean function EVENMIN(t, check [, zero])

t: `real(dp)`
check: `integer`
zero: `integer`

Checks whether time `t` (in seconds) coincides evenly with some time interval check (in minutes). As any time coincides with time 0, `.false.` is returned at t == 0. If for some reason one would want to get `.true.` at time zero, send optional integer `zero` in (can be any integer). While mostly obsoleted by `type(timetype)`, which carries booleans for different events, this function can be used to trigger one-time events etc, therefore in order to be as universal as possible, also the time is sent in as seconds instead of `timetype`.

1.4 Input

This part is under development, to be added.

1.5 Output

1.5.1 Screen output

Simulations might take a long time, and it is good to see what the model is producing even while running. Different FORMATs have been made in order to have a clear and uniform layout for output. As development of Supermodel progresses, we will have different levels of screen output (as well as file output) that the user can choose from. The formats in next table are in auxillaries.f90.

Format name (example row below)	**What it is for**	**Example of usage**
`FMT_TIME` (1)	Printing a time (hh:mm:ss) with horizontal bar; starts new "box"	`print FMT_TIME, time%hms`
`FMT10_CVU` (9)	Print a triplet of – Comment (10 characters max) – Value (REAL, will print scientific format) – unit (for example) , CHARACTER string	`print FMT10_CVU, 'C-sink:', CS_H2SO4 , ' [1/s]'`
`FMT30_CVU` (11)	Same than previous, but Comment is 30 characters long	`print FMT30_CVU, TRIM(buf), c(n)/c(1), '[]'`
`FMT10_2CVU` (5),(6),(7),	Same than FMT10_CVU, but has 2 triplets	`print FMT10_2CVU,'ACID C: ', c_acid*1d-6, ' [1/cm3]', 'Temp:', TempK, 'Kelvin'`
`FMT10_3CVU` (8)	Same than FMT10_CVU, but has 3 triplets	`print FMT10_3CVU, 'Jion1:', J_NH3_BY_IONS(1)*1d-6 , ' [1/s/cm3]','Jion1:', J_NH3_BY_IONS(2)*1d-6 , ' [1/s/cm3]','Jion1:', J_NH3_BY_IONS(3)*1d-6 , ' [1/s/cm3]'`
`FMT_LEND` (10)	prints endline of the "box". Accepts no input	`print FMT_LEND,`
`FMT_SUB` (4)	print small messages. Acceps a string	`if (time%printnow) print FMT_SUB, 'NH3 IGNORED'`
`FMT_WARN0` (12)	Print warning, with text WARNING and the string message	`print FMT_WARN0, 'UNKNOWN TIME UNIT, can not interpolate, trying with days'`
`FMT_WARN1` (2),(3)	Same that previous, but accepts a REAL after the message	`print FMT_WARN1,'real row is: ', REAL(rw)`

Example of output:

```
+.Time: 11:15:00 ..................................................................................+   (1)
| WARNING: Wrong row number is sent in to INTERP, searching for the real row now.9.0000~~~~~~~~~~~~+   (2)
| WARNING: real row is: 68.0000~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+   (3)
|   :........NH3 IGNORED                                                                           |   (4)
| ACID C:    5.058E+00 [1/cm3]    Temp: 3.008E+02Kelvin                                            |   (5)
| NH3 C:     1.735E-04 [1/cm3]    J_NH3: 6.907-316 [1/cm3]                                         |   (6)
| DMA C:     1.132E+00 [1/cm3]    J_DMA: 1.199E-65 [1/cm3]                                         |   (7)
| Jion1:     6.953-316 [1/s/cm3]  Jion1: 6.907-316 [1/s/cm3]    Jion1: 0.000E+00 [1/s/cm3]        |   (8)
| C-sink:    2.860E-02 [1/s]                                                                       |   (9)
+..................................................................................................+   (10)
| 1A3N1P/1A from NH3:              2.430E-04[]                                                     |   (11)
| WARNING: UNKNOWN TIME UNIT, can not interpolate, trying with days~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+   (12)
```

Using these (or some similar) formats will lead to more uniform and pleasing output. If they are widely used, it is easy to modify the visualisation of output.

1.5.2 File output

Currently all output is done only in NetCDF files. They can be read with many command line programs, Matlab, Python (with netCDF4 module). For more information on NetCDF: https://www.unidata.ucar.edu/software/netcdf/

This part is still under constant change, so far a file called OutputGas.nc is created, and it stores information about some gases and nucleation from ACDC.

For example how this works, see `OPEN_GASFILE()`, `SAVE_GASES()` and `CLOSE_FILES()` in `output.f90`. A header of the .nc file is below, to show what gets saved now.

```
~/supermodel-phase-1/output $ ncdump -h OutputGas.nc
netcdf OutputGas {
dimensions:
     time = 1442 ;
     Compound = 5 ;
     Constant = 1 ;
     StringL = 16 ;
variables:
     double time_in_sec(time) ;
     char gas_names(Compound, StringL) ;
          gas_names:units = "[]" ;
     double gas_concentrations(time, Compound) ;
          gas_concentrations:units = "1/m^3" ;
     double temperature(time) ;
          temperature:units = "K" ;
     double pressure(time) ;
          pressure:units = "Pa" ;
     double J_out_NH3(time) ;
          J_out_NH3:units = "1/s/m3" ;
     double J_out_DMA(time) ;
          J_out_DMA:units = "1/s/m3" ;
     double c_sink(time) ;
          c_sink:units = "1/s" ;
     double Temperature_Multipl(time) ;
     double Temperature_Shifter(time) ;
     double H2SO4_Multipl(time) ;
     double H2SO4_Shifter(time) ;
     double Base_NH3_Multipl(time) ;
     double Base_NH3_Shifter(time) ;
     double DMA_Multipl(time) ;
     double DMA_Shifter(time) ;
     double C_sink_Multipl(time) ;
     double C_sink_Shifter(time) ;

// global attributes:
          :Information = "(c) Atmospheric modelling group 2019 and (c) Simugroup 2019 (ACDC)" ;
          :Contact = "michael.boy@helsinki.fi (Superbox), tinja.olenius@alumni.helsinki.fi (ACDC)" ;
          :Software = "Superbox 0.1" ;
          :Package_Name\: = "superbox.exe" ;
          :Notes = "e.g. Sulfuric acid concentration multiplied by 0.1" ;
          :experiment = "Experiment set here" ;
}
```