Suberbox manual and developer guidelines

Date: 19 November 2019

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 Time

There is a Global Model Time (GMT) MODELTIME, which gets initialized in constants.f90. It should always be available for functions.

1.3 Derived types

1.3.1.1 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)	1d0	no	Total simulation time in hours; input from user in INIT_FILE
`SIM_TIME_S`	real(dp)	3600d0	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	yes [via ADD()]	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
`JD`	integer	0	no	Julian day of year at the start of the run
`PRINT_INTERVAL`	integer	900	no	interval of screen output in model seconds
`FSAVE_INTERVAL`	integer	300	no	interval for saving all values in model seconds
`ind_netcdf`	integer	1	yes	index of netcdf-file row (in effect the row number)
`JD`	integer	0	no (currently)	Julian day
`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`**.

1.3.1.2 type input_mod (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 **input_mod%shift**, and * for multiplying the value by **input_mod%multi**. When **input_mod** is used with + or *, only the respective variables are used, and none of the others have any influence.

Effect of different operators and **input_mod**:

real(dp) * input_mod returns **real(dp) * input_mod%multi**

real(dp) + input_mod returns **real(dp) + input_mod%shift**

The **input_mod** type holds the following variables:

	Component name	Component type	Default value	Description
General use	MODE	integer	0	0 = use values that are read in from column "col", possibly modifying by a factor or a constant 1 = Use NORMALD to create function in LINEAR mode 2 = Use NORMALD to create function in LOGARITMIC mode
	col	integer	-1	Column used for reading input if MODE is 0
	NAME	character(16)	empty	Name of the compound, variable etc.
Modify some existing values	multi	real(dp)	1d0	Used as multiplication factor if input_mod is used with operator * or .mod.
	shift	real(dp)	0d0	Used as offset if input_mod is used with operator + or .mod.
Pure Gaussian parameters	min	real(dp)	0d0	Minimum value for the function; background level.
	max	real(dp)	1d5	Maximum of the peak.
	sig	real(dp)	1d0	Width of the bell curve; sigma of the gaussian function
	mju	real(dp)	12d0	Time of the peak in hours; exact if modulation is not used, otherwise approximate.
Periodic function parameters	fv	real(dp)	0d0	Angular frequency of the modulation
	ph	real(dp)	0d0	Phase shift of the modulation
	am	real(dp)	1d0	Amplitude of the modulation function.
General use	UNIT	character(5)	'#'	Unit for the given number. Case insensitive. '#' number contentration in 1/cm3. 'ppm' parts per million (1/1e6) 'ppb' parts per billion (1/1e9) 'ppt' parts per trillion (1/1e12) 'ppq' parts per quadrillion (1/1e15) 'Pa' Pascals 'hPa' Hectopascals 'kPa' Kilopascals 'mbar' millibars 'atm' Atmosphere (101325 Pascals) 'K' Kelvin 'C' Celsius note: the user is responsible to provide units that make sense, e.g. 'C' in pressure will result in garbage.
	NAME	character(16)	'NONAME'	Name of the compound, variable etc. Filled from NAMES.DAT

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 * input_mod + input_mod`** can be used (but only in this order), also for convenience an operator .mod. has been assigned to such operation, so **`real .mod. input_mod`** has the same effect; first multiplying with **`multi`**, then adding **`shift`**.

```
real(dp) :: a = 4.0d0
type(input_mod) :: b
b%multi = 3d0
b%shift = 4.2d0
a = a .mod. b
print*, a
```

```
>>> 16.199999999999999
```

`input_mod` (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 **`input_mod%min_c == 0`** and **`input_mod%amplitude == 1`**, such operation keeps the original value intact. These are the default values of a declared **`input_mod`** variable.

Currently all input variables are stored in MODS vector, which gets allocated with the length of the NAMES.DAT file. From this vector, current concentrations at each time step are read in (interpolated with `interp`), and stored in TSTEP_CONC, which has the same length than MODS.

1.3.1.3 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.

1.4 Useful functions

All the functions under chapter 1.4 are in auxillaries.f90, except NORMALD(), which is in constants.f90

1.4.1.1 real(dp) function NORMALD(MODS[, timein])

timein: current **`type(timetype), optional`**
MODS: **`type(input_mod)`**

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 GMT, or if **`TIME`** is provided, on that 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 **`input_mod`**.

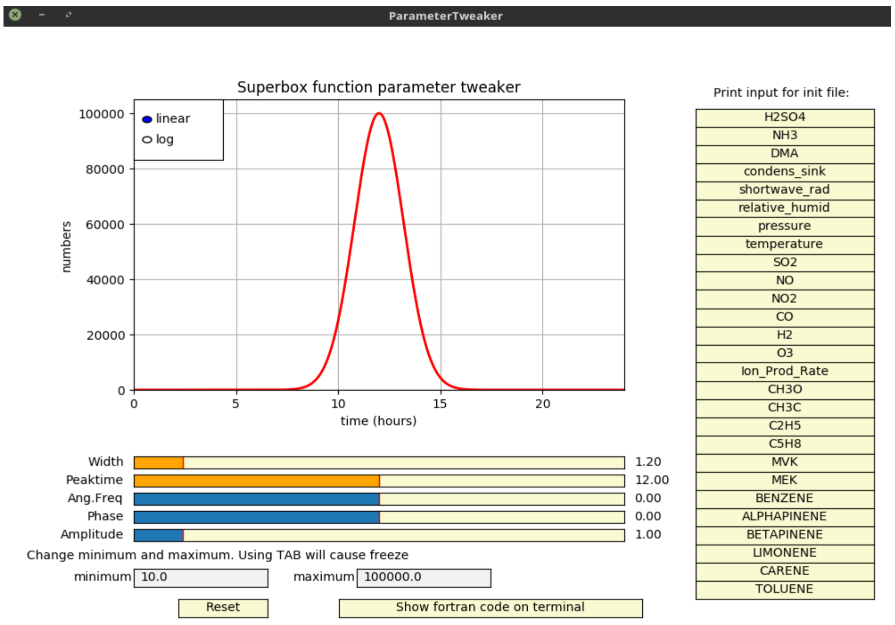

Figure 1: Screenshot of ParameterTweaker.py in action. A function is created here using pure Gaussian function. The parameters are visible in the widget, but clicking on the Show fortran code on terminal will output the Fortran code snippet that is needed to define new variable of **`input_mod`**, *which 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]`**. *On the right individual rows for INIT_FILE with correct index and name can be printed.*

1.4.1.2 real(dp) function PERIODICAL(MODS [, timein])

timein: **`type(timetype), optional`**
MODS: **`type(input_mod)`**

A simple sine function that uses **`timetype`** and **`input_mod`**. The function returns a value at GMT (or **`timein`** if provided) from a sine curve that has minimum in **`MODS%min`**, maximum at **`MODS%max`**, has exactly **`MODS%fv`** periods (e.g. **`MODS%fv == 1.5d0`** means one and half period) and is shifted forward by **`MODS%ph`** hours. Might or might not be useful for simulating temperature, humidity and other periodic values.

1.4.1.3 real(dp) function INTERP(conctime, conc [, row, unit, timein])

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

Function that linearly interpolates any value at GMT (or **`timein`** if provided) 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.

1.4.1.4 integer function ROWCOUNT(file_id [, 'comment_char'])

file_id: `integer`
comment_char: c`haracter(1)`

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. If comment_char is sent in, if first row begins with it, it will be omitted from the count. Example ROWCOUNT(51,'#')

1.4.1.5 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.

1.4.1.6 integer function IndexFromName(name [, list_of_names])

name: `Character`
list_of_names: `Character vector`
Finds out the index number used in MODS and TIMECONC using the proper name. E.g. `IndexFromName('APINENE')` would return 150 (in current NAMES.DAT). `Name` name must match the name in NAMES.DAT exactly (but need not be trimmed). If `list_of_names` is provided, function will search the name from that list and return its index.

1.4.2 Other functions

1.4.2.1 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.

1.4.2.2 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.

1.4.2.3 pure character function UCASE(word)

word: `character(:)`

Returns the uppercase string of `word`. Located in constants.f90.

1.4.2.4 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)`.

1.4.2.5 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`.

1.5 Input

Input handling is done in module input.f90. The main input method for SUPERBOX is the INITFILE, and in fact this is the only compulsory input if only parametrized and constant values are used.

1.5.1 Variable declarations

In practice all of the variables that are user-configurable, are declared in input.f90. Some exceptions are variables that contain concentrations for the whole model run, these are declared in constants.f90, but they too are allocated and filled in input.f90. During development (= currently) the details where each variable is declared is changing, so searching through the source files is your friend.

1.5.2 Named indices

The variables which are stored in MODS, come from NAMES.DAT. This file stores most of the user supplied input, including the chemical compounds that MCM can take as input. NAMES.DAT is structured so that each line contains one variable, and before MCM compounds there is a line with "#".

Those variables that come before "#" have named indices like `inm_TempK`, `inm_pres`, `inm_RH`, `inm_CS`, `inm_CS_NA`, `inm_swr` etc. The actual index number is parsed from NAMES.DAT, based on the order they are in the file. If new variables are to added, they will first and foremost be added to NAMES.DAT. Form there they will be read in MODS, and can be referred to with `IndexFromName('variable name')` or a named index could be added to input f90. In latter case, it would first be declared in the beginning, and then this variable should be filled in subroutine `NAME_MODS_SORT_NAMED_INDICES`, found in input.f90. The exact order there does not have to match the one in NAMES.DAT, but for clarity this is advised.

1.5.3 User-supplied input

The user can change all the necessary options using the INITFILE, without needing to recompile. Besides INITFILE, no additional input is absolutely necessary if no measurements or background aerosol concentrations are used. Usually they are, and therefore additional input needs to be prepared. The format for all input is same: a text file containing the values at given times, as two-dimensional matrix. The rows are values for each time, time being in the first column, and the individual variables on each column. The column number that should be used, is defined in INITFILE, by defining the corresponding MODS:

```
#     mode col multi        shift        min          max         sig         mju         fv          ph          am          unit name

&&NML_MODS

MODS(1) = 1 -1  1.00000d+00  0.00000d+00 -7.00000d+00  1.50000d+01 6.860000d0 9.960000d0 0.540000d0 2.980000d0 0.500000d0 'C'! TEMPK

MODS(2) = 0 -1  1.00000d+00  1.00000d+03  1.00000d+01  1.00000d+05 1.000000d0 12.000000d0 0.000000d0 0.000000d0 1.000000d0 'MBAR'! PRESSURE

MODS(3) = 0  3  1.00000d+00  4.50000d+01  1.00000d+01  1.00000d+05 1.000000d0 12.000000d0 0.000000d0 0.000000d0 1.000000d0 ! REL_HUMIDITY
```

Above temperature, pressure and relative humidity are defined. First row is temperature, because index in MODS(1) is, corresponding to first row in NAMES.DAT. Becuse `mode`=1, temperature is modelled using a parametrized function, in linear mode. In this case (`mode`>0) the column number (`col`), multiplication factor (`multi`) or constant shift (`shift`) does not matter. The unit is Celsius, indicated by `unit`. Pressure has `mode`=0, meaning that it is either measured or set to a constant. Since `col`=-1, no file will be read, and default value 0.0 is modified with `multi` and `shift`. In this case a constant of 1000 is added, and the `unit` tells the program that they are millibars. The units will be converted to Pascals and Kelvins for both of these variables. Relative humidity is read in from column 3 (1st column [time] being 1). The file from where the values are read is defined in INITFILE in &NML_ENV → ENV_file.

There is also another input file option, MCM_FILE → MCM_path. The MCM variables (these are the variables in NAMES.DAT that come after "#") will be read in from this file. Note that both of these variables can point to same file. Also `col` can point to same columns, and this is (currently) not checked in any way.

Input related to particles is defined under &NML_PARTICLE. The aerosol part is under construction and no manual is written. The details concerning aerosol options will probably change a lot in the future.

1.5.3.1 Validation of input

NAMES.DAT contains much more compounds that the user typically has in the chemistry. Therefore before starting, SUPERBOX goes through the filled MODS-files, and if the user tries to send input, or otherwise modify a compound that is not in the chemistry, the program will report it and terminate. Resolve the conflict by checking the index in MODS(index) in INITFILE, either using Setup.py or manually inspecting NAMES.DAT. Probably the index is wrong by accident, or if not, then the chemistry is not able to handle the input. If inorganic compounds are sent in (the ones before the "#"), and are not found in chemistry, a warning is issued but the program will continue.

1.5.3.2 Conversion of units

The possible units are shown in subsection 1.3.1.2. The units for temperature and pressure are converted before program run, and mixing ratios are converted on the fly in .mod. During the program run MODS%unit, MODS%min and MODS%multi might be changed according to the unit conversion, but when INITFILE is saved (see 1.6.2) the original settings are saved.

1.6 Output

1.6.1 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.

1.6.2 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" ;
}
```