Frank Hamberg, Cora Kohlmeier

CEMoS 5.0

C Environment for Simulation

Oldenburg, February 21, 2023

Cora Kohlmeier
Institut für Chemie und Biologie des Meeres
Carl von Ossietzky Universität Oldenburg
c.kohlmeier@uol.de

Contents

1 Introduction

CEMoS is a model environment for handling coupled ODE's with up to hundreds equations, state variables, derived variables and parameters. It has been tested for several types of models (Hamberg, 1996). The idea of **CEMoS** bases on (Ruardij *et al.* , 1995). **CEMTK** is the graphical user interface for **CEMoS**. Both are parts of the **Tiger Graphics TigerPack** , which also provides **MoViE** and **CEvoS**. Developing under **CEMoS** needs a basic understanding of **C** programming.
This document describes the functionality of **CEMoS** in the first part (3) and the use of **CEMTK** in the second part (13).
Developing under **CEMoS** needs a basic understanding of **C** programming.

License

2 Installation of TigerPack

Tiger Graphics TigerPack is written for UNIX or Linux[1]. It will also run on Mac OSX. Windows user may use it within a virtual machine.
TigerPack consists of 4 parts:

- **CEMoS** core system for the simulation
- **CEMTK** optional but highly recommended GUI for **CEMoS**
- **CEvoS** system for evolutionary parameter adaptation for **CEMoS** models
- **MoViE** visualization package

[1]Depending on the different properties of the operating systems the binary files are not portable from Intel machines to other architectures.

CEMTK and **CEvoS** can only be used if **CEMoS** is installed. **MoViE** can be used for any result files with appropriate format (see chapter **??**).

2.1 Requirements

All part of **TigerPack** needs

- `csh`, `tcsh` or `bash`
- gcc-2.6.3 or higher
- `make`
- Tcl/TK[2]

optional it is recommended to install

- `kwrite` or `gedit`
- gdb with `ddd` or `insight`
- `grace`
- `gv`
- `gnuplot`
- `kdiff3`

Nautilus/Dolphin user should also install

- `nautilus-open-terminal`

2.2 Installation

Copy the file `tigerpack.tar.gz` to the directory where do you like to install it and where do you have write permissions. Extract it by

```
tar -xzvf  tigerpack.tar.gz
```

Open a command terminal in the directory `TIGERPACK` and type the command

```
./setup
```

After that, logout and login again to set the environmental variables (reboot is NOT necessary).

[2]**CEMoS** itself needs it only for the installtion. This can also be done manually if Tcl/TK is not available

2.3 Reinstalling after update

If you already have a running **TigerPack** and you would like to install an update Open a terminal in `TIGERPACK` and type the command `./reinstall`. No need to logout but existing **CEMoS** models must be recompiled.

3 Getting started with CEMoS

There are two possibilities to operate **CEMoS** :

1. Operating from a terminal window (for details see 4.3)
   To operate **CEMoS** from a terminal window, the following shell scripts are installed:

   - to compile a model:
     `compile <model-directory> [debug]`
   - to start a simulation: `run`

2. Operating from the graphical user interface **CEMTK**
   To start **CEMTK** go to the directory where the model (the file `cemos.par`) is located or where a new one should be created and give the command `cem`. **CEMTK** is described in part 13.

4 Structure and functionality of CEMoS

CEMoS allows a convenient implementation of different model types, where the structure of variables is generated automatically from the model definitions.

CEMoS is specialized to support so called box models where similar processes work in different geographical regions under different environmental conditions.

Therefore every state variable is indexed (box number) and can be attached to a region.

To connect the regions by transport processes further variables derived from the states are made available by **CEMoS** .

The interpretation of the indexes as box numbers is not the only possibility. It is also possible to simulate one model with different parameter settings in parallel. Here every index represents a model run with its specific parameter settings.

CEMoS gives the modeler a high flexibility in controlling the simulation. Different integration methods are available and changes of parameter values during sensitivity analysis can be made without recompiling the model. Furthermore the set of variables which may be stored is arbitrary and can be modified for every simulation without recompiling.

If a complex model shall be divided into several submodels on different time scales is it quite easy to simulate these submodels with different integration methods or with different time steps; the underlying concept *operator-splitting* is described in 6.2. Here **CEMoS** takes care of the integration control.

4.1 The structure of models in CEMoS

The location of a model is independent of the location of the **CEMoS** package. It may be located in an arbitrary directory. All files needed for a **CEMoS** model may be located in one directory, but in the following a better way is described. If a model shall be implemented in a directory called `mymodel` the executable model, the files controlling the simulation and the result files are placed in it whilst the model code, the parameter files and the files generated by **CEMoS** are located in a subdirectory commonly called `main`. The directory `mymodel` should be the working directory during the work with the model. The usual structure of a model is shown in figure4.1.

The following files must be prepared by the modeler, all other files are generated automatically:

`model.def`	**one** file for the central model definitions in `main`
`*.def`	files containing further definitions of model parameters in `main`
`*.c`	files containing the **C**-Code of the model in `main`
`cemos.par`	**one** file for the simulation control in `$pwd`

Additionally an executable shell script or program named `install` may exist which will be executed automatically before the compilation starts.

4.2 Implementation of a model

In this section the implementation of a small predator-prey-system is described:

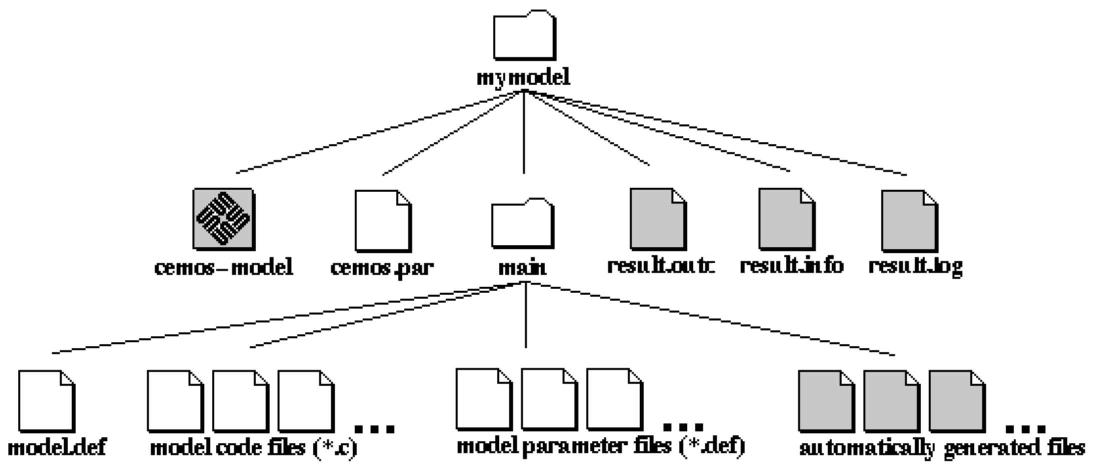


Figure 4.1: The file structure of a model under **CEMoS** . The white icons are the files which must be prepared by the modeler, the grey ones are automatically created during compilation resp. simulation.

Let
a be the growth rate of species X,
b a feeding parameter for the density dependent grazing rate $\mathrm{b}\cdot\mathrm{X}$ of the predator Y and
m the mortality of the predator Y .

Then the model is described by the following ODE-system:

$$\begin{aligned}\dot{\mathrm{X}} &= \mathrm{a}\cdot\mathrm{X}-\mathrm{b}\cdot\mathrm{X}\cdot\mathrm{Y}\\ \dot{\mathrm{Y}} &= \mathrm{b}\cdot\mathrm{X}\cdot\mathrm{Y}-\mathrm{m}\cdot\mathrm{Y}\end{aligned}$$

To implement this model in a directory called `mymodel` the following four files are needed:

1. `model.def` model definitions in `mymodel/main`
2. `par.def` parameter definitions in `mymodel/main`
3. `model.c` **C**-code in `mymodel/main`
4. `cemos.par` simulation control in `mymodel`

The file `model.def` contains the central variables and parameters:

```
/*
*     Definition of state variables and other global variables:
*/
%numeric double

%states
X[1]={0.5};
Y[1]={1.0};
```

The line `%numeric double` forces **CEMoS** to simulate the model in double precision, the identifier `%states` marks the definition block for state variables. **Remark:** states must be defined as one dimensional arrays and initial values must be set.

The file `par.def` contains the definition of model parameters.

```
%real_par
a=0.5;
b=0.3;
m=0.1;
%change
m=0.15;
```

The identifier `%real_par` marks the block of scalar real parameters. At the start of the simulation the here defined parameters will get their values. The identifier `%change` assigns an overruling of the values from the definition block by these values without recompiling the model. The content of this file can also be set in the file `model.def`, but the splitting into different files keeps care of the track, if the models become larger. The file `model.c` contains the **C**-code of the model:

```
#include "struct.h"
#include "par.h"

void model(void)
{
SX[1]=a*X[1]-b*X[1]*Y[1];
SY[1]=b*X[1]*Y[1]-m*Y[1];
}
```

With the `#include`-statements the variables defined in the previous files are made available, where the files `struct.h` and `par.h` are automatically generated by **CEMoS** from the files `model.def` and `par.def`.

The temporary derivatives ($\dot{X}$) are identified in **CEMoS** by a leading `S`, these `S` variables are automatically generated by **CEMoS** and available for all state variables. The file `cemos.par` contains all information of the duration of a simulation, the integration method etc., the \%`change`-statement for variables and parameters defined in the `model.def` , as well as the \%`store`-statement for the information about variables to be stored.

```
#include "main/model.def"
%simulation_parameters
starttime=0.0;
endtime=100.0;
storetime=50.0;
outdelt=1.0;
model_dir main
recalc_globals=0;
storestart=1;

%integration_par
mindelt=1.e-15;
maxdelt=1.0;
accuracy=.1e-3;
method=3;

%change
X[1]=0.45;
%store
X[1];Y[1];
```

The time unit (TU) for the simulation is 1.

In the block `%simulation_parameters` the simulation parameters are set: the starting time 0, the endtime 100 TU and the storing interval of 1(`outdelt`), where the storing of the results starts with the 50th TU (`storetime`) . The line `model_dir main` specifies where the model code and the parameter files are located relatively to this file (`cemos.par`).
In the block `%integration_par` the integration parameters are set: a minimum time step of 10^{-15} TU, a maximum time step of 1 TU, a maximum relatively error of $0.1 * 10^{-3}$, as well as the integration method, here a Runge-Kutta method of 4th order with time step adaptation (see 6).
Below the line `%change` the variable `X` gets a starting value different of its definition.
Below `%store` the variables which shall be stored are listed.

4.3 Compilation and simulation start

Note: This section describes the control of **CEMoS** without the graphical front end **CEMTK** (see **??**).

It is most convenient to give all **CEMoS** commands in the model directory `mymodel`.
The compilation is then started by the command
`compile main`
in this directory.
The argument `main` determines the directory where the model code is located.

With this simple script the compiler flags are set as follows:

(a) Warning levels: `-Wall -Wredundant-decls -pedantic`

(b) Optimization: `-O3`

Remark: Those automatically generated fiiles which are only executed once at simulation start are compiled without optimization because they may be very long and the optimization by the compiler may need more time than ever can be gained during execution.

After the successful compilation (and linking, which is automatically done) the executable model will be moved to the working directory `mymodel`. The object files will be archived in a file `cemos.a` in the `main` directory. All automatically generated stuff and all objects will be removed so that the contents of the model directory stays clear.

The simulation can now be started by the command
`run`
The simulation results will be stored in a file called `result.outc` .
If the simulation doesn't run correctly the model can be debugged. In this case the debug flag must be set in the compilation. This can be done by starting the compilation with the command
`compile main debug`
or
`compile debug main`
In this case all automatically created files will remain in the model directory and the compiler flags are as follows:

(a) Warning levels: `-Wall -Wredundant-decls -pedantic`

(b) Debug: `-g`

(c) Optimization: `none` (because debuggers often have problems with optimized executables)

(d) **CEMoS** specials: `-D__verbose__` (to activate extensive information output from **CEMoS** by conditional compilation)

If all errors are corrected the model should be 'cleaned' to get rid of the automatically generated files by the command
`clean main`
and then recompiled without debug option to save simulation time.
Cleaning is also done when the command `compile main` is given.

4.4 Model definition and data types

During the compilation the `.def` files are evaluated and the information is transformed into **C**-header-files and **C**-files. This construction allows to modify parameter values without recompilation. (see also 5).

If f.e. the definition `xyz=3.0;` is set in the file `xxx.def`, this variable is available in every code segment which contains the line `#include <xxx.h>`. This will be explained in more detail in the next section.
The file `model.def` plays an important role. The name of the file may be expanded in front (f.e. `sea_model.def`). In any case a file named `struct.h` is generated, which disposes the model definitions.

The line `#include <struct.h>` in a file makes all model definitions available. Therefore the modeler is not forced to think about the variable handling between his model files and the integration.

Within **CEMoS** different data types are available for automatic generation.

For all variables with a dimension the indices start with one and end with the specified number (an array defined by `xyz[15];` can be addressed by `xyz[1]` – `xyz[15]`).

Remark: **CEMoS** supports one dimensional and two dimensional arrays for automatic generation.
Remark: All names are case-sensitive: `P1c` and `p1c` are different variables.
Remark: The number of characters (alphanumeric) in a variable name mustn't exceed 80 including the indices and brackets.

4.4.1 Scalar variables (0D variables)

The basic structure and the content of the file `model.def` regarding 0D variables and their derivatives can be seen in following example:

```
/*
*     Definition of state variables and other global variables:
*/
%0D_states
A=0.5;
B=0.9;

%0D_globals
X;

%int_par
iswTRSP=1;

%real_par
Zero=1.0e-6; /* Pseudo-Zero for some processes */
```

State variables

State variable are of type `real`. They are the central variables of a model. They will be integrated.

Example:

```
%0D_states
A=0.5;
B=0.9;
```

By evaluation of these lines the following variables are generated and set automatically:

- the variable `num_0D_states` is set the number of scalar state variables (here 2)
- a scalar `real A` of length 5, containing the values of the state variable `A` during simulation (same for `B`).
- a scalar `real SA` of length 5, containing the values of the right hand sides of the ODE's. This values are available for the integration. The vector elements are set to zero after every integration step (same for `SB`).

- some internal pointers, which allow access to these variables f.e. for storing of simulation results.
- a vector `real` `KKK` here `KKK[2]`
- a vector `real` `SKKK` here `SKKK[2]`

The vectors `KKK[2]`, `SKKK[2]` are associated with `A B` resp. `SA SB`, so that the content of `A` can be retrieved also by `KKK[1]`, and the statement `SKKK[2]=0.0;` also sets `SB` to zero.

Remark: The identification of thes vector elements and the scalar state variables is not done instantaneously but only during integration (which means after complete model execution.

If the simulation is started the state variables are initialized with the given values.

Remark: Initial values must be set for every state variable!

Remark: The identifier `%0D_states` is only allowed in the file `model.def`!

Scalar global variables

Variables of type `real`. In contrast to state variables global variables are not initialized and they are not integrated. They are useful if f.e. intermediate results are needed in different model parts. Variables of this type can be stored, if they are defined in the file `model.def`.

For example:

```
%0D_globals
X
```

Parameters

Scalar variables of type `integer` or `real`. They contain f.e. biological or physical values which don't change during the simulation. They can also contain control values for controlling the simulation

For example:

```
%int_par
iswTRSP=1;
%real_par
Zero=1.0e-6; /* Pseudo-Zero for some processes */
```

the values of these parameters may be changed for a simulation by the statement

```
%change
iswTRSP=0;
Zero=1.0e-12;
```

without recompiling.

Due to the structure of **CEMoS** it is allowed to define a constant f.e. by the following statement:

```
%real_par
eight_thirds=8./3.;
```

Attention: This is only allowed in the definition and **not** in the `%change` block! This construction is only valid for the data type `integer` or `real` The evaluation of functions (f.e. `Pi=4.0*atan(1.0);`) is not allowed.

Remark: These variables can't be stored.

Local variables

User defined local variables within the model underly the **C**-syntax. They are not affected by the integration and can't be stored.

4.4.2 1D variables

The basic structure and the content of the file `model.def` regarding 1D variables and their derivatives can be seen in following example:

```
/*
*     Definition of state variables and other global variables:
*/
%states
X[5]={0.5,2.0,5,0.5,1.0,0.5};
Y[5]={1.8,0.7,0.5,0.5,0.9};

%globals
sumX_Y[5]

%int_par
iswTRSP=1;

%real_par
Zero=1.0e-6; /* Pseudo-Zero for some processes */

%real_ind_par
vol[5]={2.0,1.0,1.0,1.0,1.0};

%int_ind_par
upper[5]={0,1,2,3,4};

%real_derived_from_states
wHI+variable

%int_derived_from_states
itrsp+base

%change
itrspX=11;
itrspY=12;
```

State variables

State variable are of type `real`. They are the central variables of a model. They will be integrated.

Example:

```
%states
X[5]={0.5,2.0,5,0.5,1.0,0.5};
Y[5]={1.8,0.7,0.5,0.5,0.9};
```

By evaluation of these lines the following variables are generated and set automatically:

- the variable `numstates` is set the number of state variables (here 2)
- a vector `real X` of length 5, containing the values of the state variable `X` during simulation (same for `Y`).
- a vector `real SX` of length 5, containing the values of the right hand sides of the ODE's. This values are available for the integration. The vector elements are set to zero after every integration step (same for `SY`).
- some internal pointers, which allow access to these variables f.e. for storing of simulation results.
- a matrix `real CCC` here `CCC[2][5]`
- a matrix `real SCCC` here `SCCC[2][5]`

The arrays `CCC[2][5], SCCC[2][5]` are associated with `X Y` resp. `SX SY`, so that the content of `X[3]` can be retrieved also by `CCC[1][3]`, and the statement `SCCC[2][5]=0.0;` also sets `SY[5]` to zero.

If the simulation is started the state variables are initialized with the given values.

Remark: Initial values must be set for every state variable!

Remark: The identifier `%states` is only allowed in the file `model.def`!

Remark: The scope of indices must be identical for all variables of the type `%states`.

Global variables

Vectors of type `real`. In contrast to state variables global variables are not initialized and they are not integrated. They are useful if f.e. intermediate results are needed in different model parts. Variables of this type can be stored, if they are defined in the file `model.def`. Then their scope must be the same as the scope of the state variables.

For example:

```
%globals
sumX_Y[5]
```

Parameters

Scalar variables of type `integer` or `real`. They contain f.e. biological or physical values which don't change during the simulation. They can also contain control values for controlling the simulation

For example:

```
%int_par
iswTRSP=1;
%real_par
Zero=1.0e-6; /* Pseudo-Zero for some processes */
```

the values of these parameters may be changed for a simulation by the statement

```
%change
iswTRSP=0;
Zero=1.0e-12;
```

without recompiling.

Due to the structure of **CEMoS** it is allowed to define a constant f.e. by the following statement:

```
%real_par
eight_thirds=8./3.;
```

Attention: This is only allowed in the definition and **not** in the `%change` block! This construction is only valid for the data type `integer` or `real` The evaluation of functions (f.e. `Pi=4.0*atan(1.0);`) is not allowed.

Remark: These variables can't be stored.

Parameter vectors

Vectors of type `integer` oder `real`; they are similar to parameters, but they are indexed. They contain f.e. biological or physical values which don't change during the simulation, but are different for the different boxes.

For example:

```
%real_ind_par
vol[5]={2.0,1.0,1.0,1.0,1.0};
%int_ind_par
upper[5]={0,1,2,3,4};
```

They may be changed for a simulation by

```
%change
vol[1-2]={3.0,10.0};
upper[1-2]={2,1};
```

without recompiling.

Remark: In `%change` blocks the numbers in `[...]` are treated as lists, such `vol[2]={3.0,10.0}` will set `vol[2]` to 3.0 and will not affect `vol[1]`.
Remark: These variables can't be stored.

Derived variables

Arrays of type `integer` oder `real`. There are two different types available, `+variable` and `+base`.
These types are mainly needed to structure large models and allow (similar to the matrices `CCC` and `SCCC`) loops over all state variables. The scalar form `...+base` is useful to activate or deactivate processes for some variables, f.e. transport processes can be implemented for all variables similarly, but are activated only for a few, using these variable to control them.
The type `+variable` is the indexed version, to have also control over the boxes. The definition of these variables is only allowed in the file `model.def` . They are strongly associated to the definition of state variables.
There are three possibilities to get variables derived from the definition of state variables

- `%real_derived_from_states` in the variants `+variable` and `+base`

- `%global_derived_from_states` only as `+variable`
- `%int_derived_from_states` in the variants `+variable` and `+base`

which are handled differently during the simulation:

- `%real_derived_from_states +variable` are zeroised every time before the model is executed from an integration method and additionally after every storing of model results, if a recalculation of external processes (e.g. time series from data files or interpolation functions) before storing is forced by setting `recalc_globals=1`.
- `%global_derived_from_states +variable` are zeroised after every storing of model results.
- `%int_derived_from_states +variable` are zeroised every time before the model is executed from and integration method and additionally after every storing of model results, if a recalculation of external processes (e.g. time series from data files or interploation functions) before storing is forced by setting `recalc_globals=1`.

All derived variables of type `+base` are not affected by the integration or the storage control. Some examples as a little help:

%real_derived_from_states

The definition

```
%real_derived_from_states
wHI+variable
pA+base
```

generates the following variables:

- The two vectors `wHIX[5]` and `wHIY[5]` of type real, and a matrix `wHICCC[2][5]` of type real. These variables are not integrated, but set to zero everytime before the model is executed during integration.
- A parameter vector `pACCC[2]` of type real corresponding to the pointers `*pAX, *pAY`. These variables are not affected by the integration.

%global_derived_from_states

The definition

```
%global_derived_from_states
wDO+variable
```

generates the following variables:

- The two vectors `wDOX[5]` and `wDOY[5]` of type real, and a matrix `wDOCCC[2][5]` of type real. These variables are not integrated, but set to zero everytime after results are stored.

%int_derived_from_states

The definition

```
%int_derived_from_states
iH+variable
itrsp+base
```

generates the following variables:

- The two vectors `iHX[5]` and `iHY[5]` of type integer, and a matrix `iHCCC[2][5]` of type integer. These variables are not integrated, but set to zero everytime after results are stored.
- A parameter vector `itrCCC[2]` corresponding to the pointers `*itrX, *itrY`. These variables are not affected by the integration.

The values of `+base` parameters can only be set (and may be changed) by:

```
%change
itrspX=11;
itrspY=12;
iHX[1-5]={1.0,2.0,3.0,4.0,0.0};
iHY[1-5]={0.0,1.0,2.0,3.0,4.0};
```

Local variables

User defined local variables within the model underly the **C**-syntax. They are not affected by the integration and can't be stored.

4.4.3 2D variables

The extension of **CEMoS** to provide matrices of variables which are managed and which can be handled in the same way as vectors (1D variables) is a first step to the implementation of partial differential equations with **CEMoS** . Anyhow, there are some restrictions in the current implementation which are described at the palce where they become effective. The basic structure and the content of the file `model.def` regarding 2D variables and their derivatives can be seen in following example:

```
/* 2D variables */
%2D_states
AAA2D[5][7]={7*1,7*2,7*3,
4,4,4,4,4,4,4,7*5};
BBB2D[5][7]={7*1,7*2,7*3,
4,4,4,4,4,4,4,7*5};

%2D_globals
aaa2d[5][7]
bbb2d[5][7]

%2D_real_ind_par
rrr2D[5][7]={7*1,7*2,7*3,4,4,4,4,4,4,4,7*5};
%2D_int_ind_par
iii2D[5][7]={7*1,7*2,7*3,4,4,4,4,4,4,4,7*5};

%2D_real_derived_from_states
xxx2d+variable
fff2d+base
%2D_int_derived_from_states
vvv2d+variable
bbb2d+base
```

Remark: Statements like `{7*1,3*2,4*9,7*3,....}` are expanded by **CEMoS** to `{1,1,1,1,1,1,1,2,2,2,9,9,9,9,3,3,3,3,3,3,3,....}` and line breaks may appear in those statements as well as standard `C` comments `/*Comment */`.

2D State variables

Same as 1D-state variables 2D-state variables are of type `real`. They are the central variables of a model. They will be integrated.

Example:

```
%2D_states
AAA2D[5][7]={7*1,7*2,7*3,
4,4,4,4,4,4,4,7*5};
BBB2D[5][7]={7*1,7*2,7*3,
4,4,4,4,4,4,4,7*5};
```

By evaluation of these lines the following variables are generated and set automatically:

- the variable `num_2D_states` is set the number of state variables (here 2)
- a matrix `real AAA2D` with 5 columns and 7 lines, containing the values of the state variable `AAA2D` during simulation (same for `BBB2D`).
- a vector `real SAAA2D` with 5 columns and 7 lines, containing the values of the right hand sides of the ODE's. This values are available for the integration. The matrix elements are set to zero after every integration step (same for `SBBB2D`).
- some internal pointers, which allow access to these variables f.e. for storing of simulation results.
- a so called 3D-tensor `real DDD` here `DDD[2][5][7]`
- a 3D-tensor `real SDDD` here `SDDD[2][5][7]`

The arrays `DDD[2][5][7]`, `SDDD[2][5][7]` are associated with `AAA2D BBB2D` resp. `SAAA2D SBBB2D`, so that the content of `AAA2D[3][4]` can be retrieved also by `DDD[1][3][4]`, and the statement `SDDD[2][3][5]=0.0;` also sets `SBBB2D[3][5]` to zero.

If the simulation is started the 2D state variables are initialized with the given values.

Remark: Initial values must be set for every 2D state variable!
Remark: The identifier `%2D_states` is only allowed in the file `model.def`!
Remark: The scope of indices must be identical for all variables of the type `%2D_states`.

Global variables

Matrices of type `real`. In contrast to 2D state variables 2D global variables are not initialized and they are not integrated. They are useful if f.e. intermediate results are needed in different model parts. Variables of this type can be stored,

if they are defined in the file `model.def`. Then their scope must be the same as the scope of the 2D state variables.
For example:

```
%2D_globals
aaa2d[5][7]
bbb2d[5][7]
```

Parameter matrices

Matrices of type `integer` oder `real`; they are similar to parameters, but they are indexed. They contain f.e. biological or physical values which don't change during the simulation, but are different for the different grid cells.
For example:

```
%2D_real_ind_par
rrr2D[5][7]={7*1,7*2,7*3,4,4,4,4,4,4,4,7*5};
%2D_int_ind_par
iii2D[5][7]={7*1,7*2,7*3,4,4,4,4,4,4,4,7*5};
```

They may be changed for a simulation by

```
%change
rrr2D[3][1-2]={3.0,10.0};
iii2D[2][1-2]={2,1};
```

without recompiling. **Remark:** These variables can't be stored.

Derived variables

Arrays of type `integer` oder `real`.

There are two different types available, `+variable` and `+base`.
These types are mainly needed to structure large models and allow (similar to the tensors `DDD` and `SDDD`) loops over all state variables. The scalar form `...+base` is useful to activate or deactivate processes for some variables, f.e. transport processes can be implemented for all variables similarly, but are activated only for a few, using these variable to control them.
The type `+variable` is the indexed version, to have also control over the grid cells.
The definition of these variables is only allowed in the file `model.def` . They are strongly associated to the definition of 2D state variables.
There are three possibilities to get variables derived from the definition of 2D state variables

- `%2D_real_derived_from_states` in the variants `+variable` and `+base`
- `%2D_global_derived_from_states` only as `+variable`
- `%2D_int_derived_from_states` in the variants `+variable` and `+base`

which are handled differently during the simulation:

- `%2D_real_derived_from_states +variable` are zeroised every time before the model is executed from an integration method and additionally after every storing of model results, if a recalculation of external processes (e.g. time series from data files or interpolation functions) before storing is forced by setting `recalc_globals=1`.
- `%2D_global_derived_from_states +variable` are zeroised after every storing of model results.
- `%2D_int_derived_from_states +variable` are zeroised every time before the model is executed from an integration method and additionally after every storing of model results, if a recalculation of external processes (e.g. time series from data files or interploation functions) before storing is forced by setting `recalc_globals=1`.

All derived variables of type `+base` are not affected by the integration or the storage control.

Some examples as a little help:

%2D_real_derived_from_states

The definition

```
%2D_real_derived_from_states
xxx2d+variable
fff2d+base
```

generates the following variables:

- The two matrices `xxx2dAAA2D[5][7]` and `xxx2dBBB2D[5][7]` of type real, and a tensor `xxx2dDDD[2][5][7]` of type real. These variables are not integrated, but set to zero everytime before the model is executed during integration.
- A parameter vector `fff2dDDD[2]` of type real corresponding to the pointers `*fff2dAAA2D, *fff2dBBB2D`. These variables are not affected by the integration.

%2D_global_derived_from_states

The definition

```
%global_derived_from_states
ggg2d+variable
```

generates the following variables:

- The two matrices ggg2dAAA2D[5][7] and ggg2dBBB2D[5][7] of type real, and a tensor ggg2dDDD[2][5][7] of type real. These variables are not integrated, but set to zero everytime after results are stored.

%2D_int_derived_from_states

The definition

```
%2D_int_derived_from_states
vvv2d+variable
bbb2d+base
```

generates the following variables:

- The two matrices vvv2dAAA2D[5][7] and vvv2dBBB2D[5][7] of type integer, and a tensor vvv2dDDD[2][5][7] of type integer. These variables are not integrated, but set to zero everytime after results are stored.
- A parameter vector bbb2dDDD[2] corresponding to the pointers *bbb2dAAA2D, *bbb2dBBB2D. These variables are not affected by the integration.

The values of +base parameters can only be set (and may be changed) by:

```
%change
bbb2dAAA2D=21;
bbb2dBBB2D=42;
vvv2dAAA2D[5][7]={7*1,7*2,7*3,4,4,4,4,4,4,4,7*5};
vvv2dBBB2D[5][7]={7*5,7*6,7*8,4,4,4,4,4,4,4,7*9};
```

4.4.4 User defined structures

Actually user defined structures are only allowed within the model code for structuring the model. They are not affected by the integration and can't be stored.

4.4.5 Preprocessor-statements

All files containing definitions (`.def` files) are prepared by the `C` preprocessor during compilation and once again, when the simulation is started. Therefore all preprocessor statements such as comments, `#include`-statements etc. may be used in variable definitions and `%change`-bocks as well as in `%store`-blocks.

Remark: During simulation the file `model.def` is not checked for `change` blocks, but the file `cemos.par` is. Therefore the file `model.def` must be included in the file `cemos.par` by the statement `#include "main/model.def"` to get changes evaluated. All variables defined in the file `model.def` may be changed by `%change` in the file `cemos.par`.

4.4.6 Conditional compiling

The file `model.def` may contain lines of the type `%setup ABCD`. If lines starting with `%setup` appear, a file `compiler_setup.h` is generated, which contains just the lines with the respective compiler directives:
`#define ABCD` (looking at the a.m. example)
The file `compiler_setup.h` is automatically included via `struct.h`, but may also be included in all `*.def` files, as well as in all files being included in those `*.def` and the `cemos.par`. Such, contructions like:

```
#if defined(ABCD)
%store
X1x[1-3];
#elif
%store
X1x[1,2];X2x[1-3];
#endif
```

are possible, as well as all conditional compiling within the model code files, to have an easy possibilty to generate model variants without having parallel setups in different directories.
If no line starting with `%setup` appears in the file `model.def`, the result is an empty file `compiler_setup.h`.

5 Controlling the simulation

The following example explains the structure and content of the file `cemos.par`:

```
%boxes_not_active 2
#include "main/model.def"

%simulation_parameters
starttime=0.0;
endtime=720.0;
outdelt=1.0;
year=88.0;
cycle=360.0;
model_dir main
multi=2;
recalc_globals=1;
storestart=0;

%integration_par1
mindelt=1.e-15;
maxdelt=1.0;
accuracy=.0001;
method=3;

%integration_par2
mindelt=1.e-15;
maxdelt=1.0;
relrate=1.0;
relchange=0.5;
method=1;

%change
iswTRSP=1;

%change
X[1-2]={2.0,0.5};
Y[1-2]={1.0,0.5};

%store
X[1-2];
Y[1-2];
```

With the statement `%boxes_not_active` boxes are set not to be considered by the integration. The initial values of these boxes are kept constant during simulation. This is f.e. useful for transport processes over boundaries. This statement is optional.

With the statement `#include "main/model.def"` all information from the file `main/model.def` is available and may be changed in the `%change` block. (this is important if derived variables of the type `+base` shall be initialized with values – this is only possible in the `%change` block and not in the definition). This statement is optional.

The block `%simulation_par` is already described in 4.2.
For the statement `multi=2;` see 6.2.

The setting `recalc_globals=1` forces a recalculation of the model without integration to get `global` and `global_derived_from_states` recalculated before storing simulation results. This is very helpful, if the model is run with different integration methods on different time steps (operator splitting, see 6.2), and e.g. not all derived variables are effected by all integration methods, but shall be stored for diagnostic purposes. The default setting is `recalc_globals=0`. For further information see appendix 11.

For the parameter `storestart` see section 8 In `%change` all parameters may be modified which have been defined in the file `model.def`, f.e. state variables can get new initial values here without recompiling the model.

Furthermore files may be included by `#include` statements to keep care of the track.

6 Integration methods

CEMoS supports several integration methods. They are controlled in the block `%integration_par` in the file `cemos.par` by the parameter `method`:

`method`	numerical integration method
`0`	Fixstep (`FixStep`): fix timestep for discrete models
`-1,1`	Euler: Euler's method with time step adaptation[1]
`-2,2`	RK2: Runge-Kutta's-method 2nd order with time step adaptation, controlling the local error by 3rd order method
`-3,3`	RK4: Runge-Kutta's-method 4th order with time step adaptation, controlling the local error by the same method with the half timestep ('classical Runge-Kutta-method')
`-4,4`	RKCK: Runge-Kutta-Cash-Karp-method 4th order with time step adaptation, controlling the local error by an embedded fifth order method ('classical Runge-Kutta-method')
`-9,9`	Backward Euler: Euler's method with time step adaptation. The correction term is calculated for the end time of the integration step, thus, this method is more stable than the normal Euler method `-1,1`
`-99,99`	No Integration (`NoInt`): Special and fast handling of the model to evaluate time series and access external data without any effect to state variables

If the positive value is chosen the integration allows only positive values for state variables. This is useful if the state variables represent biomasses or concentrations.
If the negative value is chosen the state variables can also become negative.

The following table lists the methods and their control variables:

`method`	:	Euler	RK2	RK4	RKCK	BackEuler
`maxdelt`	:	+	+	+	+	+
`mindelt`	:	+	+	+	+	+
`relrate`	:	+				+
`relchange`	:	+				+
`accuracy`	:		+	+	+	

`FixStep (0)` and `NoInt (-99,99)` are controlled by `maxdelt`, only.
`maxdelt`: maximum time step (for fixstep **the** time step).
`mindelt`: minimum time step, if the time step adaptation fall short of it, **CEMoS** stops the simulation.
`relrate`: maximum allowed relative rate (maximum value for the fraction $|\dot{X}(t)|/|X(t)|$ of a state variable). If the fraction exceeds this value the time step will be reduced until the fraction reaches `relrate`. .
`relchange`:maximum allowed relative rate if the rate changes its sign. If in this case the fraction $|\dot{X}(t)|/|X(t)|$ exceeds `relchange` the time step will be reduced until their fraction reaches `relchange`. If only positive values for state variables are allowed (`method=1`),`relchange` is also valid if a state variable would become negative.

`accuracy`: maximum relative local error for the Runge-Kutta-methods; if the error exceeds this value the time step will be reduced. The error estimation is done by a method of higher order.
All methods are implemented in such a way that after a time step adaptation the integration tries to reach the maximum time step again as fast as possible. During a simulation the integration is started again for every time step, and not once for the whole simulation. Here the results of the previous time steps are the initial values for the next. This makes it quite more easy to guarantee the storing of equidistant results and shortens the simulation time. If only the result at the end of the simulation is of interest, the variables `maxdelt` and `outdelt` may be set to the difference of `endtime` and `starttime`.

6.1 Runge-Kutta methods

Let

$$\mathrm{y}' = \mathrm{f}(\mathrm{t}, \mathrm{y}(\mathrm{t})), \quad \mathrm{y}(\mathrm{t}_0) = \mathrm{y}_0$$

be the given initial value problem. For the correct solution Y_k at step k the solution at $\mathrm{k}+1$ are calculated as follows.

6.1.1 Runge-Kutta method 2nd/3rd order

$$\begin{aligned}
\mathrm{y}_{\mathrm{k}+1} &= \mathrm{y}_\mathrm{k} + \mathrm{a}_2 \qquad \text{predictor} \\
\mathrm{y}_{\mathrm{k}+1} &= \mathrm{y}_\mathrm{k} + \frac{1}{6}[\mathrm{a}_1 + 4\mathrm{a}_2 + \mathrm{a}_3] \qquad \text{corrector} \\
\mathrm{a}_1 &= \mathrm{hf}(\mathrm{t}_\mathrm{k}, \mathrm{y}_\mathrm{k}) \\
\mathrm{a}_2 &= \mathrm{hf}(\mathrm{t}_\mathrm{k} + \frac{1}{2}\mathrm{h}, \mathrm{y}_\mathrm{k} + \frac{1}{2}\mathrm{a}_1) \\
\mathrm{a}_3 &= \mathrm{hf}(\mathrm{t}_\mathrm{k} + \mathrm{h}, \mathrm{y}_\mathrm{k} - \mathrm{a}1 + 2\mathrm{a}_2)
\end{aligned}$$

h is the step size from step k to step $\mathrm{k}+1$.

The results are mostly better than of order 2 because the predictor method is of third order and the results of the corrector method are taken for further calculations. This method is recommended for large models because the model will be evaluated only three times per time step.

6.1.2 Runge-Kutta method 4th order

$$\begin{aligned}
y_{k+1} &= y_k + \frac{1}{6}[a_1 + 2a_2 + 2a_3 + a_4] \\
a_1 &= hf(t_k, y_k) \\
a_2 &= hf(t_k + \frac{1}{2}h, y_k + \frac{1}{2}a_1) \\
a_3 &= hf(t_k + \frac{1}{2}h, y_k + \frac{1}{2}a_2) \\
a_4 &= hf(t_k + h, y_k + a_3)
\end{aligned}$$

In **CEMoS** the predictor and the corrector method are both of fourth order. The corrector method works with a modified time step. This method has the highest accuracy but evaluates the model eleven times per time step. It is only recommended for small models.

h is the step size from step k to step $k + 1$.

For the theoretical background see Engeln-Müllges & Reuter (1988) and Kohlmeier (1995)

6.1.3 Cash-Karp Runge-Kutta method

To gain comparability to more modern integration methods a fifth order Runge-Kutta with an embedded fourth order method has been inegrated to **CEMoS** . This method is adaptive regarding the time step and needs only six evaluation of the model per time step. Details can be found in the code of **CEMoS** .

6.2 The concept of *operator-splitting*

Under the conditions of the example above it is assumed that the two parts (operators) $\dot{X} = \ldots$ and $\dot{Y} = \ldots$ need different integration methods . This might be necessary to reach a high numeric accuracy or to shorten the simulation time (in this underlying example these reasons are not given, but the technical implementation can be more easily described with such a small example) .

CEMoS supports the parallel handling of different integration methods with only minor effort during the model implementation.

The principle flow diagram for a model with two separated processes is shown in figure(6.1).

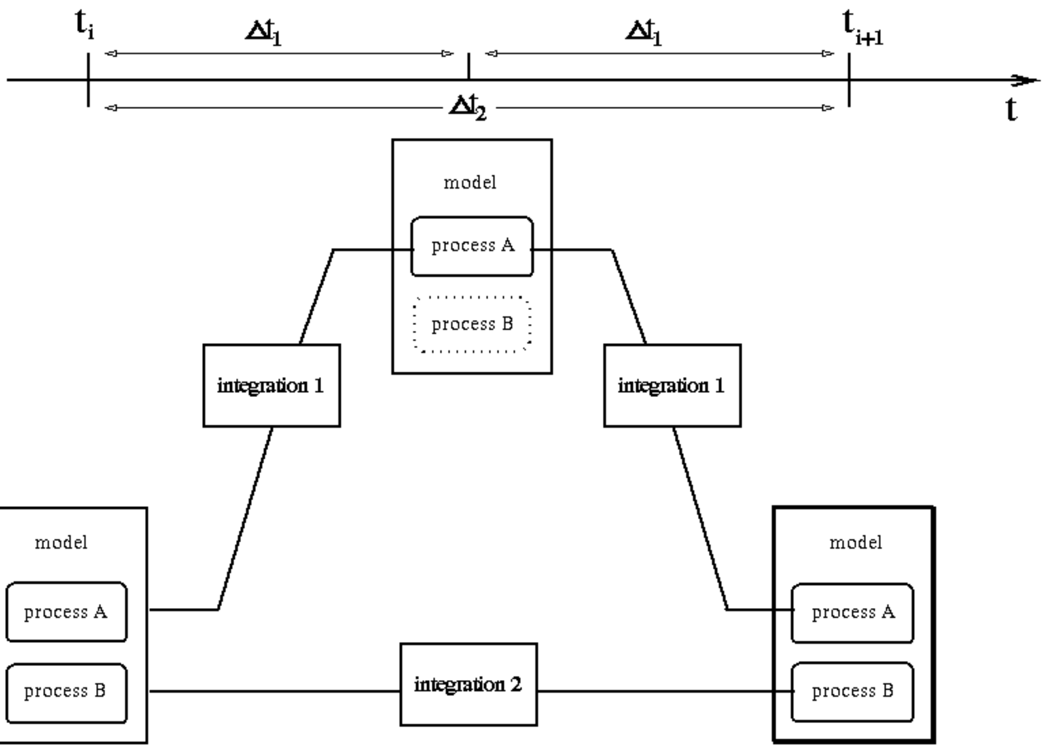


Figure 6.1: The principal progress of the different integrations during a time step using operator-splitting

Let t_s be the starting time and t_e the endtime, and let $\Delta t_1 = \frac{1}{2}\Delta t_2$ be the maximum time steps and let the simulation parameter `multi` be set to 2 (the other simulation-and integration parameter are of minor interest in this context).
The time interval $t_{i+1} - t_i$ is equidistantly divided into steps of length $\delta =$

$\min\{\Delta t_i\}_{i=1,2}$, where $t_{i+1} - t_i = \max\{\Delta t_i\}_{i=1,2}$.

If during the simulation a Δt_i becomes a multiple of δ, the integration method I_i is called and the global parameter `assign` is set to i for identification. Every integration method calls the whole model (`model()`) without distinction. In the model the parameter `assign` controls the execution of the processes. If only one integration method is active (`multi=1` or unset), `assign` is set to `-1`, to activate all processes.

The implementation of the example must be modified only slightly to activate the operator splitting:

```
#include "struct.h"
#include "par.h"
void model(void)
{
if (assign==-1 || assign==1) SX[1]=a*X[1]-b*X[1]*Y[1];
if (assign==-1 || assign==2) SY[1]=b*X[1]*Y[1]-m*Y[1];
}
```

The case `assign == -1` is valid during the very first execution of the model (without integration) to run through initializing routines and preparation f.e. of access to external data sets. In the file `cemos.par` now two blocks of integration parameters are needed. Additionally the parameter `multi` has to be set:

```
#include "main/model.def"

%simulation_parameters
starttime=0.0;
endtime=100.0;
outdelt=1.0;
model_dir main
multi=2;

%integration_par1
mindelt=1.e-15;
maxdelt=0.5;
accuracy=.00001;
method=3;

%integration_par2
```

```
mindelt=1.e-6;
maxdelt=2.0;
accuracy=.01;
method=2;

%store
X[1];Y[1];
```

Remark: If `multi` is set to 1 or is not set anywhere, all processes will be controlled by the parameters set in the block `%integration_par1`.

7 CEMoS Variables

In this section the automatically generated variables are explained.
They can be made available everywhere in the model code by the statement `#include <struct.h.>` and can be used to control the model or for debugging.

7.1 0D variable related information

Additional to the variables which can be stored
(`%0D_states`, `%0D_globals`)
the following variables are made available:

type of variable	number in	name in
state variables	`num_0D_states`	`_0D_state_names[1-number]`
global variables	`num_0D_globals`	`_0D_global_names[1-number]`

7.2 1D variable related information

Additional to the variables which can be stored
(`%states`, `%globals`,`%float_derived_from_states`'+variable')
the following variables are made available:

type of variable	number in	name in
state variables	num_states	state_names[1-number]
global variables	num_globals	global_names[1-number]
derived variables	num_float_derivs	float_derivs[1-number]
	num_int_derivs	int_derivs[1-number]

With these variables is it possible to loop over all variables, check their names and get a fast overview of the model run (especially during debugging).

7.3 2D variable related information

Additional to the variables which can be stored (%2D_states, %2D_globals,%2D_float_derived_from_states'+variable') the following variables are made available:

type of variable	number in	name in
state variables	num_2D_states	_2D_state_names[1-number]
global variables	num_2D_globals	_2D_global_names[1-number]
derived variables	num_2D_float_derivs	_2D_float_derivs[1-number]
	num_2D_int_derivs	_2D_int_derivs[1-number]

With these variables is it possible to loop over all variables, check their names and get a fast overview of the model run (especially during debugging).

7.4 Simulation- and integration related informations

The following variables and their related values are available:

starttime, endtime	starting and ending time of the simulation
storetime	optional, simulation time when storing of results starts
infotime	optional, a .info file is written after every infotime steps resp. simulatio
year, cycle	optional, starting year and cycle of the model (f.e for control of external data series)
sim_time, delt	actual time and time step during simulation
mindelt, maxdelt	minimum and maximum time step of the actual integration method
method, assign	actual integration method and process identifier
int_time, day_of_year	integer of sim_time and int_time mod cycle

Remark: The old names startim and storetim are still valid.

7.4.1 Info time

The value of the variable `infotime` is the time where a `.info` file is written. After that every `infotime` time steps `.info` files are written until endtime is reached. The `.info` files contain the values of all state variables at that time. Thus, a model can be restarted again by including this file in the cemos.par under the change statment as new initial conditions. This is useful if intermediate values of long runs are needed to restart the model resp. if long runs don't reach the endtime.

Example:
With `starttime`=3, `endtime`=100, `infotime`=20 info files are written at time 20, 40, 60, 80, 100. The same happens for `starttime`=0 or `starttime`=17.

7.5 Information related to the 1D model structure

These variables allow to activate or deactivate specified boxes, in contrast to the variable `assign`, which allows to attach specified processes to certain integration methods.

With these variables boxes can be taken out of the integration to get f.e. constant boundary conditions for other boxes (see also 5).

`n_comp`	number of model boxes
`l_comp`	number of active boxes
`i_com`	vector of length `l_comp`, containing the numbers (indices) of active boxes

A loop over all model boxes should be implemented in the following way:

```
for (i=1;i<=l_comp;i++)
  {
  box=i_com[i];
  SX[box]=.......
  .
  .
  .
  }
```

Normally this corresponds to a loop like `for (box=1;box<=n_comp,box++)`. But if f.e. in the file `cemos.par` the statement `boxes_not_active 1-2` is present, the construction above guarantees that boxes 1 and 2 are not evaluated and will keep their initial values.

7.6 Information related to the 2D model structure

g_comp1	number of grid columns
g_comp2	number of grid lines

A loop over all model grid cells should be implemented in the following way:

```
for (i=1;i<=g_comp1;i++)
  {
  for (j=1;i<=g_comp2;i++)
  {
  SAAA2D[i][j]=.......
  .
  .
  .
  }
```

7.7 Lists of variables and their handling

To have easy an comfortable control on the simulation process, the execution of the model and the storing of simulation results, **CEMoS** provides functions with a special syntax compared to C. The rules to be followed to gain the complete functionality of models in the **CEMoS** environment are given below.

Parameter and values are attached by lists. Here the **C**-syntax has been extended so that single elements of vectors can be changed. The statement

```
%change
t[1-3,5,8-10]={1,2,3,4,5,6,7};
```

```
t[1]=1,t[2]=2,t[3]=3,t[5]=4,t[8]=5,t[9]=6,t[10]=7.
```

It has to be remarked that 'inner' lists (lists in `[...]` or in `{...}`) are separated by commas while 'external' lists (f.e. `iswt1=1; iswt2=0;`) are separated by semicolons.

8 Storing in CEMoS

8.1 General

The model output will be stored every `outdelt`. If `maxdelt` exceeds `outdelt` `outdelt` will be set to `maxdelt` to avoid redundant storing. The storing normally starts at `starttime`. If the variable `storetime` is set storing starts at `storetime`.

If the simulation parameter `storestart` is set to 0 simulation results are stored every time **before** the integration starts, if `storestart=1` the simulation results are stored after the integration. To have control on this behaviour is helpful, when simulation results which are depending on time series shall be adjusted to those timeseries or measurement data.

All indexed variables including all globals and derived variables defined in the `model.def` can be stored.

The values will be stored for every `outdelt` set in `cemos.par`. At the beginning of a simulation at `starttime` resp. `storetime` all values will be stored for the first time. Then, all state variables will hold their initial values, all other variables will hold the values after one model evaluation. Such all time dependent values are evaluated at that time.

For every `outdelt` the state variables will hold the values after the integration, while all globals hold the values of the last model evaluation. The values which are stored depend on the integration method. In the case of FIXSTEP the states at time $\mathrm{t}+\Delta\mathrm{t}$ contain the integrated values from time t, the globals at time $\mathrm{t}+\Delta\mathrm{t}$ hold the values of time t because the model is evaluated at time t to determine the rates of change.

In the case of the Runge-Kutta-Method RK4 the result file contains the global values from the fourth model evaluation (see 6.1.2).

If the model is set up with operator splitting the global and derived variables contain the values of the last evaluation of the last integration method. Such, derived variables which are not used in the last integration are stored as zero because they are set to zero before starting the last integration.

8.2 Storing of 0D variables

All 0D variables which should be stored during simulation must be set in the `%0D_store` section of the `cemos.par`. This can be done in the following form

```
%0D_store
```

```
A;
B;
```

or shorter

```
%store
A;B;
```

Every statement should be closed by `;`. It is also possible to include a file which contains all variables to be stored in the correct syntax:

```
%0D_store
#include "store0.dat"
```

where the file `store0.dat` in this example contains the line

```
A;B;
```

8.3 Storing of 1D variables

All 1D variables which should be stored during simulation must be set in the `%store` section of the `cemos.par`. This can be done in the following form

```
%store
X[1];
Y[1];
X[2];
Y[2];
```

or shorter

```
%store
X[1];Y[1];
X[2];Y[2];
```

or again shorter

```
%store
X[1-2];Y[1-2];
```

Every statement should be closed by `;`. It is also possible to include a file which contains all variables to be stored in the correct syntax:

```
%store
#include "store.dat"
```

where the file `store.dat` in this example contains the line

```
X[1-2];Y[1-2];
```

It is not necessary that the variables are numbered consecutively.

8.4 Storing of 2D variables

All variables of types

```
2D_states
2D_globals
2D_real_derived_from_states+variable
2D_global_derived_from_states+variable
```

can be stored during simulation. The output step is controlled by the general simulation setup. All 2D variables which should be stored during simulation must be set in the `%store_2D` section of the `cemos.par`.
This can be done in the following form:

```
%store_2D
AAA2D;
BBB2D;
ggg2dAAA2D;
```

Here the first big difference to the storing of 1D variables is obvious: no indices are given.
This results in the variables being stored for the complete grid, which makes the handling of stored results much easier. The next restriction is motivated by the fact that **MoViE** is currently not able to handle 2D variables directly.
Therefore, the `%store_2D` section only evaluated if the output format `netCDF` with the extension `.nc` is selected.

When storing in the `.outc` format, the `%store_2D` section is simply ignored.

Remark: 2D variables can only be stored to `netCDF` outputs with the extension `.nc`

The default manner of storing 2D variables is the 'matrix' approach which means that data are store in the sorting `[line-index] [row-index]`. This leads to a 90 degrees rotation when result files are visualized with programs like `ncview` or `VisIt`. To remedy this it is possible to get the 'co-ordinate' approach of storing `[row-index] [line-index]` (comparable to x- and y-axes coordinates) by conditional compilation (see 4.4.6) which is activated by inserting the line `%setup NC_FLIP` in the file `model.def`. After re-compiling the model 2D data will be stored in the co-ordinate oriented way and can directly be processed with the above mentioned and other tools.

9 C-Extensions by CEMoS

9.1 The type real

By **CEMoS** the one type of variables is added to the standard `C` variable types.

This type `real` makes it easy to control the complete numerics of the model and the integration methods. It also provides safe interfaces between the internal numerics and the storing of simulation results.
Controlled by the identifier `\%numerics` in the file `model.def` all variables of the type `real` will be set to the following types:

`%numeric single` means a `#typedef float real`

`%numeric double` means a `#typedef double real`

`%numeric long double` means a `#typedef long double real`

If no line with the identifier `%numerics` appears in the file `model.def` the default definition `#typedef float real` is used.
With this construction the variable type `real` is available in every file which include the the header file `struct.h`.

9.2 Vectors, matrices and tensors

The following data structures are available for the main basic types (`int, float, double`) and for `real`, as well:

```
int     *ivector(first, last);
float   *fvector(first, last);
double  *dvector(first, last);
real    *vector (first, last);
```

```
int     **imatrix(first_row, last_row, first_col, last_col);
float   **fmatrix(first_row, last_row, first_col, last_col);
double  **dmatrix(first_row, last_row, first_col, last_col);
real    **matrix (first_row, last_row, first_col, last_col);

int     ***i3tensor(first_row, last_row,
                    first_col, last_col,
                    first_lay, last_lay);
float   ***f3tensor(first_row, last_row,
                    first_col, last_col,
                    first_lay, last_lay);
double  ***d3tensor(first_row, last_row,
                    first_col, last_col,
                    first_lay, last_lay);
real    ***r3tensor(first_row, last_row,
                    first_col, last_col,
                    first_lay, last_lay);
```

The variables holding the indices (`first`, `last`, `first_row`, `last_row`, `first_col`, `last_col`, `first_lay`, `last_lay`) are of type `long`.

The underlying functions provide an optimized allocation of memory for the a.m. structures.
Additionally, the access to any element of the more complex types `matrix` and `tensor` is very much faster with these constructions than by simply allocation memory with standard `C` methods. Each line of a `matrix` is identified by a pointer to a `vector` and each `matrix` inside a `tensor` is identified by a pointer to a `matrix` with again `vectors` a sub-structures. This avoids time consuming pointer arithmetics during program execution by directly working with pointers to the memory addresses.
The prototypes for these functions are automatically available by including the file `struct.h`, such the use is easy.

Example:

```
#include "struct.h"
int *example_vector;
float **example_matrix;
real ***example_tensor;
/* Allocate a vector of integers with available indices from 7 to 25 */
example_vector   = ivector(7,25);

/* Allocate a matrix of floats with
```

```
available indices from 3 to 17 and from 0 to 25 */
example_matrix   = fmatrix(3,17,0,25);

/* Allocate a 3tensor of reals with
available indices from 0 to 10, from 5 to 25 and from 1 to 100 */
example_tensor   = r3tensor(0,10,5,25,1,100);
...
```

In case of allocation errors all functions will stop the execution of the program and will give an error message. More details (and some additional stuff) can be found in the source code of **CEMoS** in the files `constructs.h` and `constructs.c`

10 Data structures of CEMoS simulation outputs

CEMoS actually provides three different output formats for storing simulation results:

- `.outc` default binary format. The `.outc` format allows names of 80 characters.
- `.outa` ASCII format. The `.outa` format allows names of 80 characters.
- `.nc` is the **NetCDF** format (see Rew *et al.* , 1997)). It is mainly used to make **CEMoS** simulation results available in environments where no **Mo ViE** is used. **NetCDF** files can be directly read by **xmGrace** . They are containing the same amount of information as the `.outc` files. Information about the content of a `.nc` file can be read by typing the command `ncdump <filename>` in a terminal window. Information about the usage of that program will be displayed by simply typing `ncdump`.

Remark: `ncdump` works for all `.nc` files, even if they were not stored by **CEMoS** .

The complete **NetCDF** interface is described in (Rew *et al.* , 1997), the part of the **NetCDF** interface used in **CEMoS** and **MoViE** is described in (Kohlmeier & Hamberg, 2023).
MoViE supports all three file types (`.outb`, `.outc` and `.nc`). The following gives a description of the `.outc` files structure.

10.1 Data structures of the files `mymodel/xxx.outc`

The `.outc` file starts with structural information given in different formats:

(1) total number of stored variables (`nvars` automatically derived by **CEMoS**)

(2) start time of the simulation (`start` read from `cemos.par`, normally 0 for January 1 of year (see (6))

(3) end time of the simulation (`endtim` read from `cemos.par`)

(4) maximum timestep of the simulation (`maxdelt` read from `cemos.par`)

(5) simulation time between two outputs (`outdelt` read from `cemos.par`)

(6) year, where the simulation starts (`year` read from `cemos.par`, stored is max(0,year))

(7) length of a model's year (`cycle` read from `cemos.par`, normally 360 days representing 12 months of 30 days.)

(1) is stored as **C** `unsigned int` (4 Bytes).
(2) - (7) are stored as **C** `floats` (4 Bytes).

(8) relative path to the main model's directory, stored as string of 80 ASCII characters (`model_dir` read from `cemos.par`)

(9) This header is followed by a set of `nvars` strings of each 80 ASCII characters. These strings are containing the identifier of a variable and the box number it is stored for (f.e. `'p1c(137)'` indicates p1c of box 137 being stored.)

(10) After this block of strings (which may be some ten kilobytes long) the numerical output of the simulation is stored as follows:
One **C** `float` (4 Bytes) is stored in binary format for each variable-box combination appearing in the string block (9) is stored for all simulation times between `starttime` (2) and `endtime` (3) that are multiples of `outdelt` (5). This block may have a length of some megabytes.

11 The mystic recalc_globals statement

The setting `recalc_globals=1` forces a recalculation of the model without integration to get `global` and `global_derived_from_states` variables recalculated before storing simulation results. This is needed if the integration

method calls the model at intermediate interpolation points. Normally the values of global variables are calculated at the last interpolation point and therefore this value is stored. This is a well known problem with accurate integration methods. Even if the differences between the values at the end of the step and the values at some intermediate points is not serious, the results might be misinterpreted (f.e in budget computation where total mass conservation is expected). The differences increase if the system is non autonomous (directly dependent from the actual time, f.e. in the case of a forcing function). **CEMoS** provides the possibility to recalculate the values at the end of the step with the statement `recalc_globals=1;` in the file `cemos.par` (the default setting is `recalc_globals=0`).
In this case the model is called once again to calculate the global variables at the sampling point (with the actual simulation time) but without changing state variable values.

This is also very helpful, if the model is run with different integration methods on different time steps (operator splitting), and not all derived variables are affected by all integration methods, but shall be stored for diagnostic purposes.

Remark: No differences in the values of state variables shall occur with or without setting `recalc_globals=1` because the integration is not affected by this!!

Under some practical circumstances differences might occur:

- A state variable is directly set within the model to a new value (this is not allowed in the context of differential equation but may occur if state variables a misused, f.e. for diagnostic purposes.). During the recalculation such a state variable gets a new value which may force the integration routine to a slightly different behavior. If f.e. the state is set to a total different value the integration adapts the time step and this may lead to differences in all state variables.

- A global variable which determines the rate of a state variable is calculated at the wrong position in the model code. Globals are initialized with zero by **CEMoS** . Such no warning is given if a global is used before setting it to its right value. Because **CEMoS** passes through the model once before starting the simulation, normally no problems occur. But if the global itself is determined by the value of another state variables things go wrong. The following model will show the effect:

  ```
  #include "struct.h"
  void model(void)
  {
  ```

```
SX[1]=a[1]*X[1];
SY[1]=Y[1];
a[1]=Y[1];
}
```

The state variables `X[1]`,`X[2]`,`Y[1]`,`Y[2]` get all the initial value 1. The global variable `a[1]` is initialized by zero (**CEMoS** does it). The time step is fixed to 1. The results are taken from a simulation with a second order Runge-Kutta integration which has an intermediate calculation point:

	recalc=0	recalc=1
Time	x(1)	x(1)
0.00000	1.00000	1.00000
1.00000	2.50000	2.50000
2.00000	18.12500	16.56250

12 Handling of data files

12.1 Reading csv-files

CEMoS provides some routines for reading `csv`-files (comma separated values). The files to be read must habe the following structure:

The first line must be a header line. This is only for convenience to describe the following lines. No more header lines are allowed! The following arbitrary number of lines contain the data. The first column of a line must be a time stamp in decimal, the following columns contain the data. The columns must be seperated by a comma. A common data file is given by f.e.

```
time,value1,value2
1.0,5.5,7.0
2.0,5.4,7.6
...
```

All columns must have the same length, the time stamps must be in increasing order!

Such a file can be read by the command `read_csv`. The call of `read_csv` needs three arguments. The first one is a string containing the name of the data file realtive to the code file where it is called. The second argument contains the column to be read and the third the name of a vector the data should be stored. The length of the vector must match the number of data. To get the correct number, `read_csv` can be called with column 0 and and dummy, the return value is the number of data sets.

Reading the data must only be done once during simulation time if the vectors are declared as `static`.

12.2 Interpolation of data

The data must not be equidistant and the number must not match the simulation time. Thus it is often neccessary to interpolate them. **CEMoS** provides the routines `lin_int` and `no_interpolation`. For customizing the interpolation see 12.4.

12.2.1 Linear interpolation

`lin_int` needs three arguments. The first one is the vector containg the time stamps, the second the vector containing the corresponding values and the third is the pointer to the number of data. `lin_int` makes a linear interpolation for the actual `sim_time` between to neighboured data points, while `no_interpolation` holds the last value until `sim_time` matches the next time stamp.

If the **CEMoS** variable `cycle` is set, the data are repeated after `sim_time` reaches a multiple of cycle. and `no_interpolation`.

For a better control of the interpolation a second routine `lin_int2` exist which is called by four arguments where the first (additional) one is the time at which the interpolated value should be calculated. This is useful if data and simulation time have an offset. If `lin_int2` is called with `sim_time` as first argument both routines behave equal.

12.2.2 Step functions

`no_interpolation` needs four arguments. `no_interpolation` needs as first argument either `__RIGHTSLOPE` or `__LEFTSLOPE`, the second one is the vector containg the time stamps, the third the vector containing the corresponding values and the fourth is the pointer to the number of data.:

`no_interpolation` holds the last value until `sim_time` matches the next time stamp.

For a better control of the interpolation a second routine no_interpolation2 exist which is called by five arguments where the first (additional) one is the time at which the value should be evaluated. If no_interpolation2 is called with sim_time as first argument both routines behave equal.

12.3 Example

Reading data from a file called data.csv which contains the following lines:a

```
ETW(1)/tim/,ETW(1)/val/
1,8
10,5
30,-2
63,5
120,11
149,15
185,17
210,22
239,17
247,16
253,16
280,15
310,8.3
338,-1
345,-2
```

The model code (model.c):

```
#include "struct.h"
#define datafile "./main/data1.csv"

real *dummy=NULL;
static real *time,*value;
static int init=1;
static int num=0;

void model(void)
{
if (init==1)
{
   init = 0;
  /* Determing number of values    */
  num=read_csv(datafile,0,dummy);

 /* Preparing vectors */
   time = vector(1,num);
   value= vector(1,num);

 /* Reading values */
 if (read_csv(datafile,1,time) != num) nrerror("Error in datafile");
 if (read_csv(datafile,2,value) != num) nrerror("Error in datafile");
}  /* end if (init==1) */

/* Model code */

/* Linear interpolation at sim_time */
ETW[1] = lin_int(time,value,&num);

/* Step function at sim_time */
ETW[2] = no_interpolation(__LEFTSLOPE,time,value,&num);

/* Step function at sim_time */
ETW[3] = no_interpolation(__RIGHTSLOPE,time,value,&num);

}
```

The time stamps are stored in the vector `time`, the data in the vector `value`. Both are initialized as vector (see 9.2) starting at index 1 and ending at index `num`. The number of data points `num` is determined by the call of `read_csv` with column 0 and a dummy pointer.

The time stamps and values are read by two calls of `read_csv` with the corresponding column number. If some thing si wrong with the number of data, an error is given.

The reading of data is done only once (`init==1`). In this case the vectors containing the data and the number of data must be defined as `static`.

During the simulation the linear interpolated data are read into the global variable `ETW[1]`, the step function is read into `ETW[2]` (`__LEFTSLOPE`) and `ETW[3]` (`__RIGHTSLOPE`). The results are shown in figure 12.1.

12.4 Tricks

The interplay of the data time steps, the integration step, the integration method and the outdelt is always a complicated task.

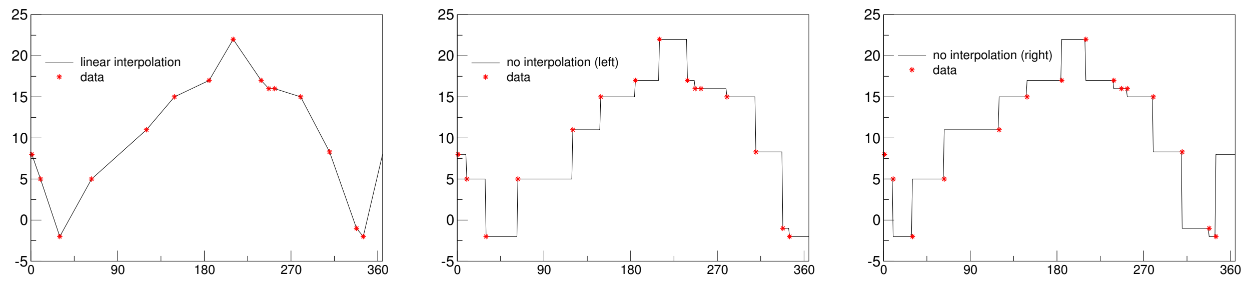


Figure 12.1: The data (red stars) and the result from the linear interpolation, the left hand side step function (mid) and the right hand side step function (right).

The interpolation result depends on

- `storestart` (8)

  If the simulation results are stored before integration `storestart=0` the last time step is taken into account for the interpolation, if `storestart=1` the interpolation starts at the first data point

- `recalc_globals` (11)

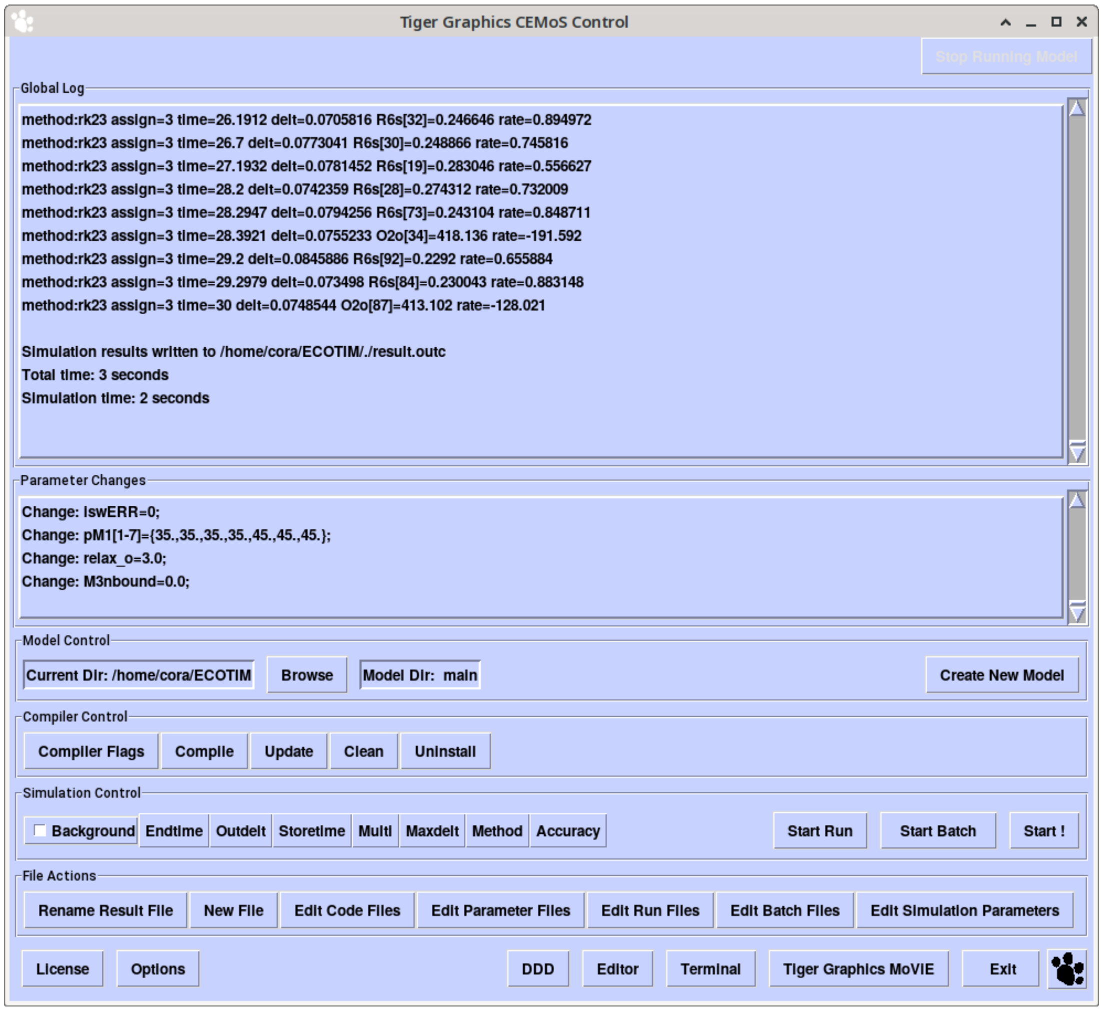
  If an integration method with intermediate time steps is used, the results differ depending on the setting of `recalc_globals`. If it is set to 0 the data are interpolated at the last intermediate time step of the integration, otherwise at the end of the integration step. It is recommended –to avoid this problem– that reading the data is done in a mode (6.2) with a fixed time step (6). Sometime it is useful to prepare a special mode for reading the data without integration (`method=99`).

13 Starting with CEMTK

To start **CEMTK** go to the directory where the model (the file `cemos.par`) is located or where a new one should be created and give the command

```
cem
```

The **CEMTK** window appears (figure 13.1).



```
method:rk23 assign=3 time=26.1912 delt=0.0705816 R6s[32]=0.246646 rate=0.894972
method:rk23 assign=3 time=26.7 delt=0.0773041 R6s[30]=0.248866 rate=0.745816
method:rk23 assign=3 time=27.1932 delt=0.0781452 R6s[19]=0.283046 rate=0.556627
method:rk23 assign=3 time=28.2 delt=0.0742359 R6s[28]=0.274312 rate=0.732009
method:rk23 assign=3 time=28.2947 delt=0.0794256 R6s[73]=0.243104 rate=0.848711
method:rk23 assign=3 time=28.3921 delt=0.0755233 O2o[34]=418.136 rate=-191.592
method:rk23 assign=3 time=29.2 delt=0.0845886 R6s[92]=0.2292 rate=0.655884
method:rk23 assign=3 time=29.2979 delt=0.073498 R6s[84]=0.230043 rate=0.883148
method:rk23 assign=3 time=30 delt=0.0748544 O2o[87]=413.102 rate=-128.021

Simulation results written to /home/cora/ECOTIM/./result.outc
Total time: 3 seconds
Simulation time: 2 seconds
```

```
Change: IswERR=0;
Change: pM1[1-7]={35.,35.,35.,35.,45.,45.,45.};
Change: relax_o=3.0;
Change: M3nbound=0.0;
```

Current Dir: /home/cora/ECOTIM

Model Dir: main

Figure 13.1: Main window of **CEMTK** .

14 Controlling the model setup

14.1 [Stop Running Model] (Linux only)

A running simulation can be stopped. This is only possible if the model is not running in background.

14.2 Info window 1

Within this window all information during compilation, simulation etc is shown.

14.3 Info window 2

Within this window all parameter changes of the actual simulation are shown. For information on parameter changes see 5.

14.4 [Current Directory] and [Model Directory]

Here the current directory is shown. It can be changed by **[Browse]** . If f.e. **CEMTK** is started from anywhere and your model directory is located in `$HOME/mymodel`, then go the directory `$HOME` and mark `mymodel`. Pressing **[OK]** resets the current directory to `$HOME/mymodel`. If there a file `cemos.par` exists **CEMTK** will read all information including the model directory from it. If the file `cemos.par` doesn't exist only **[create New Model]** is active.

14.5 [Create New Model]

If neither a file `cemos.par` nor a `main` directory exists in the current directory an "empty" model will be created there. This means that a file `cemos.par` is created with default settings and a model directory main is created which includes the files `model.def`, `par.def` and `model.c`. This files may be edited by **[Edit Simulation parameters]** , **[Edit Code Files]** and **[Edit Parameter Files]** . How to built a model under **CEMoS** see 4.2.

14.6 [Compiler-Flags]

A menue opens holding compiler flags. The flags affect in general the compilation of the model files and the **CEMoS** files (some flags are omitted for the building of internal files and preprocessing). The evaluated compile flags for every file can be traced in the info window during compilation.

In the upper part useful combinations of flags are predefined:

default:	`-O2 -Wall`
debug:	`-g -pedantic -D__verbose__ -Wall -Wredundant-decls`
gprof:	`-pg -static`

If `-pg -static` is selected the model will also be linked with `-pg -static`.

In the lower part the compiler flags can be changed manually.

14.7 [Compile]

Starts the compiler. The model in the model directory will be compiled from scratch, all old object-files will be deleted before compilation. After the successful compilation the objects will be archived in a file `cemos.a` automatically and all **CEMoS** stuff will be deleted. The executable file `cemos-model` will be moved to the directory above (current directory). If the all files are compiled with `-g` , the **CEMoS** stuff and all object files will remain in the directory. Otherwise intermediate files will be removed after compilation.

14.8 [Update]

After modifying the model code it is not necessary to compile the whole model. **[update]** compiles all `.c`-files which are younger than the archive file `cemos.a` and renews the archive.

Note: If parameter files have been modified (except the `%change` statements) the whole model has to be compiled.

14.9 [Clean]

Removes all `.o`-files, `core`-files %-files and all **CEMoS** stuff from the model directory. Only the user defined model files remain. This is automatically done before compilation.

14.10 [Uninstall] (optional)

If the model contains an install script to link model code from other locations into the model directory this links will be removed.

15 Controlling the simulation

15.1 [Background]

If active the simulations will start in the background, no output is displayed in the info windows, but in the `.log`-file (see **[Start!]**).

The following variables may be changed within **CEMTK** . All settings are valid during a **CEMTK** session but will not be saved in the file `cemos.par`. For the meaning of these variables see 5.

15.2 [Endtime],[Outdelt], [Storetime]

Pull down menus which holds the values from the cemos.par in the first line.

15.3 [Multi]

Only valid if the model has been set up for operator splitting
(see 5). Depending on the setting of `Multi` the buttons **[Maxdelt]** , **[Method]** , **[Accuracy]** will change their behavior. They can be set for each integration method independently.

15.4 [Maxdelt]

If `Multi` is set to one a pull-down menu occurs which contains the setting of `Maxdelt` from the `cemos.par` in the first line.
If `Multi` is set to n>1 n menus occur for the different integration methods. Each contains the setting of `Maxdelt` in the file `cemos.par` in the first line. If **CEMTK** is started with `Multi=1` from the `cemos.par` all methods are set two the `Maxdelt` of the first integration method (`%integration_par1`). If the file `cemos.par` contains `Maxdelt` for n methods `Maxdelt` will be read from it for every method into **CEMTK**
.

15.5 [Method]

The numerical integration methods can be chosen. Concerning the role of **[Multi]** see **[Maxdelt]** , concerning the numerical methods see 6.

15.6 [Accuracy]

This setting is only valid for the Runge-Kutta methods. Concerning the role of **[Multi]** see **[Maxdelt]** , concerning the numerical methods see 6.

15.7 [Start !]

Starts the simulation. A window appears where the result filename can be chosen. By two radio buttons the output format can be selected to be `outc` or `.nc`. If **[store**

as result] is chosen the simulation results will be stored in the file `result.outc` resp. `result.nc` and the values of all state variables of the last simulation steps will be stored in `result.info` and all output from the simulation will be stored in `result.log`.

Note: CEMTK will overwrite an existing file `result.*` without a warning.

If a filename is specified (f.e. `myresult`) the simulation results, the log and the info will stored under the specified name (`myresult.outc`, `myresult.log` and `myresult.info`) in the current directory. If an `.outc` file with the same name (`myresult.outc`) already exist **CEMTK** will not start the simulation. In this case another name can be chosen or the result file (`myresult.outc`) must be renamed (see **[rename result file]**) or removed.

If **[Background]** is active no information of the simulation will be shown in the info windows and the simulation starts in the background. Otherwise all output is displayed in the info windows and additionally stored in the `.log` file. In this case the simulation can be stopped (Linux only).

15.8 [Start Run] -evaluating cin-files

This is an alternative to start a simulation.
A window appears where a `*.cin` (`cin` stands for **CEMoS** -initialisation) file can be chosen. Files of this type may contain any subset of parameters and controls which normally appear in the `cemos.par`. These settings from the selected `*.cin` file will override the settings which are stored in the `cemos.par`. The `*.cin` files are processed by the `C`-preprocessor before the simualtion starts and may thus contain all useful preprocessor directives which may be needed (e.g. `%include`). If e.g. the `ABC.cin` is selected, the according result file will get the name `ABC.outc`.

15.9 [Start Batch]-evaluating bat-files

This is an alternative to start a simulation.
A model simulation will be started for every line in the selected `.bat` file. The `.bat` file can contain parameter changes for an arbitrary number of variables. This file overrules the settings in the `.def` files. The settings in every line must be separated by `;` and every line must be closed by `;`. The result files will be named automatically.
F.e.: If the file `sensitivity.bat` contains the following lines:

```
a=1.0;b=0.0;
a=0.0;b=1.0;
```

```
a=1.0,b=1.0;
```

three simulations are started where the variables a and b (which must be defined in any `.def`-file, get these values. The files are named `a10b10.outc`, `a00b10.outc` and `a10b10.outc` and are stored in a directory named similar to the `bat`-file (without extension).
Another possibility is to define a range of values for a parameter for sensitivity analysis.
If the file `sensitivity.bat` contains the following lines:

```
%series
sigma:5.0:15.0:1
```

eleven simulations will be run, the parameter `sigma` getting values from 5.0 to 15.0 in steps of 1. The results will be written to a directory `sigma_sens` and the result files will be named automatically, again. Additionally, a file (`sigma.lst` in this example) is written, that holds the names of all result files produced in this sequence. This file can be used to get an animated phase plot running through this sequence in **MoViE** (see Kohlmeier & Hamberg, 2023).

Also `*.cin` files (see 15.8) may be included to have different complex scenarios run from a batch file. In this case the batch file may look as follows:

```
#include A1.cin
#include A2.cin
#include A3.cin
```

Each line will result in one simulation run using the setting from the respective `*.cin` file. The according result files will get the names `A1.outc`, `A2.outc` and `A3.outc`.
Remark: Only one parameter will be evaluated for one sequence. Any more parameters will be ingnored!

Remark: This functionality may not work in background on all machines.

16 Editing the model files

16.1 [Rename Result File]

A chosen result file may be renamed, the `.log`-file and the `.info`-file will be renamed, too, if existing. If a file with the new name already exists **CEMTK** gives an warning in the info window and the name stays unchanged. The existing file will not be overwritten.

16.2 [Edit Code Files]

A file select box occurs and shows all `.c`-files in the model directory. After selecting a file it can be opened by **[open]** . A text editor opens in the background and the code may be modified. After saving it the model must be updated or compiled.

16.3 [Edit Parameter Files]

A file select box occurs and shows all `.par`-files in the model directory. After selecting a file it can be opened by **[open]** . A text editor opens in the background and the parameter file may be modified. After modifications in the `#change` block of the file the model can be started without recompiling. Any other changes will be valid only after recompilation.

16.4 [Edit Run Files] – the cin-files

Run files `.cin` can be edited (see also **[Start Run]**).

16.5 [Edit Batch Files] – the bat-files

Batch files `.bat` can be edited (see also **[Start Batch]**).

16.6 [Edit Log Files] – the log-files

A file select box occurs and shows all `.log`-files. These files are only for information.

16.7 [Edit Simulation Parameters] – the cemos.par

The file `cemos.par` is opened in a text editor. Here all simulation parameters can be modified. This editor blocks **CEMTK** . After saving the changes the editor must be left to continue with **CEMTK** . The reason for this is that after quitting all changes will read again by **CEMTK** and all buttons will be updated. This means that changes made in the cemos.par will overrule the actual settings in **CEMTK** .

17 Misc

17.1 [Options]

Here different setting for **CEMTK** can be selected:
[Editor]
The editor used with **CEMTK** can be set. If no editor is set **CEMTK** uses `kwrite` for Linux and `WordPad` on Windows.
Remark: When running on Windows it turned out to be very helpful to select the desired editor because the path to `wordpad.exe` is changing from version to version ;-).

[Select Button/Label Font]
With a font selector the font for buttons and labels can be selected from the fonts available for the running window system.

[Select Info Window Font]
With a font selector the font for the output in the information windows can be selected from the fonts available for the running window system.

[Output Type]
By two radio buttons the default output type for simulation runs can be selected: `.outc` or `.nc` according to section 10.

All settings will take effect after **[Apply & Exit]** . **[Cancel]** closes this window ingnoring any changes been made.

17.2 [DDD] (Linux only)

Starts the debugger `ddd`. For further information see man page of `ddd`. The settings for simulation parameter are read from `cemos.par` and not from the **CEMTK** menus when the model is debugged.

17.3 [Insight] (Linux only)

Starts the debugger `insight` if installed. For further information see man page of `insight`. The settings for simulation parameter are read from `cemos.par` and not from the **CEMTK** menus when the model is debugged.

17.4 [Editor]

Opens the selected editor with an empty unnamed file in the model directory. This file can be saved under the desired filename.

17.5 [Terminal] (Linux only)

Opens a terminal window. The working directory is set to the model path.

17.6 Tiger Graphics MoViE]

Starts the graphic tool **MoViE** if installed. For further information see Kohlmeier & Hamberg (2023).

References

Engeln-Müllges, G., & Reuter, F. 1988. *Formelsammlung zur Numerischen Mathematik mit Standard FORTRAN 77-Programmen*. 6. auflage edn. BI Wissenschaftsverlag.

Hamberg, F. 1996. *CEMoS, eine Programmierumgebung zur Simulation komplexer Modelle*. Diplomarbeit, Fachbereich Mathematik, Carl von Ossietzky Universität Oldenburg.

Kohlmeier, C. 1995. An alternative integration method for Sesame. *In: Progress Report 1994 of the European Regional Seas Ecosystem Model II*. EU Contract No. MAS2-CT92-0032-C.

Kohlmeier, C, & Hamberg, F. 2023. *MoViE 5.0*. http://www.staff.uni-oldenburg.de/cora.kohlmeier/software/movie.pdf.

Rew, R., Davis, G., Emmerson, S., & Davies, H. 1997. *NetCDF User's Guide for C Version 3*. Unidata Program Center.

Ruardij, P., Baretta, J., & Baretta-Bekker, J.G. 1995. SESAME , a software Environment for Simulation and Analysis of Marine Ecosystems. *Netherlands Journal of Sea Research*, **33 (3/4)**, 261–270.