# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>3</td>
</tr>
<tr>
<td>Temperature Solver</td>
<td>4</td>
</tr>
<tr>
<td>Skeleton</td>
<td>100</td>
</tr>
<tr>
<td>Posther</td>
<td>104</td>
</tr>
<tr>
<td>B-Plot</td>
<td>114</td>
</tr>
<tr>
<td>Batch Mode</td>
<td>121</td>
</tr>
</tbody>
</table>
Temperature Solver

Temperature Solver Overview

The temperature solver of THERMISOL is the main module of this package. It is based on the ESATAN language and may be linked with THERMICA through the skeleton module of THERMISOL. In the nodal method, also called "lumped parameter method", the problem is described by a set of nodes, each with:

- thermal properties (temperature, thermal capacitance, area, etc.)
- exchanges with the other nodes
- internal and external powers.

Refer to Chapter "Nodal Description Module" in the THERMICA User Manual to get a detailed explanation of the nodal method.

The network properties are as follows:

<table>
<thead>
<tr>
<th>Network properties</th>
<th>Symbol</th>
<th>Type</th>
<th>Entity type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node status</td>
<td>NS</td>
<td>Character</td>
<td>Nodal</td>
</tr>
</tbody>
</table>
| 'D' for diffusive
| 'B' for boundary
| 'X' for inactive                     |        |          |             |
| Model status                        | MS     | Character | Nodal       |
| 'A' for active
| 'X' for inactive                     |        |          |             |
| Temperature                         | T      | Real     | Nodal       |
| Capacitance                         | C      | Real     | Nodal       |
| Area                                | A      | Real     | Nodal       |
| Epsilon                             | EPS    | Real     | Nodal       |
| Epsilon with wavelength dependency  | EPSWLB | Real Array | Nodal       |
| Alpha                               | ALP    | Real     | Nodal       |
| Solar flux                          | QS     | Real     | Nodal       |
| Planet IR fluxes                    | QE     | Real     | Nodal       |
| Planet albedo fluxes                | QA     | Real     | Nodal       |
| Internal fluxes                     | QI     | Real     | Nodal       |
Residual fluxes | QR | Real | Nodal
Conductive coupling | GL | Real | Coupling
Radiative coupling | GR | Real | Coupling
Radiative coupling with wavelength dependency | GRWLB | Real Array | Coupling
One-way linear coupling | GF | Real | Coupling
Conductive coupling status | GLS | Character | Coupling
Radiative coupling status | GRS | Character | Coupling
Radiative coupling with wavelength dependency status | GRWLBS | Character | Coupling
One-way linear coupling status | GFS | Character | Coupling

Only T, C, QS, QE, QA, QI and QR are used in the solution routines (energy balance for each node).

Wavelength dependent properties are available only if it has been enable in the $ENTITIES block by declaring a wavelength band discretization.

Each datum can be time-dependent or can depend on any criterion defined by the user. Here, the objective is to compute the node temperatures for steady-state or time-dependent cases. In this approach, the thermal problem is a mere numerical integration. Broadly defined, THERMISOL is a numerical integrator dealing with steady-state and transient thermal equations.

**Temperature computation methods**

Some classical numerical methods are provided (including Newton Raphson and Crank Nicholson schemes), improved by an automatic management of time steps and the management of different time steps within a thermal network. Some other functions are also available to compute heat transfers or to make dynamic changes during the solution.

These routines have been intensively validated and used on real projects such as: MEX, Pleiades, Ariane5, Arabsat, VEX, Melfi, Intelsat10, Immarsat4, HotBird8, Metop, Amazonas, Anik, W3A, ISS, Gaia, LHP, etc.

**Input data**

The selected language format is based on the European standard language ESATAN. Since its creation, the THERMISOL language, as the ESATAN one, has evolved so they may have differences. However, even if new specific THERMISOL features have been developed, the compatibility with ESATAN input files is largely maintained.
The input data are described in text file(s), with a Fortran-like syntax to define the nodal network and a Fortran-like language to program any arbitrary behavior (time-dependent phenomena, temperature-dependent data, etc). The input data are translated into a Fortran code, compiled and linked with a computation library; this produces an executable program which generates the solution. The selection of a Fortran-like language provides a very flexible approach and is a solution to complex problems. The Fortran used is called "Mortran" and provides simple access to nodal entities, couplings or any THERMISOL specific datum.

Free and Fixed formats

In order to avoid Fortran compilation problem, the THERMISOL pre-processor automatically shift the beginning of the lines to the 6th column (except in the cases described bellow) and cut them so they don't exceed 72 columns.

<table>
<thead>
<tr>
<th>The automatic column positioning requires the following restrictions on the FORTRAN language:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The character used to specify that a line is the continuity of the previous one is ' &amp; ' (it is automatically positioned to the 5th column, as specified by the FORTRAN)</td>
</tr>
<tr>
<td>- A line beginning with a series of number written in the first columns is interpreted as a tag and is not modified by the preprocessor</td>
</tr>
</tbody>
</table>

However, for compatibility with older version of THERMISOL or with ESATAN files which are in fixed format (meaning that all lines shall start at column 6 and a first character on column 5 indicates a line continuation), THERMISOL also checks the meaning of a line when it starts with a character on the 5th column. If the instruction does not make sense, it is automatically interpreted as the continuation of the previous line. This behavior is called the MIXED format (meaning that it handles both free and fixed formats). In rare cases, the mixed interpretation of the format may fail to interpret correctly a fixed format input file, when this one is written with lines starting on the 5th column that could be interpret as they are (this is not very smart to write this kind of input files, but it is allowed by the FORTRAN77 fixed format). For this reason, it is possible to modify the interpretation of the input file by specifying that a strict fixed format shall be interpreted only. The option for free format only is also available even if it may not be required.

The file format may be specified at the beginning of the file with the instruction: !FORMAT = MIXED (default value) !FORMAT = FIXED !FORMAT = FREE

Structure of the input file

A model is a combination of different blocks of instructions. Some are used to define the thermal network, others to declare constants, variables or arrays, and finally some are executive instructions called at different moments of the solver execution.
Usually, the whole process is launched with only one parameter: the name of the main input file. Then, several output files are produced. In most cases, their name is based on the name of the input file, with a “zz_” prefix and a suffix which depends on the file type. The “zz_prefix” is a way to gather all these files in a file explorer. The following tables present the different INPUT and OUTPUT files:

**INPUT Files**

<table>
<thead>
<tr>
<th>Name</th>
<th>File function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Main input file</td>
<td>This is the only input the user has to specify to the solver.</td>
</tr>
<tr>
<td>(any name)</td>
<td>Other input files</td>
<td>Any arbitrary number of files, used by means of $INCLUDE instructions, or by sub-modeling techniques.</td>
</tr>
</tbody>
</table>

**OUTPUT Files**
### File function and Description

<table>
<thead>
<tr>
<th>Name</th>
<th>File function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>zz_filename_for.f</code></td>
<td>Fortran code file</td>
<td>These files contain the output of the Fortran-like language translation</td>
</tr>
<tr>
<td><code>zz_filename_com.fi</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>zz_filename_for.o</code></td>
<td>Compiled code</td>
<td>Compiled code</td>
</tr>
<tr>
<td><code>zz_filename_prg_SYS</code></td>
<td>Solution program</td>
<td>(SYS refers to the operating system)</td>
</tr>
<tr>
<td><code>Filename.log</code></td>
<td>Log file</td>
<td>This file relates the main events occurring during the translation task</td>
</tr>
<tr>
<td><code>filename.out</code></td>
<td>Text file</td>
<td>This file contains the standard output text generated during the calculation</td>
</tr>
<tr>
<td><code>filename.csv</code></td>
<td>Spreadsheet run report</td>
<td>Spreadsheet run report</td>
</tr>
<tr>
<td><code>Modelname.temp.h5</code></td>
<td>V4 result file</td>
<td>New result file in the HDF5 format</td>
</tr>
<tr>
<td>(any name)</td>
<td>Other specific output files</td>
<td>Other specific output files, depending on the solver subroutines called during the calculation</td>
</tr>
<tr>
<td>(any name)</td>
<td>User specific output files</td>
<td>User specific output files, which may be created by user subroutines</td>
</tr>
</tbody>
</table>

The results are returned into an ASCII output file. The output data consist of node temperatures, and optionally all the properties required for a complete analysis of heat transfers. The user can access several output subroutines (including spreadsheet formats) and can also define his own data storage. A comma-separated values file is also updated during the execution of the resolution program so the user can check the convergence of the model. There are no results but the convergence criteria only. More complete results are exported to a hdf5 binary file for post-processing use. This file is compatible with SYSTEMA (for graphic post-processing) and with POSTHER (for tabulated post-processing analysis) or BPLOT (for curve creations).

### Software Architecture

The temperature solver includes three main parts:

- **A translator**, which loads the input data and produces the corresponding Fortran code.
- **A computation library**, which provides a set of solution routines and several other services.
- **A Fortran compiler**, which produces the solution program. In general, any compiler can be used (depending on the computer configuration). However, the use of a generic compiler (GNU) is recommended.

The user input file is translated into a Fortran code, which is compiled and linked with the solver library; this produces a solution program which generates the required results. A launcher program automatically performs the chaining of all these tasks.
Temperature Calculation

The solution program is a merging of the Fortran code written by the user and the solver computation library. After some initialization operations, the program makes a call to the user subroutine $INITIAL from which some specific initializations can be performed, and then calls the user subroutine $EXECUTION from which solution routines are called, typically for a steady-state calculation followed by one or more transient calculations. Each solution routine will make some regular calls to the user subroutines $VTEMPERATURE, $VTIME, $VRESULT and $OUTPUTS, at moments which depend on the solution routine (at each time step, at each iteration, ...). For the user, its is an effective way to program specific events: temperature-dependent data (such as conductive couplings or capacitance), time-dependent data (such as activation/deactivation of heaters or dissipation of equipment), through a Fortran-like language which enables any arbitrary complex programming. These calls concern the main model as well as sub-models. Sub-model routines are called before their father, so that the father model’s instructions will take precedence.
User Input Data Structure

Model definition

The user input is text data, contained in one or several files. The language is based on the ESATAN standard language.
Comments can be inserted anywhere in the file. The sharp symbol (#) indicates the beginning of a comment, until the end of line.
A thermal model can be a standalone model or can include sub-models, whose declarations have to be placed before the definition paragraphs of the father model. There is no restriction on the number of submodels and a submodel can be used many times as long as the path are unique.

There are different ways to insert a Submodel. Those are described in section 1.5
Example of submodel hierarchy

The information of one model is divided into several paragraphs. There are classified in two families:

- **Declaration and definition paragraphs** Definition of nodes and couplings, Declaration of variables, arrays and tables. Each paragraph uses a Fortran-like syntax to declare the data. The number of columns is not limited. The original specification for those paragraphs is that all instruction lines end by a semi-coma character. There is no line continuity character. Since v4.5.0, the semi-coma at the end of instructions is not required but may be advised for compatibility with the ESATAN language.

- **Execution paragraphs** Definition of user subroutines and solver special subroutines. They use the Mortran language, a thermal-oriented Fortran-like language, based on Fortran 77. The format is free meaning that there is no limitation on the number of columns or on the column index to start a command. The continuation character is advised to be the ‘&’ character. However for compatibility with the older fixed format, any meaningless character placed on the 5th column will also be interpreted as a continuation character. This behavior is called mixed format.

A thermal model begins with a $MODEL instruction and ends with a $ENDMODEL instruction. The definition paragraphs and execution paragraphs are inserted between them.

All these data are not necessarily stored in the same input file; the use of $INCLUDE instructions enables to randomly organize data in several files.

**Declaration paragraphs**

Any type of declarative paragraph may appear several times in one model and in any order. They can also be mixed with the definition paragraphs.
\$LOCALS

This paragraph contains the declaration of constants available in all the definition paragraphs. The constants can be: integers, reals or strings. They cannot be modified at any time. The type may be defined using the sub-blocks $REAL, $INTEGER and $CHARACTER (like in example 1) or using individual type definitions (like in example 2).

**Example of a $LOCALS paragraph (1/2)**

```
$LOCALS
$REAL
RL111 = 4.4; # conductivity longitudinal 20*0.3
RL112 = 1.5; # conductivity transversal 20*0.3
RL115 = 20.6E-3; # thickness 20*0.3
RL119 = (32.*900.*20.E-3) + (2700.*900.*2.*0.3E-3);
$INTEGER
NB_CELLS = 4;
```

**Example of a $LOCALS paragraph (2/2)**

```
$LOCALS
REAL* RL111 = 4.4;
INTEGER* NB_CELLS = 4;
INTEGER IFLAG = 3;
CHARACTER* CASE1 = 'NOMINAL_HOT';
```

The local constants may be defined equal to expressions involving literal values and local constants of the same type, provided that the latter have already been defined themselves. They may be referenced in the declaration, the definition and the execution blocks. They must be defined before they can be referenced. Local constants must be preferred to variables one if they are not updated during the simulation. It will save memory but also the use of those constants will not generate unnecessary GENMOR code (see paragraph on the GENMOR code).

\$VARIABLES

This paragraph contains the declaration of variables (integers, reals, strings) available in all the execution paragraphs. These variables can be used in Mortran expressions (i.e. definition of node temperature, etc.).

```
This paragraph was previously known as $CONSTANTS but due to its definition its name has changed since version 4.3.0. However the old name $CONSTANTS is still admitted for compatibility with previous THERMISOL models and with ESATAN ones.
```

The variables can be: integers, reals or strings and are declared like in the $LOCALS paragraph. Unlike the variables declared in $LOCALS, they can be modified by the user. If a constant is used to define another in the $VARIABLES block or in any other declarative block, then changing the value of the first during a solution run will automatically cause that of the second to be recalculated accordingly.
The second one will be referenced by a specific code used to make such automated updates, the GENMOR subroutine.
The GENMOR subroutine is generated in the same order than the instructions of the input file. It is called at each end of $VTEMPERATURE call (which is the one the most often called). If an update is explicitly necessary, then the user can make a call to that routine anywhere in its code.

Example of a $VARIABLES paragraph

$VARIABLES
# Characteristics of air at 20° C
REAL*rho_air = 1.2; # kg/m3
REAL*Cp_air = 1.2; # J/kg/K

# Characteristics of MLI
$REAL
cond_MLI=0.019; # W/m2K
rad_MLI=0.014; # m2/m2
MCp_MLI=235.0; # J/K/m2 per face
Rho_MLI=0.26; # kg/m2
Cp_MLI=1090.0; # J/kg/K
EFF_MLI=1.D0;
$INTEGER
NCAS=99;
ISLARG=0;
NEWDEPRESSU=1;
$CHARACTER
CASE='NOMINAL_COLD;
HEATER_PZ='OFF';

$CONTROL

This paragraph contains the definition of variables used by the solution routines during computation. The control variables have specific names. User’s variables cannot be defined here. Their values can be changed during the solution (i.e. in the $INITIAL, $VTEMPERATURE, $VTIME, $VRESULT or $EXECUTION subroutines).

General Controls

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Typical value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABS</td>
<td>R</td>
<td>273.15</td>
<td>Conversion from Kelvin to the user unit (generally degree Celsius)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If TABS = 0, the unit of input and output files is Kelvin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If TABS = 273.15, the unit of input and output files is degree Celsius</td>
</tr>
<tr>
<td>DAMPT</td>
<td>R</td>
<td>1.0</td>
<td>Damping factor for iterative resolution. If this value is equal to 1, it is automatically updated during computation</td>
</tr>
<tr>
<td>STEFAN</td>
<td>R</td>
<td>5.67e-8</td>
<td>Stefan-Boltzmann constant.</td>
</tr>
</tbody>
</table>
### Name  
**Type**  
**Typical value**  
**Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Typical value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSV_FREQ</td>
<td>I</td>
<td>2</td>
<td>The csv control file will be written at this frequency</td>
</tr>
<tr>
<td>H5_FREQ</td>
<td>I</td>
<td>2</td>
<td>Results will be stored every n iteration</td>
</tr>
<tr>
<td>H5_RES0</td>
<td>S</td>
<td>NS,MS,C, A,ALP, EPS, GL, GR, GF</td>
<td>Results stored upon initialization. See the Posther paragraph for more details</td>
</tr>
<tr>
<td>H5_RES1</td>
<td>S</td>
<td>T,QS,QA, QE,QI,QR</td>
<td>Results stored with the H5_FREQ frequency</td>
</tr>
</tbody>
</table>

### Steady-State Controls

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Typical value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELXCA</td>
<td>R</td>
<td>1e-4</td>
<td>Maximum permissible temperature change between two iterations</td>
</tr>
<tr>
<td>INBALA</td>
<td>R</td>
<td>1e-3</td>
<td>Maximum allowable absolute flux exchange between diffusive nodes and boundary nodes</td>
</tr>
<tr>
<td>INBALR</td>
<td>R</td>
<td>1e-6</td>
<td>Maximum allowable relative flux exchange between diffusive nodes and boundary nodes</td>
</tr>
<tr>
<td>INBALT</td>
<td>R</td>
<td>1e-3</td>
<td>Maximum allowable sum of all diffusive nodes flux budget (is equivalent to INBALA but takes into account all unbalanced fluxes between diffusive nodes)</td>
</tr>
<tr>
<td>NLOOP</td>
<td>I</td>
<td>10000</td>
<td>Maximum number of iterations allowed</td>
</tr>
<tr>
<td>RELXCC</td>
<td>R</td>
<td></td>
<td>Maximum temperature change obtained between two iterations</td>
</tr>
<tr>
<td>NRLXCC</td>
<td>I</td>
<td></td>
<td>Node on which the maximum temperature change has been obtained</td>
</tr>
<tr>
<td>ENBALA</td>
<td>R</td>
<td></td>
<td>Absolute flux exchange between diffusive nodes and boundary nodes</td>
</tr>
<tr>
<td>ENBALR</td>
<td>R</td>
<td></td>
<td>Relative flux exchange between diffusive nodes and boundary nodes</td>
</tr>
<tr>
<td>ENBALT</td>
<td>R</td>
<td></td>
<td>Sum of all diffusive nodes flux budgets</td>
</tr>
<tr>
<td>LOOPCT</td>
<td>I</td>
<td></td>
<td>Number of iterations performed</td>
</tr>
</tbody>
</table>
## Transient Controls

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Typical value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMEO</td>
<td>R</td>
<td>0</td>
<td>Simulation start time</td>
</tr>
<tr>
<td>TIMEND</td>
<td>R</td>
<td>&gt;0</td>
<td>Simulation end time</td>
</tr>
<tr>
<td>DTIMEI</td>
<td>R</td>
<td>&gt;0</td>
<td>Time step specified</td>
</tr>
<tr>
<td>OUTINT</td>
<td>R</td>
<td>&gt;0</td>
<td>Time interval to execute a $OUTPUT$ paragraph. DTIMEU can be automatically adjusted to reflect an OUTINT time.</td>
</tr>
<tr>
<td>RELXCA</td>
<td>R</td>
<td>1e-4</td>
<td>Maximum allowed temperature change between two iterations, for a given time step</td>
</tr>
<tr>
<td>INBALT</td>
<td>R</td>
<td>1e-3</td>
<td>Maximum allowable sum of all diffusive nodes flux budget (including capacitive fluxes)</td>
</tr>
<tr>
<td>NLOOP</td>
<td></td>
<td>10000</td>
<td>Maximum allowed number of iterations, for a given time step</td>
</tr>
<tr>
<td>ERRMIN</td>
<td>R</td>
<td>1e-4</td>
<td>Minimum error allowed on temperature solution when using auto-adaptative algorithms</td>
</tr>
<tr>
<td>ERRMAX</td>
<td>R</td>
<td>1e-4</td>
<td>Maximum error (see errmin)</td>
</tr>
<tr>
<td>SRATIO</td>
<td>R</td>
<td>0.9</td>
<td>Necessary ratio of nodes respecting the maximum error in order to create a sub time step</td>
</tr>
<tr>
<td>TIMEM</td>
<td>R</td>
<td></td>
<td>Current time</td>
</tr>
<tr>
<td>TIMEO</td>
<td>R</td>
<td></td>
<td>Initial time of current time-step</td>
</tr>
<tr>
<td>TIMEN</td>
<td>R</td>
<td></td>
<td>End time of current time-step</td>
</tr>
<tr>
<td>DTIMEU</td>
<td>R</td>
<td></td>
<td>Time step used by the solution routine</td>
</tr>
<tr>
<td>RELXCC</td>
<td>R</td>
<td></td>
<td>Maximum temperature change obtained between two iterations, for a given time step</td>
</tr>
<tr>
<td>NRLXCC</td>
<td>I</td>
<td></td>
<td>Node on which the maximum temperature change has been obtained (internal numbering)</td>
</tr>
<tr>
<td>ENBALT</td>
<td>R</td>
<td></td>
<td>Sum of all diffusive nodes flux budgets</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Typical value</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>---------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>LOOPCT</td>
<td>I</td>
<td></td>
<td>Number of iterations performed</td>
</tr>
</tbody>
</table>

**Example of a $CONTROL paragraph**

```plaintext
$CONTROL
STEFAN=5.6686E-08;
RELXCA=0.0001;
NLOOP=5000;
TIMEND=24.*3600.0;
TIMEO=0.0000E+00
DTIMEI=60.D0
OUTINT=1200.D0
```

**$ARRAYS**

This paragraph contains the definition of user arrays. These arrays can be of 1 or 2 dimensions. They can be integers, reals or strings. They are declared with the same convention than for the $LOCALS and $VARIABLES. An array is declared with its dimensions into parenthesis. If 2 dimensions are declared the first dimension is the number of columns and the second is the number of lines. The last dimension may left to '*' in order to adjust automatically the size of the array according to the declared elements (number of lines in case of a 2D array).

For 2D arrays if the number of declared elements is not a multiple of the number of columns an error will be raised and the program will stop. Otherwise, if the second dimension is set to '*', it will be automatically adjusted to the number of lines. If the second dimension is valued and is greater than the number of declared elements, it will also be adjusted to the number of declared lines. In case the specified dimension is smaller than the number of declared elements an error will be raised and the program will stop.

In case an array shall be declared with more elements than specified in the instantiation of the array, a parameter called SIZE may be used to force the size of the array. For 2D array, the specified size shall be a multiple of the number of column otherwise an error is raised. If the declared size is greater than the number of declared elements, the array will keep the original size given and will not be automatically adjusted. On the other hand if the declared size is too small an error is raised.

It exist two instantiation modes for setting the array values:

- **Values separated by commas**
- **A repeated value defined by using the @ character (number of iteration @ value to be affected)**

These two modes can be used together (see the above examples). For the repetition mode, it is possible to use the '*' character meaning the total number of array element. Of course using *@value is meaningless in the case of a variable length array.

**Example of a $ARRAYS paragraph (1/2)**

```plaintext
$ARRAYS
$REAL
QI_ES (2,4) =
0.,28.60,
1708.,28.60,
1708.5,20.00,
99999.,20.00,
REAL*HconvPret (2,*) =
.0,1.5,
```

© Airbus Defence and Space SAS 2023 - All rights reserved
Example of a $ARRAYS paragraph (2/2)

$ARRAYS
$REAL
NN (99999)=99999@0;
REAL*MeanQR(99)=1.0, 2.0, 96@0.0D0, 1.0;
CHARACTER*LineName(99)=@'Initial name';

Since version 4.3.2, $LOCALS references can be used in addition to explicit values. $VARIABLES (or $CONSTANTS) references are however not allowed (no GENMOR code created for $ARRAYS definitions).

Example of an $ARRAYS setting and usage

$ARRAYS
# Definition of panel elements existence (default value is exist)
INTEGER FillPanel(8,6)= *@1;
# Define holes in panel
SET FillPanel(3,2) = 0;
SET FillPanel(5,2) = 0;
$NODES
DO I=1,8
   DO J=1,6
      IF (FillPanel(I,J).EQ.1) THEN
         INUM = I*1000+J
         D:INUM = 'Panel Node';
      ENDIF
   ENDDO
ENDDO

$TABLES

This paragraph contains the definition of user tables. Tables are similar to arrays but are defined in a spreadsheet-like syntax. They contain real values only.

As for $ARRAYS element, since version 4.3.2, $LOCALS references can be used in addition to explicit values. $VARIABLES (or $CONSTANTS) references are however not allowed (no GENMOR code created for $ARRAYS definitions).
2D Tables

Those tables can be considered as a function \( v = f(x,y) \), where \( x \) and \( y \) are independent variables.
The syntax of a 2D table is:
\[
\text{TAB}(X,Y)
\]
\[
X = x_1, ..., x_n, \\
Y = y_1, v_{11}, ..., v_{n1}, \\
... \\
Y = y_m, v_{1m}, ..., v_{nm};
\]
A table can be defined with both classical (example 1) and transposed (example 2) definitions.

Example of a 2D $TABLES paragraph (1/2)

\[
\text{$TABLES} \\
\text{TAB1 (TI, TE)} \\
\text{TE} = 20.0, 50.0, 70.0, \\
\text{TI} = 100.0, 120.0, 150.0, 170.0, \\
\text{TI} = 200.0, 220.0, 250.0, 270.0, \\
\text{TI} = 300.0, 320.0, 350.0, 370.0;
\]

Example of a 2D $TABLES paragraph (transposed table) – (2/2)

\[
\text{$TABLES} \\
\text{TAB2 (TI, TE)} \\
\text{TI} = 100.0, 200.0, 300.0, \\
\text{TE} = 20.0, 120.0, 220.0, 320.0, \\
\text{TE} = 50.0, 150.0, 250.0, 20.0, \\
\text{TE} = 70.0, 320.0, 350.0, 370.0;
\]

In spite of this transposition, the tables TAB1 and TAB2 defined in these 2 examples produce the same table in memory.

3D Tables

Those tables can be interpreted as a function \( v = f(x,y,z) \), where \( x \), \( y \) and \( z \) are independent variables.
Those tables have been newly implemented in version 4.4.0 for some specific cases and for compatibility with ESATAN models which allow those definitions. However if the complete ESATAN definition handles many ways to specify those tables, only the most understandable and the most used ones have been implemented (the others may leading to unclear interpretations by the user).
The following definitions are supported, with all the possible permutations of the \( X \), \( Y \) and \( Z \) specifications:

- Definition 1: Customized Table

This definition allows the specification of a customized table in the sense that it does not define a 3D matrix but values for given triplets \((x,y,z)\) not necessary aligned in all dimensions. The syntax is the following:
\[
\text{TAB}(X,Y,Z) \\
X = x_1, Y = y_1, z_1, v_1, ..., z_k, v_k, \\
... \\
Y = y_m, z_1, v_1, ..., z_k', v_k', \\
... \\
X = x_n, Y = y_1, z_1, v_1, ..., z_k'', v_k'';
\]
... 
\[ Y = y_m', z_1, v_1, ..., z_k'^{-}, v_k'^{-}; \]

- **Definition 2: Condensed Table**

In this definition, 2 dimensions are constant vectors and the values of the other one are not necessary the same for all pairs of the 2 first dimensions.

\[
\text{TAB}(X, Y, Z) \\
Z = z_1, ..., z_k, \\
X = x_1, Y = y_1, v_{11}, ..., v_{1k}, \\
... \\
Y = y_m, v_{1m}, ..., v_{mk}, \\
... \\
X = x_n, Y = y_1, v_{n1}, ..., v_{nk}, \\
... \\
Y = y_m', v_{nm'}, ..., v_{nm'k};
\]

In this example, the values of \( y \) may be different depending on the specific occurrence of \( x \) but the order of \( x, y \) and \( z \) may be also switched.

- **NOT SUPPORTED**: Expended 3D matrix

This definition allows the specification of a full 3D matrix, meaning that values are given for triplets \((x, y, z)\) where \( x, y \) and \( z \) are constant vectors. The syntax is the following:

\[
\text{TAB}(X, Y, Z) \\
X = x_1, ..., x_n \\
Y = y_1, ..., y_m \\
Z = z_1, v_{11}, v_{21}, ..., v_{1k}, ..., v_{nk} \\
... Z = z_k, v_{11k}, v_{21k}, ..., v_{1mk}, ..., v_{nmk}
\]

**$EVENTS**

An event defines a time where the user either wants to call the output block or to update the thermal model. The first category of event is declared by "$OUTPUT" events and the second one by "$TIMESTEP" events.

Any event forces the end of transient analysis time-step to coincide with the event time. In the case of an output event the $OUTPUTS block is called at that specific time.

In the case of a time-step event, the transient analysis knows that there may be a discontinuity in the model or in its environment and it updates the flux at that time so the final flux of the time-step just before the event and the initial flux of the next one are not identical but take into account the discontinuity. The next figure shows the relationships between the computed fluxes at the different times involved by the solution routine. This scheme is respected independently from the use of the time-step event in executive blocks (call to BEFORE, AFTER, AT or BETWEEN test...
Crank-Nicholson scheme using time-step events

Events define a "BEFORE", an "AT" and an "AFTER as shown in the previous figure. Both time step and output events can be periodic. Periodic events allow an event to repeat at regular intervals. A periodic event also has a definition for the "BEFORE", "AT" and "AFTER" states as shown in the next figure. The notion of "BETWEEN" between two periodic events is also represented (concerns only periodic element with the same period)

The declaration of events is made through the \$EVENTS block. It is possible to give literal values, local constants, variables or expressions using literal values, local constants and variables.

For more convenience, the periods can be declared in a preliminary "$PERIOD" block bellow the "$EVENTS" one. Those are considered as real local values.

Example of an SEVENTS block
$EVENTS
$PERIOD # Those are a real local constants
ORBITAL_PERIOD = 5250.4;
P_EQUIP1 = ORBITAL_PERIOD / 20;
$TIMESTEP
My_event = 123.45; # A time-step event
New_event = My_event + 300; # Another time event
SWITCH_EQUIP1 = 32.0 [P_EQUIP1]; # A periodic time-step event
SWITCH_EQUIP2 = 41.0 [P_EQUIP1]; # A periodic time-step event
$OUTPUT
Out_event = 680.0; # A output event
Periodic_out = 500.0 [500.0];

The OUTINT control parameter already defines a periodic output event. It is considered to be the standard periodic output (i.e. it is not necessary to redefine that event in this block)

Definition paragraphs

Those paragraphs describe the network configuration, i.e. the nodes and the couplings. The symbols related to the defined network are called "Mortran symbols" (node numbers, temperatures, fluxes, areas..., couplings). The Mortran syntax is described in the "execution paragraphs" section. It is however possible to use Mortran expressions to affect a nodal property or a coupling.

Any network data (nodal entity or coupling) defined with formulas or Mortran expressions will be automatically added to the GENMOR subroutine.

$NODES

This paragraph contains the declaration of thermal nodes, defined by a node number and initial properties. The syntax is as follows:
Type number = 'name', properties_list;
where:
- type = D (diffusive), B (boundary) or X (inactive)
- number = a positive integer (up to 9 digits)
- 'name' = a string, with single quotes
- properties_list = initial values of node properties (temperature, capacitance, etc.), separated by commas.

Mortran expressions are available and are applied automatically during the SVTEMPERATURE routine via the GENMOR code (see the MORTRAN language section). A boundary node is used to represent a mathematical boundary to the problem, the temperature being prescribed by the user as a fixed value, or one varying with respect to time, or possibly some other quantity. Boundary node temperatures are not changed by the solution. A diffusive node is one whose temperature will be calculated during solution. Inactive nodes are ignored by solution routines, but may be re-activated at any point during the solution by library functions. Similarly, "active" nodes (i.e. diffusion or boundary) may be made inactive during solution. Nodes are defined by the appropriate type symbol (D, B or X), followed by a user-chosen reference integer. This may be either a literal value or a local index. The index can have any name but shall not conflict with any other local
constant declared in the model. Fortran loops FOR can also be used to implicitly declare nodes. Expressions are not allowed in a node declaration since it might conflict with a supernode definition.

The default properties of a THERMISOL node are:

- **T** for the temperature
- **C** for the capacitance
- **QS** for the absorbed solar flux
- **QA** for the absorbed albedo flux
- **QE** for the absorbed IR planet flux
- **QI** for the internal dissipation
- **QR** for the residual flux

The following properties are also default properties but are not directly used by THERMISOL:

- **A** for the radiative area
- **ALP** for the thermo-optical coefficient alpha
- **EPS** for the thermo-optical coefficient epsilon
- **EPSWLB** for wavelength dependent epsilon

In the Mortran code it is also possible to call the following properties:

- **NS** for the node status
- **L** for the node label

The nodes can be declared with many syntax possibilities. Here are some examples that demonstrate them.

**Example of a $NODES paragraph (1/6)**

```
$NODES
D115 = '<6,5,7,1> Strap / Rectang0',
    T = .000E+00,
    C =1.0000E+01,
    A =3.3480E-02,
    ALP =.300, EPS =.030;
D4142 = 'Mur-Z/-X', T = 0.0D0;
X12028 = 'LIQUID MMH SIDE', T = 20.00D0;
B 9999 = 'space node', T =-269.0D0;
```

**Example of a $NODES paragraph using loops (ESATAN-like syntax) (2/6)**

```
$NODES
FOR INODE = 1 TO 6 DO
    M1 = INODE + 770;
    M2 = INODE + 760;
    D M1 = 'MEMBRANE 1', T = 27.5, C = 8.43E-01*517.0, A = 8.43E-01;
    D:M2 = 'MEMBRANE 2', T = 27.5, C = 9.529E-01*517,0, A = 9.529E-01;
END DO
```

**Example of a $NODES paragraph using loops (FORTRAN-like syntax) (3/6)**

```
$NODES
FOR INODE = 1 TO 6 DO
    M1 = INODE + 770;
    M2 = INODE + 760;
    D M1 = 'MEMBRANE 1', T = 27.5, C = 8.43E-01*517.0, A = 8.43E-01;
    D:M2 = 'MEMBRANE 2', T = 27.5, C = 9.529E-01*517,0, A = 9.529E-01;
END DO
```
$NODES
DO INODE = 1, 6
M1 = INODE + 770;
M2 = INODE + 760;
D M1 = 'MEMBRANE 1', T = 27.5, C = 8.43E-01*517.0, A = 8.43E-01;
D:M2 = 'MEMBRANE 2', T = 27.5, C = 9.529E-01*517.0, A = 9.529E-01;
END DO

Since version 4.3.2, a FORTRAN-like syntax has also been added for the loop declarations in order to have more consistency with the FORTRAN/MORTRAN syntax used in the executive blocks.

Example of a $NODES paragraph using pre-defined numbers (4/6)

$LOCALS
$INTEGER
PZ = 101;
MZ = 201;
PY = 301;
MY = 401;
$NODES
D PZ = 'Equipment +Z', T = 0.0, C = 18000.0;
D:MZ = 'Equipment -Z', T = 0.0, C = 4500.0;
D:PY = 'Equipment +Y', T = 0.0, C = 27000.0;
D:MY = 'Equipment -Y', T = 0.0, C = 32000.0;

Example of a $NODES paragraph using variables and Mortran syntax (5/6)

$NODES
D12004 = 'CYL INT.SUP', T = 40.5,
C = 159.4D-9 *ROPTRH * NTRP1(T12004,CPPTRH,1);

Example of a $NODES paragraph using wavelength dependencies (6/6)

$NODES
D115 = 'Wavelength dependent structure', T = 0.0, C = 2500.0,
EPS = EPSWLBEF(), EPSWL=[0.9, 0.8, 0.7, 0.83, 0.85],
ALP = 0.35;

The EPSWL vector has to be valued using square parenthesis because this property is already in a list of properties separated by commas. There shall be exactly NWLBANDS declared values.

The function EPSWLBEF() may be used in order to assign to the standard EPS parameter the effective epsilon of the node according to its wavelength definition (EPSWL) and to its temperature.

Since version 4.5.0, it is important to notice that a node declaration using a symbol has to be mandatory written with two dots or with at least a blank DSYMBOL is no longer accepted because of a possible confusion with a symbol starting with a D character
D:SYMBOL or D SYMBOL are accepted
$ENTITIES

The "Entities" block can be used to declare new nodal entities (integer, real or character – i.e. string of 24 characters) that will be attached to the nodes in addition to the ones already known. The entities are attached to the nodes of the model or sub-model on which it has been declared. This property ensures that a stand alone model on which new nodal entities have been declared would not affect the nodal properties of an existing model that would be its son or its father. If a nodal entity is request in different model, the $ENTITIES block should be present in each one of them.

The $ENTITIES block should be placed before the $NODES block. The "Entities" block replaces the obsolete USRNOD.DAT file.

Example of a $ENTITIES paragraph

$ENTITIES
REAL ON_DISSIP;
REAL OFF_DISSIP;
CHARACTER STATE

The new entities can be directly specified in the $NODES block like in the following example or can be used in any Mortran expression like the original nodal entities.

Example of a $NODES paragraph with user nodal entities

$NODES
D:PZ = 'Equipment +Z', T=0.0, C=18000.0, ON_DISSIP=120, OFF_DISSIP=30,
STATE='ON';
D:MZ = 'Equipment -Z', T=0.0, C=4500.0, ON_DISSIP=70, OFF_DISSIP=0,
STATE='ON';
D:PY = 'Equipment +Y', T=0.0, C=27000.0, ON_DISSIP=150, OFF_DISSIP=50,
STATE='ON';
D:MY = 'Equipment -Y', T=0.0, C=32000.0, ON_DISSIP=120, OFF_DISSIP=10,
STATE='OFF';

When a wavelength discretization is required for modelling the thermal behavior of non-grey bodies, the definition of the wavelength bands shall be placed in the $ENTITIES block. It will then enable the nodal property EPSWLB and the coupling GRWLB in the current model. Any node with an EPSWLB or any GRWLB coupling declared in this model will refer to this wavelength discretization.

The declaration of a wavelength discretization is done as in the following example:

Example of a wavelength discretization declaration

$ENTITIES
WLBANDS (NWLBANDS=6) 0, 0.5, 2.0, 5.0, 10.0, 20.0, 1000.0;

WLBANDS is a real vector defining the wavelength bounds. If the number of wavelength bands is NWLBANDS, the vector WLBANDS shall be declared with exactly NWLBANDS+1 values. The unit of wavelength is micro-meter.
adequate extreme values of this vector are 0 to a sufficiently large value, typically 1000, so the integration of the fractional blackbody emissive power is equal to 1 over the total wavelength interval.

Note: If the model includes sub-models, an $ENTITIES block with a definition of the wavelength bands is also required in each sub-models.

$EDGES

This paragraph contains the declaration of the edges used for the conduction computation. In previous versions edges where considered as standard nodes but their specific meaning has lead to split their definitions. This especially ensures that they are properly handled by the user and the software. The connectivity between the nodes and their edges is also known thanks to its specific declaration so the conductive heat transfer between shapes may be known through the standard FLUXL, FLUXT, FLUXGL etc… even if the shapes are not directly connected but through the edges.

The declaration of edges is made as follow:

E = connectivity list, A=value;

where:

- Connectivity list = node reference |E edge reference relative to the node
- A means the area of the edge (thickness times length) that may be used for further contact resistance declarations.

The declaration of one edge is made by connecting the node edges together. Indeed a free border will reference only one node edge and an edge with contacts will reference several node edges.

A node edge, i.e. an edge referenced by its node belonging, shall appear only once in the $EDGES section otherwise THERMISOL will raise an error. The identification of an edge relatively to a node may be any number not necessarily sequential (but THERMICA creates the edge database using sequential numbers).

You may refer to the Conduction chapter of the THERMICA User’s Manual for more details on the edge creation.

The properties of a THERMISOL edge are:

- T for the temperature. This property is however read-only.
- A for the area (length times thickness), in case a contact resistance is applied to that edge

There is no capacitance or power on edges.

The edges can be declared with many syntax possibilities. Here are some examples that demonstrate them.

Example of a $EDGES paragraph (1/2)

\[
\begin{align*}
$EDGES \\
E & = 100|E1, A=0.0024; \\
E & = 100|E2, 200|E3; \\
E & = 200|E4, 150|E2, A=0.0015 ;
\end{align*}
\]

Example of a $EDGES paragraph using loops (FORTRAN-like syntax) (2/2)

\[
\begin{align*}
$EDGES \\
DO INODE = 1, 6 \\
   M1 = INODE + 770; \\
   M2 = INODE + 760; \\
   E= M1|E2, M2|E3; \\
END DO
\end{align*}
\]
$CONDUCTORS

This paragraph contains the declaration of couplings between nodes. Linear conductive couplings (GL), linear one-way couplings (GF), non-linear radiative couplings (GR) and wavelength dependent non-linear radiative coupling (GRWLB) are available.

Couplings can be declared between nodes of the current model or with nodes of any sub-model by indicating the path of the sub-model from the current one.

It is also possible to define several couplings between the same nodes and at any model level. They can then be accessed via a third index giving the position of the coupling in the model structure. If no coupling index is specified when referencing a coupling, it is assume to be the first one.

As for the nodes, the couplings can be declared using loops and local index. The local index name is not restricted to the ESATAN syntax $KL$ but can be any name of 16 characters. The index can also be combined with mathematical formulae eventually involving other local constants.

The syntax is as follows:

```plaintext
type (n1, n2) = expression;
```

where:

- **type** = GL, GR, GF, GV or GRWLB.
- **n1, n2** = node numbers, with eventual model references and/or mathematical expression using local index and local constants.
- **expression** = a value or an expression. Mortran expressions are available and are applied automatically during the $VTEMPERATURE$ routine through the GENMOR code.

**Example of a $CONDUCTORS$ paragraph (1/4)**

```plaintext
$CONDUCTORS

# S/C Conductive Couplings
GL (16350, 16355) = 0.9;
GL (16360, 4116) = 0.067;
GL (33333, 33332) = 200. * 0.017 * 0.001 / (0.0624 / 2. + 0.12865 / 2.);

# Radiation links
GR (5210, 5213) = 0.035566; # Lower box, electronics to housing
GR (5210, 5211) = 0.004321; # Lower box, base to housing
```

**Example of a $CONDUCTORS$ paragraph using a loop declaration (2/4)**

```plaintext
FOR KL1 = 100 TO 110 DO
   GL(KL1, SUB1:(KL1+900)) = 0.082;
END DO
```

**Example of a $CONDUCTORS$ paragraph using the Mortran syntax (3/4)**

```plaintext
GL (15101, 15151) = A15101*INTRP1 (T15101+T15151)/2.0, SUB1, 1)/6.32D-03;
GL (15102, 15152) = A15102*INTRP1 (T15102+T15152)/2.0, SUB1, 1)/6.32D-03;
GL (15103, 15153) = A15103*INTRP1 (T15103+T15153)/2.0, SUB1, 1)/6.32D-03;
```

⚠️ The expression on the right side of the equal sign must not exceed 1020 characters (Mortran string caption limit)

**Example of a $CONDUCTORS$ paragraph using wavelength dependencies (4/4)**

```plaintext
$CONDUCTORS
GRWLB (16350, 16355) = 0.8, 0.7, 0.72, 0.75, 0.68, 0.65;
```

A wavelength dependent radiative coupling GRWLB is a vector of radiative couplings for each wavelength band. It shall be declared with exactly NWLBANDS values.
Execution paragraphs

Generalities

The user can control the solution using some specific blocks that are executed during the execution. The language used in those blocks is called Mortran which is derived from the Fortran language and has specific syntax related to the thermal model. It is so possible to access all the data of the model by use of their name, i.e. any declared local constant, variable, array or table, any coupling or node property.

The execution blocks are the following:

- EXECUTION
- INITIAL
- OUTPUTS

Complete mode:
- VTEMPERATURE
- VTIME
- VRESULT

or Standard mode:
- VARIABLES1
- VARIABLES2

In addition to those paragraphs, the user can define its own subroutines and functions. Those can then be called from any of the previous blocks.

- SUBROUTINES

All the data of Thermisol are stored in double precision. The pre-processor translate any single precision real value to avoid problems during the solution. It is however recommended that the user checks that no confusion between integer and double precision can be made.

The blocks used to update the data during the solution computation are of 2 kinds:

1. Standard mode ($VARIABLES1 / VARIABLES2)
2. Complete mode ($VTEMPERATURE / VTIME / VRESULT)

To understand the mean of the blocks, the following figures present the execution process of steady-state and transient analysis:
**Standard Executive blocks**

In a steady-state process, $VARIABLES1$ is used to update temperature dependent data. $VARIABLES2$ is used to update data depending on the converged results.
Transient computation process with standard executive blocks

In a transient case, $VARIABLES1$ is used to update time dependent data. For time dependencies, since we are computing the temperature and fluxes at the end time of the current time-step, note that the TIMEM used for the Crank-Nicholson scheme is in fact TIMEN. $VARIABLES2$ is used to update data after the solution have computed for the current time-step. About temperature dependencies, they can be updated in either $VARIABLES1$ or $VARIABLES2$. It is important to notice that those dependencies are not solved during the convergence loop. Temperature dependencies are so discretized with a fixed value during an entire time-step.
Automatic time-step control Transient computation process with standard executive blocks
Complete Executive blocks

For steady-state computation, using standard or complete executive blocks are equivalent. $\text{SVTEMPERATURE}$ is used to update temperature dependant data. $\text{SVRESULT}$ updates data after the results have converged.

For Steady-State analysis:
- $\text{SVTEMPERATURE}$ is equivalent to $\text{VARIABLES1}$
- $\text{SVRESULT}$ is equivalent to $\text{VARIABLES2}$
Transient computation process with complete executive blocks

In this case, compared to the classical blocks, we have a more complete setting of data updates.

For Steady-State analysis:
- $VTIME$ is equivalent to $VARIABLES1$
- $VTEMPERATURE$ has no equivalent
- $VRESULT$ is equivalent to $VARIABLES2$
Advantages of the complete definition blocks

- **Clear definition of blocks purposes**

  In the complete executive block mode, each block has a specific meaning. It is then easy to find the best location for a specific update.

  In the standard executive block mode, except for $VARIABLES2$ which is completely equivalent to $VRESULTS$, $VARIABLES1$ is required to update temperature dependant data in steady-state cases or temperature plus time dependant data in transient cases.

  To give a meaning of the $VARIABLES1$ block, additional statement such as "IF (SOLVER.EQ. 'TR') THEN" are usually required.

- **Real update of temperature dependant properties**
In the complete executive blocks mode, the temperature is updated properly at each convergence loop of a transient run. If, for convergence or other matter, some temperature dependant data require being discontinuously updated (i.e. with a constant value during a time-step), it is then possible to update those in the $VRESULT block.

Use of EVENTS

The events can then be used in executive blocks through the keywords "BEFORE", "AT", "AFTER" and "BETWEEN". Note that if you use an "OUTPUT" event to modify thermal data or any data that influence the thermal environment, the event should also be placed in the $TIMESTEP block (using a different name) in order to correct the flux at that specific time.

The correct syntaxes to use the events are:

**Examples of AFTER and BEFORE**

AFTER My_event DO # The instructions will be executed only [instructions] # if the current time is after the event ENDDO

BEFORE (My_event) DO # The instruction will be executed only [instructions] # if the current time is before the event ENDDO# (note that the parentheses are optional)

BEFORE SWITCH_EQUIP1 DO # The instruction will be executed only [instructions] # if the current time is before any odd ENDDO # occurrence of the periodic event

AFTER (SWITCH_EQUIP1,2) DO # The instruction will be executed only [instructions] # if the current time is after the ENDDO # 2nd occurrence of the periodic event

**Examples of AT**

AT My_event DO # The instructions will be executed only [instructions] # at the event time ENDDO

AT (My_event) DO # Same than previous with parentheses [instructions] ENDDO

AT SWITCH_EQUIP1 DO # The instruction will be executed only [instructions] # at the occurrences of the periodic event ENDDO

AT (SWITCH_EQUIP1,2) DO # The instruction will be executed only [instructions] # at the 2nd occurrence of the ENDDO # periodic event

**Examples of BETWEEN**

BETWEEN My_event & New_event DO # The instructions will be [instructions] # executed between the two ENDDO # events

BETWEEN (My_event & New_event) DO # Same with parentheses [instructions] ENDDO

BETWEEN (SWITCH_EQUIP1 & SWITCH_EQUIP2) DO # The instructions will [instructions] # executed according to
GETTING CLOSE TO AN EVENT FORCES THE TRANSIENT SOLUTION PROGRAM TO ADAPT THE TIME-STEP. THIS IS GENERALLY MADE 2 TIME-STEP BEFORE BEING AT ONE EVENT TIME. THE TIME-STEP RETRIEVE ITS ORIGINAL VALUE ONCE THE EVENT HAS BEEN MET.

$SUBROUTINES

This paragraph contains user functions and subroutines, written in the Mortran language. The pre-processor translates all the Mortran code into Fortran, whereas mere Fortran routines are not changed. Each function or subroutine is available during the solution program.

The syntax is as follows:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>SUBROUTINE MYSUB(arg1,arg2,...)</td>
</tr>
<tr>
<td>SUBROUTINE MYSUB(arg1,arg2,...)LANG=MORTRAN</td>
<td>Mortran subroutine</td>
</tr>
<tr>
<td>SUBROUTINE MYSUB(arg1,arg2,...)SPLIT=N</td>
<td>Mortran subroutine</td>
</tr>
<tr>
<td>SUBROUTINE MYSUB(arg1,arg2,...) LANG=FORTTRAN</td>
<td>Fortran subroutine</td>
</tr>
<tr>
<td>TYPE FUNCTION MYFTN(arg1,arg2,...)</td>
<td>Mortran function</td>
</tr>
<tr>
<td>TYPE FUNCTION MYFTN(arg1,arg2,...) LANG=MORTRAN</td>
<td>Mortran function</td>
</tr>
<tr>
<td>TYPE FUNCTION MYFTN(arg1,arg2,...) LANG=FORTTRAN</td>
<td>Fortran function</td>
</tr>
<tr>
<td>End</td>
<td>RETURN END</td>
</tr>
</tbody>
</table>

where:

- **TYPE** is the type of returned values: INTEGER, CHARACTER or DOUBLE PRECISION.
- **"** represents the blank character; in the context of Fortran, each new line must begin after 6 blank spaces.
- **arg1, arg2, ...** are the input / output parameters of the subroutine or function.
- *The LANG=MORTRAN or LANG=FORTTRAN instruction refers to the type of subroutine / function, and is optional. By default, the Mortran language is used (this means that the LANG=MORTRAN instruction may be absent).*
- *The SPLIT=N can be used with the Mortran subroutine to automatically split the routine every N line.*
Between the declaration and the end, the user can perform programming tasks as in a classical Fortran code, with the 2 following sections:

- Declaration of variables
- Instructions.

In a FUNCTION, the value is returned as follows:

```fortran
MYFTN = <value>
```

where:

- `MYFTN` is the name of the function
- `<value>` is the value to be returned: an integer, a real or a string.

Example of a $SUBROUTINES$ paragraph (1/2)

```fortran
# MODIFY ABSORBED SOLAR FLUX
SUBROUTINE REDQ (QQ,CENT)
    CENTU = CENT/100.
    FACTOR = CENTU
    QQ = QQ*FACTOR
RETURN
END
SUBROUTINE CHNGQR (TM1,TMA,PM1,PD) LANG = MORTRAN
    DOUBLE PRECISION TM1,TMA,PM1,MAXT,MIDT,PD
    MIDT = TMA + 0.1D0
    MAXT = TMA + 1.0D0
    IF (TM1.LT.TMA.OR.TM1.GT.MAXT) THEN
        PM1 = PM1 + PD*(MIDT-TM1)
        IF (PM1.LT.0.0D0) THEN
            PM1 = 0.0D0
        ENDIF
    ENDIF
RETURN
END
```

Example of a $SUBROUTINES$ paragraph (2/2)

```fortran
DOUBLE PRECISION FUNCTION GL_raccord(DIAM1,DIAM2) LANG = FORTRAN
    DOUBLE PRECISION DIAM1, DIAM2
    DOUBLE PRECISION S1, S2
    DOUBLE PRECISION GG1, GG2
    DIAM1 = .001*DIAM1
    DIAM2 = .001*DIAM2
    S1 = PI*(.5*DIAM1)**2
    S2 = PI*(.5*DIAM2)**2
    GG1 = KL_cable*S1/h_raccord
    GG2 = LAM_connect*S2/(0.5*L_connect)
    GL-raccord = 1./(1./GG1 + 1./GG2)
RETURN
END
```
$INITIAL
This paragraph contains the subroutine called by the solution program at the beginning of its execution or for any "INITIAL" case called from the $PARAMETERS block. The user will use this paragraph to initialize some data:

- Definition of cases,
- Initial values of capacitance and fluxes,
- Initialization of temperatures (using classical instructions or loading an external file).

$EXECUTION
This paragraph contains the subroutine called by the solution program at the beginning of its execution. It can be considered as the main subroutine that controls the calculation; the calls to solution routines are generally placed in this paragraph. This block may be called many times if specified in a $PARAMETERS block.

$VTEMPERATURE – Complete Mode
The user will use this paragraph to define temperature-dependent data (capacitance, conductive couplings...). This paragraph is called at each iteration of the temperature convergence loop. During transient analysis, this paragraph is also called before the beginning of computation, just after the $INITIAL paragraph. During steady-state analysis, if using the iterative Newton-Raphson algorithm (called by the syntax SOLVIT), the execution of this block is optimized in order to prevent any divergence due to the fact that the initial temperatures are far from the converged results.

$VTIME – Complete Mode
VTIME is used to update continuous time-dependant phenomena that will be taken into account for the current time-step, like external heat fluxes. This paragraph is called during transient analysis at the appropriate moment with the correct value of TIMEM in order to evaluate the heat balance of each node at the correct time (usually, for a Crank-Nicholson resolution, TIMEM is equal to TIMEN in order to estimate the flux at the end of the current time-step). This paragraph is also called before the beginning of transient computation, just after the $INITIAL paragraph. When using an automatic time-step control, VTIME is recalled if the definition of the current time-step has changed.

To model discreet time-dependant phenomena, it is recommended to use events so the fluxes are updated to take into account the discontinuity at that specific time.

$VRESULT – Complete Mode
This paragraph is called at the end of each time-step of a transient analysis or at the end of a steady-state analysis. It is also called before the beginning of transient computation, just after the $INITIAL paragraph. This paragraph can be used when converged results are required. It can be used to compute user data depending on the results at a given time, which is the end time of the time-step (TIMEN) or at the end of a steady-state analysis. It should be used to post-process the results at a given time (user defined data storage, computation of user data not influencing the thermal analysis). It is however possible to code a change that influence the thermal data (T, GL, GR, GF, QS, QA, QE, QI, QR). This will impact the flux computation of the time $t^{k+1}$ of the next time-step which means that the change in the thermal environment of the model is taken into account during the next time-step (at its middle time exactly).
If such a modification needs to be taken into account at the exact time of the current TIMEN, it is possible to call a function `UPDATE_FLUX` that will force the initial flux of the next time-step to be re-evaluated instead of using the final flux of the current one. The change is so taken into account immediately in the simulation.

**$VARIABLES1 – Standard Mode**

The definition of this block varies depending on the solution kind:
For steady-state analysis, it is used to update temperature dependant data (equivalent to `$TEMPERATURE`). For transient analysis, it is used to update both time and temperature data. Temperature data remain constants during a time-step.
Due to the inconstant use of the block, it is often required to use statements such as
"IF (SOLVER.EQ.'SS') THEN" or "IF (SOLVER.EQ.'TR') THEN"
in order to specify to context of the block.

**$VARIABLES2 – Standard Mode**

This block is equivalent to `$VERSULTS` from the complete Mode

**$OUTPUTS**

This block is automatically called by the solution routine:

- After the solution in steady-state solution routines,
- Regularly with a periodicity defined by the `OUTINT` control constant or at each OUTPUT event during transient solution routine

It can contain any Fortran instruction but is primarily concerned with instructions for the output of information. Only the main-model `$OUTPUTS` block is executed.

**Parametric Cases**

An additional block in THERMISOL allows the user to define cases for parametric analysis, for example determining the sensitivity of predicted temperatures to the material and optical properties used in a model. The solution is performed repeatedly with certain parameters in the model being varied for each run, as prescribed by the user. If the paragraph `$PARAMETERS` is present, the model defined in the output file, the nominal case, is run as normal prior to any parameter cases being executed.
The results from a parametric run are stored in a single text file (for the ASCII outputs). However the hdf5 results are split into several files. In this case, the default name is reserved for the nominal case and the others are combined with the parameter case name.

**INITIAL and FINAL cases**

Following the `$PARAMETERS` marker, each parameter case definition takes the form

```
!INITIAL|!FINAL [PARNAM = name]
```

command

There may be any number of parameter cases.
Two types of parameter case are available, initial and final. An initial parameter case first restore the state of the model to that defined by the data blocks, and then executes the content of the $INITIAL block. Next, the parameter case commands are enacted, after which the model's $EXECUTION block is obeyed. Final parameter cases execute the commands and the $EXECUTION block only; i.e. the state of the model at a start of a final case is its current state when the previous solution is completed.
Parametric commands

The CHANGE/ command assigns a new value to the entity specified. The OFF/ and ON/ commands turn on and off couplings or submodels.

Example of $PARAMETERS use

$PARAMETERS
!INITIAL PARNAM='HOT_CASE1'
CHANGE/ C:SUB1:2059 = 4500
CHANGE/ GL(1000, SUB1:234) = 0.2
!FINAL PARNAM='HOT_CASE2'
CHANGE/ N4500 = X
OFF/ GF(8300, 6500)
ON/ GF(8400, 6500)

The MORTRAN language

Use of MORTRAN syntax

The THERMISOL input language is based on the FORTRAN language but provides to the user a set of direct references to nodal entities, couplings or control variables. Those specific accesses define the MORTRAN language. The reason for providing a MORTRAN language is to create an easy and understandable language interface which does not require the knowledge of the internal memory structure of the solver. Since the user is dealing with its own node numbering, the MORTRAN language is entirely based on the user’s node ids. In THERMISOL, there are four types of possible MORTRAN syntax:

- **Explicit references**
  
  An explicit reference is when the node ids are explicitly given using either integer values or local integer constants (defined in the $LOCALS block). A model path can also be included. Explicit references can be used in declaration or definition blocks to create dependencies between the declared entity and the reference included in its value (see the GENMOR section).

- **Semi-explicit references**

  In definition blocks it is possible to use loops to create new entities (see examples in $NODES or $CONDUCTORS sections). The index of the loops or any new integer value created inside the loop (as long as it does not depend on an external variable) can be used as a reference to either create the entity or to set a dependency with another MORTRAN reference. This type of reference is semi-explicit because it is using a parameter that is resolved by the THERMISOL pre-processor.

- **Implicit references**

  This functionality has been introduced in version 4.3.2. In executive blocks, it is possible to set references to entities which depend on variables (using loops or declared variables). In that case the reference is not resolved at the pre-process level but during the execution of the code. The implicit references are expressed exactly like the implicit ones but contain in the node ids variables or formulae instead of integers. Model path are so allowed. If formulae are used it has to be written between brackets and preceded by ":=" in order to set the operation priorities and to avoid ambiguities.

Whenever an implicit reference is given, the preprocessor cannot check the existence of the nodes and/or couplings. If the reference is not correct, an error will be raised during the execution of the code.
Macro references

This functionality has been introduced in version 4.3.3.

In executive blocks, it is now also possible to set or modify the values of a group of references at once using a ZNODES descriptor (see next section).

Unlike the previous MORTRAN syntax, the macros are a complete statement and cannot be included into another statement (i.e. the macro defines a full line instruction and cannot be a sub-part of more complete instruction). The structure of the macro is as following:

[nodal entity] : 'ZNODES' [operator] [right member]
[coupling entity] ([Node 1 reference], 'ZNODES') [operator] [right member]

The nodal entity can either be any nodal entity except N (for the internal node number) which not modifiable.
The coupling entities can be GL, GLS, GR, GRS, GF or GFS.
The operator may be = to assign the right member value to all entities or *= to multiply the entity’s values by the right member.
The right member can be as any other right members allowed in the executive blocks and as consequence cannot use itself a 'ZNODES' reference.
For couplings, the first node reference can be either explicit or implicit.
The functionality is particularly interesting for coupling modifications, since a development of the macro expressed with a loop is not possible (due to the fact that not all couplings from a node to the others exist).
Here are some examples of correct syntaxes:

For nodal entities

(N, NS, L, T, C, A, ALP, EPS, QS, QA, QE, QR, QI, or user defined entity)

- Using an integer: T\text{100}, T\text{100}, T\text{100}
- Using a symbol: T\text{SYMBOL}
- Using an expression: T:(\text{SYMBOL1}+\text{SYMBOL2} - 300)
- In a submodel: T\text{SUBMOD}:\text{100}
\begin{align*}
  &\text{T\text{SUBMOD}}:\text{SYMBOL} \\
  &\text{T\text{SUBMOD}}:(\text{SYMBOL1}+\text{SYMBOL2} - 300)
\end{align*}
- Using a macro: T\text{#100-300}' = 10.0
  \begin{align*}
    &\text{T\text{SUBMOD}}:'\text{#100-300}' ' = 10.0
  \end{align*}

It is worth mentioning that all possible syntax available for nodal entities, except the Mortran macro, may be also used with wavelength dependent emissivities adding a reference to the wavelength band:

- EPSWL\text{B100} (WLB=2), EPSWL\text{B100} (WLB=2), EPSWL\text{B100} (WLB=2)
- EPSWL\text{B100} (WLB=\text{IWL}+1)
- EPSWL\text{B100} ('\text{#100-200}' (WLB=\text{IWL}) + 0.28) (Mortran macro are not supported)

For couplings

(GL, GLS, GF, GFS, GR, GRS, GV)

- Using integers: GL\text{100}, GL\text{101}
- Using symbols: GL(\text{SYMBOL, 101})GL(\text{SYMBOL1, SYMBOL2})
- Using an expression: GL(\text{SYMBOL1}+\text{SYMBOL2} - 300, 101)
- In a submodel: GL\text{SUBMOD}:\text{100}, \text{SUBMOD}:\text{101}
  \begin{align*}
    &\text{GL\text{SUBMOD}}:\text{100}, \text{SUBMOD}:
    &\text{101} \\
    &\text{GL\text{SUBMOD}}:\text{SYMBOL, SUBMOD}:\text{101}
    &\text{GL\text{SUBMOD}}:(\text{SYMBOL}-300), \text{SUBMOD}:\text{101}
  \end{align*}
- Using a macro: GLS(\text{1000, '100-300'}) = 'X' (or 'OFF')
  \begin{align*}
    &\text{GR\text{SUBMOD}}:\text{SYMBOL, '100-300'} ' = \text{COEFF}
  \end{align*}
It is worth mentioning that the wavelength dependent radiative coupling GRWLB may be used in executive blocks as other GL, GR or GF couplings with the addition of a wavelength band reference. All explicit and implicit spellings of GRWLB are possible. Mortran macro are however not accessible.

- GRWLB([100,101], WLB=3)
- GRWLB(SUBMOD:100, SUBMOD:101, WLB=3), GRWLB:SUBMOB(100,101, WLB=3)
- GRWLBs([100, '#100-300'])= 'X' (Mortran macro are not supported)

The wavelength band reference shall always be placed last.

Note: When using sub-models, the wavelength discretization reference of a GRWLB is the one of the model in which the coupling is declared.

**Node specifications (ZNODES)**

Many output routines have an argument describing the list of nodes concerned by the output. This specification is given as a character string based on the following rules:

- A **ZNODES specification can be a combination of different singular ZNODES separated by ‘;’**
- Any singular ZNODES specification can be additive or subtractive if it is prefixed by the character ‘!’
- A singular ZNODES specification can be:
  - **ALL**: all the nodes belonging to the current model (sub-models included)
  - @ [model path]: all the nodes belonging to the model specified by its relative path from the current model (sub-models included)
  - @[model path]:ONLY: all the nodes belonging to the model specified by its relative path from the current model (sub-model excluded)
  - #[node list]: all the nodes listed (see node list definitions below)
  - ![label]: all the nodes from the current model and its sub-models that have the sub-string ‘label’ in their names
  - ![model path]:[label]: all the nodes of the specified sub-model from the current model and its sub-models that have the sub-string ‘label’ in their names
  - ONLY: [label]: all the nodes from the current model (excluding its sub-models) that have the sub-string ‘label’ in their names
  - ![model path]:ONLY:[label]: all the nodes of the specified sub-model from the current model (excluding its sub-models) that have the sub-string ‘label’ in their names

- A node list is formed by singular nodal specifications separated by commas. A singular nodal specification can be:
  - X: A user node number of current model
  - [model path]:X: A user node number of the sub-model given by the model path from the current model
  - X-Y: A range of numbers of the current model. The node X must be an existing node number of the model. The node number Y can either be an existing node or not
  - [model path]:X-Y: A range of user node numbers of the sub-model given by the model path from the current model. The node X must be an existing node number of the specified sub-model. The node number Y can either be an existing node or not

**Examples of ZNODES specifications**

- 'ALL' # all the nodes of current model
- '#1255' # node 1255
- '#100-199' # all the nodes in the range 100–199
- '#1255,100-199' # node 1255, and all nodes in the range 100–199
- '@SUB1' # all the nodes of sub-model SUB1 and its sub-models
- '@SUB1:ONLY' # all the nodes of sub-model SUB1 only
- 'SUB1:350' # node 350 of sub-model SUB1
- 'SUB1:300–399' # range of nodes 300–399 of sub-model SUB1
- 'Equipment' # all the nodes with 'Equipment' in their names
- 'SUB1:Equipment' # all the nodes of SUB1 and its sub-models with
'Equipment' in their names
'ALL;!#1255'               # all the nodes of current model except node 1255
'SUB1;!#1255,100-199;!Equip' # all the nodes of SUB1 plus nodes 100 to 199 from current model except nodes having 'Equip' in their names

Parsing nodes in loops

For parsing nodes and access to their properties or couplings it is possible to do generate a parsing group and to iterate on this group.

To set a group from a node group specification (znodes):

CALL SETGROUP_LABEL ('ZNODES', CURRENT)

To set a group from a coupling type or from all couplings:

CALL SETGROUP_GL (N100) or CALL SETGROUP_GL (N:SYMBOL)
CALL SETGROUP_GR (N100) or CALL SETGROUP_GR (N:SYMBOL)
CALL SETGROUP_GF (N100) or CALL SETGROUP_GF (N:SYMBOL)
CALL SETGROUP_ALLG (N100) or CALL SETGROUP_ALLG (N:SYMBOL)

To get the number of elements within the group:
N = GETGROUP_NBELEM()

To get the user node number of an element of the group:
UNODE = GETGROUP_NUMBER(I)

To get the internal node number of an element of the group:
INODE = GETGROUP_INTNOD(I)

To get the internal model number of an element of the group:
IMODEL = GETGROUP_INTMOD(I)

To get the node model name of an element of the group:  (see SUBMDN function for type description)
MODELN = GETGROUP_SUBMDN(I, TYPE)

Example of node parsing

CALL SETGROUP_GR(N:3200)
N = GETGROUP_NBELEM()
DO I=1,N
   UNODE = GETGROUP_NUMBER(I)
   GR(3200,UNODE) = GR(3200,UNODE) * A:UNODE * 0.01
ENDDO
Use of sub-models

A model can have one or several sub-model(s), declared before its definition paragraphs. A sub-model can also include sub-models. The ways to declare sub-models are described in the following sections.

All the data defined in a sub-model are accessible by the father: nodes, couplings, constants, arrays, subroutines. For that purpose, the model names are used as prefixes:

- **A:HGA:100** is the area of node 100 of sub-model HGA.
- **HGA:COMPUTEHEAT (Q100,1.0)** is the call to the COMPUTEHEAT function coded in the HGA sub-model.
- **C:SUB1:SUB2:100** is the capacitance of node 100 of the SUB2 sub-model contained in the SUB1 sub-model.
- **HGA:POWER** is the POWER variable defined in the HGA sub-model.

Sub-models need to be linked together and with their father; for that purpose, the user can define couplings between them or super nodes.

A coupling between models is a classical coupling involving 2 nodes from different models.

Do not use spaces and operators in sub-model names.

Example of coupling between models

GL(100, NOZZLE:230) = 1.5;

Using super nodes

Super nodes are a way to merge several nodes, usually coming from different models. When a super node is defined, all the nodes merged are removed and all the couplings previously defined with these nodes are associated with the super node.

If a supernode connects a node of the current model, the keyword CURRENT must be explicit.

Example of super node definition

D1000 = 'Antenna baseplate' = HGA:10 + BASE:3300, T = 0.0;
D2000 = 'Equipement baseplat' = CURRENT:425 + EQUIP:8260;

Inline definition of a sub-model

As a model is delimited by the $MODEL and $ENDMODEL keywords, the insertion of sub-models is a natural feature of the language. This kind of definition is fully explicit.

Example of an inline definition of a sub-model

$MODEL MEX
$MODEL MARSIS

# Here is all the definition of model MARSIS
$ENDMODEL
# Here is all the definition of model MEX
$ENDMODEL
**Integration with $EXTERNAL**

The $EXTERNAL function can be used to define a sub-model contained in a separate file and change its name. This is useful when the sub-model is a generic modeling that must be used several times. The name specified by the $EXTERNAL instruction refers to a file that should be located in the same working directory, with a classical suffix (*.DCK or *.d), if applicable.

**Example of $EXTERNAL use**

```
$MODEL ANIK
$MODEL BATTERY1
$EXTERNAL BATT_LITH
$ENDMODEL
$MODEL BATTERY2
$EXTERNAL BATT_LITH
$ENDMODEL
$ENDMODEL
```

**Inclusion with $ELEMENTS**

A sub-model included with the $ELEMENTS function is a parametric sub-model, which means that several parameters can be set by their father. This can be useful to build generic sub-models that can be used several times with some specific changes. The name specified by the $ELEMENTS instruction refers to a file that should be located in the same working directory, with a classical suffix (*.DCK or *.d), if applicable.

The sub-model definition contains parameters written between the '%' symbols, i.e. %PARAM%, which means that the translator will detect that the value of PARAM is not specified in the sub-model but in his father. Such parametric symbols can be used in any declaration or definition block as well as in any Mortran instruction of the execution blocks.

The father specifies the parameters after the $ELEMENTS instruction, in a $SUBSTITUTIONS paragraph. In addition, some parameters can have default values specified in the sub-model, in a $DEFAULTS paragraph.

**Example of $ELEMENTS use**

```
$MODEL E3000
$MODEL LHP1
$ELEMENTS LHPMODEL
$SUBSTITUTIONS
NBNODES = 170
POWER = 200
$ENDMODEL
...
...
$ENDMODEL
```

In this example, the sub-model LHPMODEL is defined this way:

**Example of $ELEMENTS model**

```
$MODEL LHPMODEL
$DEFAULTS
NBNODES = 16
$NODES
FOR INODE = 1 TO %NBNODES%
D INODE = 'tubing', T = 0, C = 15.5;
END FOR
```
D10000 = 'Incoming power', T = 0, QI = %POWER%;
...
...
$ENDMODEL

Library Functions and Subroutines

Solver library content

The solver library contains a large number of functions and subroutines, which can be accessed by the user in the execution paragraphs. The library can be classified into several families of functions:

- **Solutions routines**
  Calculation of the node temperatures for steady-state or transient problems. These routines are usually called in the $EXECUTION paragraph.
  For more details on these routines, see chapter 4 about theoretical background.

- **Heat transfer routines**
  Calculation of flux exchanges between different families of nodes or between different sub-models.

- **Interpolation routines**
  Various interpolation functions to allow access to user arrays or tables.

- **Nodal network management routines**
  Activation, deactivation or status changing of nodes, couplings or sub-models.

- **Mathematical routines**
  Various basic mathematical functions.

- **Data output routines**
  Various routines for results or data output.

- **Other routines**
  Collection of various other functions.

Data output routines

Sorted ENTITIES definition for output routines

For the output data routines exporting a list of values for specified entities, it is also possible to sort the results in increasing or decreasing order of one of the entity.

To specify that a list shall be sorted, the attribute –DEC or –INC can be added next to the entity that should be used to sort the list. This entity is not necessary displayed unless it is explicitly specified.

If no sorted entity is specified, then it assumed to be sorted by increasing node numbering.

**Example of sorted entities**

'T, QS, QA, QE, T-INC'
'QS, QA, QE, T-DEC'
'GL, GL-DEC'
In the first example, the temperature, solar, Albedo and IR powers will be displayed sorted by increasing values of the temperatures.
In the second example, only the powers will be displayed but will however sorted by decreasing temperatures.
The third example shows that it is also possible to sort couplings when it makes sense (ordering by coupling values the temperatures is for example not possible).

**SUBROUTINE PRNDTB (ZNODES,ENTITIES,MODEL)**

- **Input:**
  - ZNODES: Character string describing the nodes concerned
  - ENTITIES: Character string describing the node entities to be printed
  - MODEL: Name of the model concerned
- **Output:** none

**Description:**
This subroutine prints the entities of all the nodes described in NODES in the model described by MODEL. The node entities are described by the ENTITIES string, and can be: label, T, QS, QA, QE, QI, QR or any nodal entity defined by the $ENTITIES block.
The node number and status are systematically printed.

**Example of a PRNDTB subroutine calls**

CALL PRNDTB('#100', 'T,C', SATEM)
CALL PRNDTB('#100-400', 'QS,QA,ENT', CURRENT)  # where ENT is a nodal entity defined in $ENTITIES
CALL PRNDTB('#100,200,300', 'QI,QR', CURRENT)
CALL PRNDTB('#100,200,1000-3000,5000', 'T,C', SATEM)
CALL PRNDTB(' ', 'T,C,QS,QA,QE,QI', SATEM)

**SUBROUTINE PRHEAD**

- **Input:** none
- **Output:** none

**Description:**
**SUBROUTINE PRHEAD**

This subroutine prints the header of the PRNDTB output subroutine, showing the name of the current solution routine, the current time step, and some key convergence parameters.

**Example of a PRHEAD subroutine call**

CALL PRHEAD

---

**SUBROUTINE PRNDBL (ZNODES,DATA,MODEL)**

- **Input:**
  - NODES: Character string describing the nodes concerned
  - DATA: Character string describing the data to be printed
  - MODEL: Name of the model concerned
- **Output:** none

- **Description:**
  This subroutine prints the data described by the DATA string, related to all the nodes described by the NODES string in the model described by MODEL. The data can be:
  - **Node entities:** T, QS, QA, QE, QI, QR or any real nodal entity defined by the user in the file USRNOD.DAT.
  - **Couplings:** GL, GR or GF.

**Example of a PRNDBL subroutine calls**

CALL PRNDBL(' ','T,QS,QI',CURRENT)
CALL PRNDBL('#1000,3000-3101','QI,QR',CURRENT)
CALL PRNDBL(' ','GL',CURRENT)
CALL PRNDBL('#1000,3000-3101','GR',CURRENT)
CALL PRNDBL(' ','T,QS,QI',SATEM)
CALL PRNDBL('#1000,3000-3101','QI,QR',SATEM:TELESCOPE)

---

**SUBROUTINE PTSINK (ZNODE1,CNAME1,ZNODE2,CNAME2,TYPE)**

- **Input:**
  - ZNODE1: Thermal group
  - CNAME1: Submodel name for thermal group
  - ZNODE2: Environment group
  - CNAME2: Submodel name for environment group
  - TYPE: Integer type of sink temperature required
- **Output:** none
**SUBROUTINE PTSINK (ZNODE1,CNAME1,ZNODE2,CNAME2,TYPE)**

- **Description:**

  This subroutine prints the sink temperature of all the nodes described in NODES, in the model described by MODEL (see function TSINK).

  This function can be used for systems with wavelength dependent thermo-optical properties.

  **Example of a PTSINK subroutine calls**

  ```
  CALL PTSINK('#102-103', CURRENT, '#104-105', CURRENT, 4)
  ```

**SUBROUTINE PTSINKN (ZNODES,MODEL)**

- **Input:**
  - NODES: Character string describing the nodes concerned
  - MODEL: Name of the model concerned

- **Output:** none

- **Description:**

  This subroutine prints the sink temperature of all the nodes described in NODES, in the model described by MODEL (see function TSINK).

  This function can be used for systems with wavelength dependent thermo-optical properties.

  **Example of a PTSINKN subroutine calls**

  ```
  CALL PTSINKN('#100', SATEM)
  CALL PTSINKN('#100-400', CURRENT)
  CALL PTSINKN('#100,200,300', CURRENT)
  CALL PTSINKN('#100,200,1000-3000,5000', SATEM)
  CALL PTSINKN('', SATEM)
  ```

**SUBROUTINE QRATES (ZNODES,MODEL)**

- **Input:**
  - NODES: Character string describing the nodes concerned
  - MODEL: Name of the model concerned
### SUBROUTINE QRATES (ZNODES,MODEL)

- **Output:** none

- **Description:**
  
  *This subroutine prints the heat exchanges of all the nodes described in NODES, in the model described by MODEL.*

**Example of a QRATES subroutine calls**

```plaintext
CALL QRATES('#100', SATEM)
CALL QRATES('#100-400', CURRENT)
CALL QRATES('#100,200,300', CURRENT)
CALL QRATES('#100,200,1000-3000,5000', SATEM)
CALL QRATES(' ', SATEM)
```

### SUBROUTINE PRQBAL (ZNODES,MODEL)

- **Input:**
  - NODES: Character string describing the nodes concerned
  - MODEL: Name of the model concerned
- **Output:** none

- **Description:**
  
  *This subroutine prints the heat balance of all the nodes described in NODES, in the model described by MODEL.*

**Example of a PRQBAL subroutine calls**

```plaintext
CALL PRQBAL('#100', SATEM)
CALL PRQBAL('#100-400', CURRENT)
CALL PRQBAL('#100,200,300', CURRENT)
CALL PRQBAL('#100,200,1000-3000,5000', SATEM)
CALL PRQBAL(' ', SATEM)
```

### SUBROUTINE PRQBOU (MODEL)
**SUBROUTINE PRQBOU (MODEL)**

- **Input:**
  - MODEL: Name of the model concerned
- **Output:** none

- **Description:**
This subroutine prints the heat flows along all conductors in submodel MODEL which cross the submodel boundary. A positive flux means the flux goes from the group of nodes specified by MODEL to the rest of the model.

**Example of a PRQBOU subroutine calls**

```plaintext
CALL PRQBOU(SATEM)
CALL PRQBOU(CURRENT)
```

---

**SUBROUTINE PRQNOD (ZNODES, MODEL)**

- **Input:**
  - NODES: Character string describing the nodes concerned
  - MODEL: Name of the model concerned
- **Output:** none

- **Description:**
This subroutine prints the heat flows along all conductors of the group of nodes described by NODES in submodel MODEL which cross the boundary of the group of nodes. A positive flux means the flux goes from the group of nodes to the rest of the model.

**Example of a PRQNOD subroutine calls**

```plaintext
CALL PRQNOD('#100,200,1000-3000,5000',SATEM)
CALL PRQNOD(' ',CURRENT)
```

---

**SUBROUTINE PRTTMD (ZNODES, MODEL)**

- **Input:**
  - NODES: Character string describing the nodes concerned
  - MODEL: Name of the model concerned
- **Output:** none
**SUBROUTINE PRTTMD (ZNODES,MODEL)**

- **Description:**
  
  This subroutine prints the maximum temperature difference between the nodes described in NODES in the model described by MODEL, and prints the 2 nodes for which the maximum difference is observed.

**Example of a PRTTMD subroutine calls**

```
CALL PRTTMD('#100',SATEM) CALL PRTTMD('#100-400',CURRENT)
CALL PRTTMD('#100,200,300',CURRENT)
CALL PRTTMD('#100,200,1000-3000,5000',SATEM)
CALL PRTTMD(' ',SATEM)
```

**SUBROUTINE DMPTHM (FILE)**

- **Input:**
  - FILE: Character string – basename for output files

- **Output:** none

- **Description:**
  
  This subroutine produces a set of CSV files containing nodal data in vector form (temperature, capacitance, and total heat load, one row per node) and conductance matrices (separately linear, radiative and fluidic). The latter are complete NxN matrices (N being the number of nodes in the model), with a zero entry wherever there is no conductor defined between the row and column nodes. This format is designed to be amenable to loading into numeric computation tools such as MATLAB®. There is also a list of nodes in the same row/column order, giving node type, number, label and model name.

  FILE is used as the first part of each file name, unless it is blank in which case the model name is used.

  The files produced are as follows:
  - FILE.nl.csv – Node list: number, type, label, model
  - FILE.nd.csv – Node data: temperature, capacitance, total heat load (vectors)
  - FILE.gl.csv – Linear conductances (matrix)
  - FILE.gr.csv – Radiative conductances (matrix)
  - FILE.gf.csv – Fluidic conductances (matrix)

**Example of a DMPTHM subroutine calls**

```
CALL DMPTHM('export')
CALL DMPTHM(' ')
```
### SUBROUTINE PRNCSV (ZNODES,ENTITIES,MODEL,ORDER,FILE)

- **Input:**
  - ZNODES: Character string describing the nodes concerned
  - ENTITIES: Character string describing the entities to be output
  - MODEL: Name of the model concerned
  - ORDER: Character string defining the ordering of the values.
  - FILE: Character string for the output file name
- **Output:** none

- **Description:**

  This subroutine prints the entities specified in ENTITIES for the group of nodes ZNODES. The data are sorted according to the ORDER specified. Column header are written, and the data are output as a comma-separated value list, appropriate for import into a post-processing tool such as a spreadsheet.

**Example of a PRNCSV subroutine calls**

```fortran
CALL PRNCSV(' ','T',CURRENT, 'NODE', 'temperatures.csv')
CALL PRNCSV('#100-300', 'GL,GR', SUB1, 'ENTITY', 'conductors.csv')
```

### SUBROUTINE PRTTMD (ZNODES,MODEL)

- **Input:**
  - ZNODES: Character string describing the nodes concerned
  - MODEL: Name of the model concerned
- **Output:** none

- **Description:**

  Prints the maximum temperature difference between nodes of ZNODES within the MODEL specified.

**Example of a PRTTMD subroutine call**

```fortran
CALL PRTTMD(' ',SUBMOD:ONLY)
```

### SUBROUTINE PRTNSM (ZNODES,ENTITIES,MODEL)

- **Input:**
  - ZNODES: Character string describing the nodes concerned
  - ENTITIES: Character string describing the entities
  - MODEL: Name of the model concerned
SUBROUTINE  PRTNSM (ZNODES,ENTITIES,MODEL)

- **Output:** none

- **Description:**
Prints the sums of the entities for all the nodes specified

**Example of a PRTNSM subroutine call**
CALL PRTNSM('SCOPE','A, Damage', CURRENT:ONLY)

Output the sums of the area (A) and the user nodal entity called Damage for all the nodes in the current model only (excluding its submodels) having 'SCOPE' as a substring of their labels.

SUBROUTINE  PRTNAV (ZNODES,ENTITIES,MODEL)

- **Input:**
  - ZNODES: Character string describing the nodes concerned
  - ENTITIES: Character string describing the entities
  - MODEL: Name of the model concerned
- **Output:** none

- **Description:**
Prints the average of the entities for all the nodes specified

**Example of a PRTNAV subroutine call**
CALL PRTNAV('SCOPE','A, Damage', CURRENT:ONLY)

H5 output

Overview

Introduction

Posther is the name of the post-processing management of Thermisol. It includes the generation of the output file in the HDF5 format and analysis tools. The advantage of this new data storage and post-processing tools is that it is based on the powerful HDF5 format which is already a standard. Free software applications can be used to manage this type of file, like HDF5 explorer, etc. The file stored in this format (.h5 file) contains general data (program version, user name, start date and time of the simulation…), convergence data (number of iterative loops, evolution of the convergence criteria…), simulation
data (nodes identification and models identification, number of couplings), model properties (areas, capacitance, couplings…), thermal results (temperature, external fluxes…) and analysis data (minmax analyses, flux budgets…).

What is HDF5

The Hierarchical Data Format (HDF) implements a model for managing and storing data. It is compatible with any hardware platform and lets the user manage its data easily. Moreover, the low-level organization of the file is such that the accesses are much faster than with a standard scan of a file.

The structure of the HDF5 format is mostly divided into 3 kinds of objects:

- **The Groups** are similar to folders. They are used to organize the data into different named categories.
- **The Datasets** are the data themselves. They are identified by a name and contain the values of the entity stored.
- **The Attributes** are single values which qualify a group or a dataset. They can be used to give the name of the axis definition of an array or to give additional data.

Generation of the H5 file

Control variables

- **H5_FREQ** This variable control the frequency of the data storage in the h5 file
  
  The default value is 4. This variable can be modified at any time during the simulation.

  | Negative times are systematically not stored in the H5 files. |

  - **H5_RES0** This variable specifies the data that only need to be stored once, at the initialization of the h5 file. The default value is 'NS,A,C,ALP,EPS,GL,GR,GF'. This value cannot be updated during the simulation.
  - **H5_RES1** This variable specifies the time-dependant data that will be stored during the simulation at the frequency given by H5_FREQ. The default value is 'T,QS,QA,QE,QI,QR'. This value cannot be updated during the simulation.

Note that it is possible to export any real or integer user entity in the h5 file at the initialization or at the frequency given by H5_FREQ by adding their name in the H5_RES0 or H5_RES1 variable. You can also export all the real and integer variables defined in the $VARIABLES block by adding 'VAR' in the H5_RES0 or H5_RES1 variable.

**Example of H5 export setting**

H5_FREQ = 2
H5_RES0  = 'NS,A,C,ALP,EPS,GL,GR,GF,ON_DISSIP,CSSGNode';
H5_RES1  = 'T,QS,QA,QE,QI,QR,CSSGNode,VAR';

In this example, the user entity ON_DISSIP (previously defined in the $ENTITIES block) is stored in h5 file once at the initialization, in addition to the other entities stored by default (NS,A,C,ALP,EPS,GL,GR and GF).

The user entity CSSGNode is stored during the whole simulation at the H5_FREQ frequency (i.e. every 2 time steps), in addition to the other entities stored by default (T,QS,QA,QE,QI and QR).

Finally, all the real and integer variables defined in the $VARIABLES block are stored during the whole simulation at the H5_FREQ frequency.
Storage functions

- **H5_INIT(‘ ’)**
  Create a default result file and store the entity of H5_RES0. A filename can be specified as an argument.

- **H5_DUMP**
  Store the results if the current iteration of the call corresponds to the storage frequency H5_FREQ (the iteration numbers are automatically reset at each start of a solution routine – last time of solution call is also stored independently of the frequency).

- **H5_CLOSE**
  Close the current h5 file

Storage behavior

If the initialization function H5_INIT is not specified in the input dck file, an hdf5 output will be automatically generated with a default name. In batch command the option "-no_auto_h5" deactivate the automatic generation of the h5 output.

The automatic generation of the h5 file is made for the entire simulation.

If the user wants to creates several h5 result files and/or if it is required to specify when to initialize and close one file, writing explicitly the command H5_INIT and H5_CLOSE will deactivate the automatic generation of the h5 file and the solver will switch to a manual control.

If the manual control is activated, the store of the results is not automatic and the use of the function H5_DUMP in the $VRESULT paragraph is required to control the dump.

Heat transfer routines

**DOUBLE PRECISION HEATBAL (D1)**

- **Input:**
  - NODE D1: Node designation

- **Output:**
  - DOUBLE PRECISION HEATBAL: Absolute heat balance of the node.

- **Description:**
  This function computes the heat balance of the node by integrating its entire environment.

**Example of a HEATBAL function call**

BALANCE = HEATBAL(N100)

Older syntax 'D100', 'B100' or 'X100' can be used instead of the normalized one 'N100'

**DOUBLE PRECISION FLUXL (J1,J2,K1,K2)**

- **Input:**
  - INTEGER J1: Node number of the first node of group J
  - INTEGER J2: Node number of the last node of group J
  - INTEGER K1: Node number of the first node of group K

DOUBLE PRECISION  FLUXL (J1,J2,K1,K2)

- **Input:**
  - INTEGER K2: Node number of the last node of group K

- **Output:**
  - DOUBLE PRECISION FLUXL: Conductive flux exchanged between group J and group K

- **Description:**
This function computes the power exchanged conductively between two groups of nodes. Two node numbers describe each group, involving all nodes having a number greater or less than the two specified numbers. The following quantity is returned:

\[
FLUXL = \sum_{J1 \leq j \leq J2} \sum_{K1 \leq k \leq K2} C_{j,k}(T_j - T_k)
\]

**Example of a FLUXL function call**

\[
CONDPOWER = FLUXL(N100, N150, N20000, N21000)
\]

Older syntax 'D100', 'B100' or 'X100' can be used instead of the normalized one 'N100'

DOUBLE PRECISION  FLUXR (J1,J2,K1,K2)

- **Input:**
  - INTEGER J1: Node number of the first node of group J
  - INTEGER J2: Node number of the last node of group J
  - INTEGER K1: Node number of the first node of group K
  - INTEGER K2: Node number of the last node of group K

- **Output:**
  - DOUBLE PRECISION FLUXR: Radiative flux exchanged between group J and group K

- **Description:**
This function computes the power exchanged radiatively between two groups of nodes. Two node numbers describe each group, involving all nodes having a number greater or less than the two specified numbers. The following quantity is returned:

\[
FLUXR = \sum \sigma(GR_{j,k}(\lambda^4(T_j - T_k))
\]

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantity is returned:

\[
FLUXR = \sum_{j=J1}^{J2} \sum_{k=K1}^{K2} \sum_{m=1}^{N_{BANDS}} (GR_{j,k,m} \times FBAND(T, \lambda_1, \lambda_2)) \frac{T_j^4}{T_k^4} - \sum_{m=1}^{N_{BANDS}} (GR_{j,k,m} \times FBAND(T, \lambda_m, \lambda_{m+1})) \frac{T_j^4}{T_k^4}
\]

where \( FBAND(T, \lambda_1, \lambda_2) \) gives the fraction of energy radiated between \( \lambda_1 \) and \( \lambda_2 \) for a blackbody at temperature \( T \).

**Example of a FLUXR function call**

\[
RADPOWER = FLUXR(N100, N150, N20000, N21000)
\]

Older syntax 'D100', 'B100' or 'X100' can be used instead of the normalized one 'N100'
**DOUBLE PRECISION**  \( \text{FLUXF} \ (J_1, J_2, K_1, K_2) \)

- **Input:**
  - INTEGER \( J_1 \): Node number of the first node of group \( J \)
  - INTEGER \( J_2 \): Node number of the last node of group \( J \)
  - INTEGER \( K_1 \): Node number of the first node of group \( K \)
  - INTEGER \( K_2 \): Node number of the last node of group \( K \)
- **Output:**
  - DOUBLE PRECISION FLUXF: Convective flux exchanged between group \( J \) and group \( K \)

**Description:**

This function computes the power exchanged through one-way linear couplings between two groups of nodes. Two node numbers describe each group, involving all nodes having a number greater or less than the two specified numbers. The following quantity is returned:

\[
\text{FLUXF} = \sum_{J_1 \leq j \leq J_2, \ K_1 \leq k \leq K_2} \text{FF}_{jk}(T_j - T_k)
\]

**Example of a FLUXF function call**

\[
\text{ONEWAYPOW} = \text{FLUXF}(N100, N150, N20000, N21000)
\]

Older syntax ‘D100’, ‘B100’ or ‘X100’ can be used instead of the normalized one ‘N100’

**DOUBLE PRECISION**  \( \text{FLUXT} \ (J_1, J_2, K_1, K_2) \)

- **Input:**
  - INTEGER \( J_1 \): Node number of the first node of group \( J \)
  - INTEGER \( J_2 \): Node number of the last node of group \( J \)
  - INTEGER \( K_1 \): Node number of the first node of group \( K \)
  - INTEGER \( K_2 \): Node number of the last node of group \( K \)
- **Output:**
  - DOUBLE PRECISION FLUXT: Total flux exchanged between group \( J \) and group \( K \)

**Description:**

This function computes the power exchanged in all possible ways (conductive, radiative and linear one-way) between two groups of nodes. Two node numbers describe each group, involving all nodes having a number higher or below the two numbers specified. The following quantity is returned:

\[
\text{FLUXT} = \sum_{J_1 \leq j \leq J_2, \ K_1 \leq k \leq K_2} \left( \text{GL}_{jk}(T_j - T_k) + \sigma \text{GR}_{jk}(T_j^{4} - T_k^{4}) + \text{GF}_{jk}(T_j - T_k) \right)
\]

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantity is returned:

\[
\text{FLUXT} = \sum_{J_1 \leq j \leq J_2, \ K_1 \leq k \leq K_2} \left( \sigma \sum_{m=1,\text{NWLBANDS}} \left( \text{GR}_{i,j,m} \times \text{FBAND}(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 \\
- \sigma \sum_{m=1,\text{NWLBANDS}} \left( \text{GR}_{j,k,m} \times \text{FBAND}(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 + \text{GL}_{j,k}(T_k - T_j) + \text{GF}_{j,k}(T_k - T_j) \right)
\]

where \( \text{FBAND}(T, \lambda_1, \lambda_2) \) gives the fraction of energy radiated between \( \lambda_1 \) and \( \lambda_2 \) for a blackbody at temperature \( T \).
DOUBLE PRECISION FLUXT \((J1,J2,K1,K2)\)

**Example of a FLUXT function call**

\[
\text{TOTPOW} = \text{FLUXT}(N100,N150,N20000,N21000)
\]

Older syntax 'D100', 'B100' or 'X100' can be used instead of the normalized one 'N100'.

DOUBLE PRECISION FLUXGL \((ZNODE1,MDL1,ZNODE2,MDL2)\)

**Input:**
- \(ZNODE1\) : string describing the first group of node
- \(MDL1\) : starting model for the \(ZNODE1\) specification
- \(ZNODE2\) : string describing the second group of node
- \(MDL2\) : starting model for the \(ZNODE2\) specification

**Output:**
- DOUBLE PRECISION FLUXGL: Conductive flux exchanged between the two groups of nodes

**Description:**
This function computes the power exchanged conductively between two groups of nodes. The following quantity is returned:

\[
\text{FLUXGL} = \sum \sigma \cdot GR_{j,k} \cdot (T_{j}^{4} - T_{k}^{4}) \quad (j \in G1, \ k \in G2)
\]

**Example of a FLUXGL function call**

\[
\text{CONDPOW} = \text{FLUXGL}('\#100-200', \text{CURRENT}, '\text{RADIATORS}', \text{CURRENT})
\]

DOUBLE PRECISION FLUXGR \((ZNODE1,MDL1,ZNODE2,MDL2)\)

**Input:**
- \(ZNODE1\) : string describing the first group of node
- \(MDL1\) : starting model for the \(ZNODE1\) specification
- \(ZNODE2\) : string describing the second group of node
- \(MDL2\) : starting model for the \(ZNODE2\) specification

**Output:**
- DOUBLE PRECISION FLUXGR: Radiative flux exchanged between the two groups of nodes

**Description:**
This function computes the power exchanged radiatively between two groups of nodes. Two node numbers describe each group, involving all nodes having a number greater or less than the two specified numbers. The following quantity is returned:

\[
\text{FLUXGR} = \sum \sum \sigma \cdot GR_{j,k} \cdot (T_{j}^{4} - T_{k}^{4}) \quad (j \in G1, \ k \in G2)
\]

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantity is returned:

\[
\text{FLUXGR} = \sum \sum \sum \sum (GR_{j,k,m} \times FBAND(T_{j}, \lambda_{m}, \lambda_{m+1})) \cdot T_{j}^{4} - \sum \sum \sum \sum (GR_{j,k,m} \times FBAND(T_{j}, \lambda_{m}, \lambda_{m+1})) \cdot T_{k}^{4}
\]

where \(FBAND(T, \lambda_{1}, \lambda_{2})\) gives the fraction of energy radiated between \(\lambda_{1}\) and \(\lambda_{2}\) for a blackbody at temperature \(T\).
**DOUBLE PRECISION** \( \text{FLUXGR} (\text{ZNODE1,MDL1,ZNODE2,MDL2}) \)

**Example of a FLUXGR function call**

\[
\text{RADPOW} = \text{FLUXGR}('\#100-200', \text{CURRENT}, '\text{RADIATORS}', \text{CURRENT})
\]

**DOUBLE PRECISION** \( \text{FLUXGF} (\text{ZNODE1,MDL1,ZNODE2,MDL2}) \)

**Input:**
- \( \text{ZNODE1} \): string describing the first group of node
- \( \text{MDL1} \): starting model for the \( \text{ZNODE1} \) specification
- \( \text{ZNODE2} \): string describing the second group of node
- \( \text{MDL2} \): starting model for the \( \text{ZNODE2} \) specification

**Output:**
- DOUBLE PRECISION \( \text{FLUXGF} \): Convective flux exchanged between the two groups of nodes

**Description:**
This function computes the power exchanged through one-way linear couplings between two groups of nodes. Two node numbers describe each group, involving all nodes having a number greater or less than the two specified numbers. The following quantity is returned:

\[
\text{FLUXGF} = \sum_{j,k} \text{GP}_{jk} \left( T_k^j - T_j^k \right) \quad \left( J \in \mathcal{G}_1, \ K \in \mathcal{G}_2 \right)
\]

**Example of a FLUXGF function call**

\[
\text{CONVPOW} = \text{FLUXGF}('\#100-200', \text{CURRENT}, '\text{RADIATORS}', \text{CURRENT})
\]

**DOUBLE PRECISION** \( \text{FLUXGT} (\text{ZNODE1,MDL1,ZNODE2,MDL2}) \)

**Input:**
- \( \text{ZNODE1} \): string describing the first group of node
- \( \text{MDL1} \): starting model for the \( \text{ZNODE1} \) specification
- \( \text{ZNODE2} \): string describing the second group of node
- \( \text{MDL2} \): starting model for the \( \text{ZNODE2} \) specification

**Output:**
- DOUBLE PRECISION \( \text{FLUXGT} \): Total flux exchanged between the two groups of nodes

**Description:**
This function computes the power exchanged in all possible ways (conductive, radiative and linear one-way) between two groups of nodes. Two node numbers describe each group, involving all nodes having a number higher or below the two numbers specified. The following quantity is returned:

\[
\text{FLUXGT} = \sum \left( \text{GL}_{j,k} \left( T_k^j - T_j^k \right) + \sigma \text{GR}_{j,k} \left( T_k^j + T_j^k \right) + \text{GF}_{j,k} \left( T_k^j - T_j^k \right) \right) \quad \left( J \in \mathcal{G}_1, K \in \mathcal{G}_2 \right)
\]

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantity is returned:
DOUBLE PRECISION  FLUXGT (ZNODE1,MDL1,ZNODE2,MDL2)

\[
\text{FLUXGT} = \sum_{j=1}^{J} \sum_{k=1}^{K} \left( \sum_{m=1}^{M} \left( G_{r,j,m} \times FBAND(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 \right) - \sigma \sum_{m=1}^{M} \left( G_{r,j,k,m} \times FBAND(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 + G_{r,j,k}(T_k - T_j) + G_{r,k,j}(T_k - T_j)
\]

where \(FBAND(T, \lambda_1, \lambda_2)\) gives the fraction of energy radiated between \(\lambda_1\) and \(\lambda_2\) for a blackbody at temperature \(T\).

Example of a FLUXGT function call

\[
\text{TOTPOW} = \text{FLUXGT('100-200',CURRENT,'RADIATORS',CURRENT)}
\]

DOUBLE PRECISION  FLUXML (M1,M2)

- **Input:**
  - MODEL NAME M1: First model
  - MODEL NAME M2: Second model
- **Output:**
  - DOUBLE PRECISION FLUXML: Conductive flux exchanged between model M1 and model M2

- **Description:**
  This function computes the power exchanged conductively between two models or sub-models. The following quantity is returned:

\[
\text{FLUXML} = \sum_{j \in M1} \sum_{k \in M2} G_{l,j,k}(\lambda_j - T_j)
\]

Example of FLUXML function call

\[
\text{CONDPOW} = \text{FLUXML(BATTERY,MARSIS:PLATE)}
\]

DOUBLE PRECISION  FLUXMR (M1,M2)

- **Input:**
  - MODEL NAME M1: First model
  - MODEL NAME M2: Second model
- **Output:**
  - DOUBLE PRECISION FLUXMR: Radiative flux exchanged between model M1 and model M2

- **Description:**
  This function computes the power exchanged radiatively between two models or sub-models. The following quantity is returned:

\[
\text{FLUXMR} = \sum_{j \in M1} \sum_{k \in M2} G_{r,j,k}(\lambda_j - T_j)^4
\]
**DOUBLE PRECISION FLUXMR (M1,M2)**

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantity is returned:

\[
\text{FLUXMR} = \sigma \sum_{j \neq j_2} \sum_{k \neq k_2} \left( \sum_{m=1}^{\text{LOWBANDS}} \left( g_{j,k,m} \times \text{FBAND}(T_j, \lambda_{m+1}) \right) T_j^4 - \sum_{m=1}^{\text{LOWBANDS}} \left( g_{j,k,m} \times \text{FBAND}(T_j, \lambda_{m+1}) \right) T_j^4 \right)
\]

where \( \text{FBAND}(T, \lambda_1, \lambda_2) \) gives the fraction of energy radiated between \( \lambda_1 \) and \( \lambda_2 \) for a blackbody at temperature \( T \).

**Example of FLUXMR function call**

RADPOW = FLUXMR(BATTERY, MARSIS:PLATE)

---

**DOUBLE PRECISION FLUXMF (M1,M2)**

- **Input:**
  - MODEL NAME M1: First model
  - MODEL NAME M2: Second model
- **Output:**
  - DOUBLE PRECISION FLUXMF: Convective flux exchanged between model M1 and model M2

**Description:**

This function computes the power exchanged through one-way linear couplings between two models or submodels. The following quantity is returned:

\[
\text{FLUXMF} = \sum \mathcal{G}_{j,k} \left( T_j - T_k \right) \quad \{ j \in M1 ; k \in M2 \}
\]

**Example of FLUXMF function call**

CONVPOW = FLUXMF(BATTERY, MARSIS:PLATE)

---

**DOUBLE PRECISION TSINK (ZNODE1,CNAME1,ZNODE2,CNAME2,TYPE,ERR)**

- **Input:**
  - ZNODE1: Thermal group
  - CNAME1: Submodel name for thermal group
  - ZNODE2: Environment group
  - CNAME2: Submodel name for environment group
  - TYPE: Integer type of sink temperature required
  - ERR: error code (integer)
- **Output:**
  - DOUBLE PRECISION TSINK: Sink temperature of thermal group
DOUBLE PRECISION TSINK (ZNODE1,CNAME1,ZNODE2,CNAME2,TYPE,ERR)

Description:
This function returns the sink temperature between a group of node (thermal group) and an environment. TYPE will define which kind of sink temperature calculation is to be used. It has to be one of the following integers:

1. Black body radiation sink temperature

\[ T_{\text{Sink,BR}} = \left( \frac{\sum_j \left( \sum_i \left( \sigma T_j^4 \right) \right) \left( \sum_i \sigma A_i \right)}{\sum_i \sigma A_i} \right) \]

2. Grey body radiation sink temperature

\[ T_{\text{Sink,GR}} = \left( \frac{\sum_j \left( \sum_i GR_{i,j} T_j^4 \right) + QS_i + QA_i + QE_i}{\sum_i \sum_j GR_{i,j}} \right) \]

3. Radiative sink temperature

\[ T_{\text{Sink,R}} = \left( \frac{\sum_j \left( \sum_i GR_{i,j} T_j^4 \right)}{\sum_i \sum_j GR_{i,j}} \right) \]

4. Linear sink temperature

\[ T_{\text{Sink,L}} = \left( \frac{\sum_i \left( \sum_j GL_{i,j} T_j \right)}{\sum_i \sum_j GL_{i,j}} \right) \]

The ERR parameter must be defined as an INTEGER.

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantities are returned:

1. Black body radiation sink temperature

\[ \tau_{\text{Sink,BR}} = \left( \frac{\sum_i \left( \sum_m \left( GR_{i,m} \times FBAND(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 \right) - \sum_m \left( GR_{i,m} \times FBAND(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 + \frac{1}{\sigma} \left( QS_i + QA_i + QE_i \right) + \epsilon_i T_j^4}{\sum \epsilon_i A_i} \right) \]

2. Grey body radiation sink temperature

\[ \tau_{\text{Sink,GR}} = \left( \frac{\sum_i \left( \sum_m \left( GR_{i,m} \times FBAND(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 \right) + \frac{1}{\sigma} \left( QS_i + QA_i + QE_i \right)}{\sum \sum \sum \left( GR_{i,m} \times FBAND(T_j, \lambda_m, \lambda_{m+1}) \right)} \right) \]
DOUBLE PRECISION TSINK (ZNODE1,CNAME1,ZNODE2,CNAME2,TYPE,ERR)

3. Radiative sink temperature

\[ T_{\text{Sink,R}} = \sqrt{\sum_i \left( \sum_m \left( GR_{i,m} \times FBAND(T_i, \lambda_m, \lambda_{m+1}) \right) T_j^4 \right)} \]

\[ \sum_i \sum_m \left( GR_{i,m} \times FBAND(T_i, \lambda_m, \lambda_{m+1}) \right) \]

4. Linear sink temperature

\[ T_{\text{Sink,L}} = \frac{\sum_i \left( \sum_j GL_{i,j} T_j \right)}{\sum_i \sum_j GL_{i,j}} \]

where \( \sum_m \) means \( \sum_{m=1,NWLBANDS} \) and \( FBAND(T, \lambda_1, \lambda_2) \) gives the fraction of energy radiated between \( \lambda_1 \) and \( \lambda_2 \) for a blackbody at temperature \( T \).

Example of a TSINK function call

\[ TS = TSINK('\#10-20', \text{CURRENT}, '\#50-60', \text{CURRENT}, 3, \text{ERR}) \]

DOUBLE PRECISION TSINKN (NODE,ERR)

- **Input:**
  - NODE: Node concerned
  - INTEGER ERR: error code
- **Output:**
  - DOUBLE PRECISION TSINKN: Sink temperature of NODE

This function returns the sink temperature of a node. The sink temperature is used to explain in a different way the thermal environment of a node: it is the temperature of a "virtual" node, which radiatively exchanges the same energy as the total energy exchanged by the node with its complex environment (including internal powers):

\[ \sigma \sum_j GR_{i,j} (TSINK^4 - T_i^4) = \Phi_i \]

where \( \Phi_i \) is the power exchanged between the node and its environment:

\[ \Phi_i = \sum_j (GL_{i,j} (T_j - T_i)) + \sigma GR_{i,j} (T_i^4 - T_j^4) + GF_{i,j} (T_j - T_i) + QB + Qb + QB + Qb + QR \]

Hence, the following quantity returned:

\[ TSINK = \sqrt{\frac{2T_i^4 + \sum_i \Phi_i}{\sum_j GR_{i,j}}} \]

The ERR parameter must be defined as an INTEGER.

This function can be used for systems with wavelength dependent thermo-optical properties. In that case, the following quantity is returned:
DOUBLE PRECISION  TSINK(NODE,ERR)

\[ T_{\text{Sink}} = \sqrt{\frac{\bar{\phi}_i}{\sum_j \sum_m \left( \sigma G R_{i,j,m} \times F B A N D(T_j, \lambda_m, \lambda_{m+1}) \right)}} \]

where

\[ \phi_i = \sum_j \left( \sum_{m=1, NW/LBANDS} \left( \sigma G R_{i,j,m} \times F B A N D(T_j, \lambda_m, \lambda_{m+1}) \right) T_j^4 \right) - \sum_m \left( \sigma G R_{i,j,m} \times F B A N D(T_i, \lambda_m, \lambda_{m+1}) \right) T_i^4 
+ G L_{j,i}(T_j - T_i) + G E_{j,i}(T_j - T_i) \]

\[ + Q S_i + Q A_i + Q E_i + Q I_i + Q R_i \]

and \( F B A N D(T, \lambda_1, \lambda_2) \) gives the fraction of energy radiated between \( \lambda_1 \) and \( \lambda_2 \) for a blackbody at temperature \( T \).

**Example of a TSINK function call**

\[
TS = TSINK(N100, \ ERR)
\]

*Older syntax 'D100', 'B100' or 'X100' can be used instead of the normalized one 'N100'*

**WARNING:**

This function was previously known as TSINK but due to ESATAN incompatible definitions, this function has been renamed TSINKN.

However for the compatibility with previous version, THERMISOL automatically recognizes this function even if written at TSINK by checking the number of arguments.

DOUBLE PRECISION  GETCSG(N1)

- **Input:**
  - NODE N1: Node designation
- **Output:**
  - DOUBLE PRECISION GETCSG: CSG value of the node.

- **Description:**
  This function returns the CSG value of a node.

**Example of a GETCSG function call**

\[
CSG = GETCSG(N100)
\]

*Older syntax 'D100', 'B100' or 'X100' can be used instead of the normalized one 'N100'*
## Interpolation routines

### DOUBLE PRECISION INTRP1 (X, ARR, N)

- **Input:**
  - DOUBLE PRECISION X: Independent variable
  - ARR: User array of size (2, n) (DOUBLE PRECISION values)
  - INTEGER N: Interpolation order
- **Output:**
  - DOUBLE PRECISION INTRP1: Interpolation result.

**Description:**

This function performs a linear interpolation over a user array. The array must be defined in the $ARRAYS definition paragraph (in a $REAL sub-paragraph to specify the DOUBLE PRECISION type), and must have 2 columns (size 2, n). The independent variable X is used to scan the 1\textsuperscript{st} column of the ARR array. The interpolation order N may be 0 (constant result over the interval), 1 (for linear interpolation) or 2 (for quadratic interpolation).

**Example of a INTRP1 function call**

```plaintext
QS100 = INTRP1(120.0D0, SOL100, 1)
```

In this example, the solar flux of node 100 (QS100) is the result of the interpolation in the SOL100 user array for the time 120s. The first column of SOL100 contains some time steps and the second column contains the corresponding values for solar flux.

### DOUBLE PRECISION INTRP2 (X, Y, TAB, N)

- **Input:**
  - DOUBLE PRECISION X, Y: Independent variables
  - TAB: User Table array with 2 independent variables
  - INTEGER N: Interpolation order
- **Output:**
  - DOUBLE PRECISION INTRP2: Interpolation result.

**Description:**

This function performs a linear interpolation over a user table array. The table array must be defined in the $TABLES definition paragraph, with 2 independent variables. The independent variables X and Y are used to scan the 1\textsuperscript{st} and 2\textsuperscript{nd} columns respectively of the TAB table. The interpolation order N may be 0 or 1. The order 0 is however not recommended on tables not defined as matrix. The second order is not supported.

**Example of a INTRP2 function call**

```plaintext
C100 = INTRP2(TIMEN, T100, CAPTAB100, 1)
```

In this example, the capacitance of node 100 (C100) is the result of the interpolation in the CAPTAB100 user table which describes a capacitance depending on temperature (T100) and time (TIMEN).
**DOUBLE PRECISION INTRP3 (X,Y,Z,TAB,N)**

- **Input:**
  - DOUBLE PRECISION X,Y,Z: Independent variables
  - TAB: User Table array with 3 independent variables
  - INTEGER N: interpolation order

- **Output:**
  - DOUBLE PRECISION INTRP3: Interpolation result.

**Description:**

This function performs a linear interpolation over a user table array. The table array must be defined in the $TABLES definition paragraph, with 3 independent variables. The independent variables X, Y and Z are used to scan the 1st, 2nd and 3rd columns respectively of the TAB table. The interpolation order N may be 0 or 1. The order 0 is however not recommended on tables not defined as matrix. The second order is not supported.

**Example of a INTRP3 function call**

V = INTRP3(X, Y, Z, TAB, 1)

---

**DOUBLE PRECISION INTERP (X,ARRX,ARRY,N)**

- **Input:**
  - DOUBLE PRECISION X: Independent variable
  - ARRX: User array of size (1,n) (DOUBLE PRECISION values)
  - ARRY: User array of size (1,n) (DOUBLE PRECISION values)
  - INTEGER N: interpolation order

- **Output:**
  - DOUBLE PRECISION INTERP: Result of the interpolation

**Description:**

This function performs a linear interpolation over an ARRY user array. The ARRX and ARRY arrays must be defined in the $ARRAYS definition paragraph (in a $REAL sub-paragraph to specify the DOUBLE PRECISION type), and must have 1 column (size 1,n) and the same size. The independent variable X is used to scan the ARRX array, while the interpolation is performed into ARRY to return the output value. The interpolation order N may be 0 (constant result over the interval), 1 (for linear interpolation) or 2 (for quadratic interpolation).

**Example of an INTERP function call**

QS100 = INTERP(TIMEN, TIMEARRAY, SOLFLX100ARRAY, 1)

In this example, the solar flux of node 100 (QS100) is the result of the interpolation in the SOLFLX100ARRAY user array for the TIMEN time. The TIMEARRAY contains a list of time values, and SOLFLX100ARRAY contains the values of solar flux for node 100 for the same time values.

This function is useful for big models where the user can store the time steps in only one array and all the fluxes in separate arrays without redundant copies of all the time steps (on the contrary, the INTRP1 function requires that all the times and values be stored in the same array).
**DOUBLE PRECISION INTRPA (X, ARR, N)**

- **Input:**
  - DOUBLE PRECISION X: Independent variable
  - ARR: User array of size (1,n) (DOUBLE PRECISION values)
  - INTEGER N: interpolation order

- **Output:**
  - DOUBLE PRECISION INTRPA: Interpolation result.

**Description:**

This function performs a linear interpolation over a user array. The array must be defined in the $ARRAYS definition paragraph (in a $REAL sub-paragraph to specify the DOUBLE PRECISION type) and must have 1 column (size 1,n). The ARR array must have a special format: xstart, deltax, y1, y2, ..., yend. The first two values (xstart and deltax) describe the way the independent variable X must be used to scan the array ARR: y1 corresponds to xstart, y2 corresponds to (xstart+deltax), y3 corresponds to (xstart+2deltax), ... This allows to obtain a result more rapidly than in the other interpolation routines. The independent variable X is used to scan the array ARR from its 3rd value. Only the 0 and 1 interpolation order N are supported by this function.

**Example of an INTRPA function call**

\[ \text{GL}(100, 200) = \text{INTRPA}((T100+T200)/2.0D0, \text{COUPL}, 1) \]

In this example, the conductive coupling between nodes 100 and 200 is interpolated from the COUPL table containing coupling values for temperatures starting from –20°C with a 5°C step. The independent variable input to the routine is the mean temperature of the two nodes.

**DOUBLE PRECISION INTCY1 (X, ARR, N, PERIOD, SHIFT)**

- **Input:**
  - DOUBLE PRECISION X: Independent variable
  - ARR: User array of size (2,n) (DOUBLE PRECISION values)
  - INTEGER N: interpolation order
  - DOUBLE PRECISION PERIOD, SHIFT: period and shift concerning the independent variable X

- **Output:**
  - DOUBLE PRECISION INTCY1: Interpolation result.

**Description:**

This function performs a linear interpolation over a user array, assuming a periodic problem concerning the independent variable X. The array must be defined in the $ARRAYS definition paragraph (in a $REAL sub-paragraph to specify the DOUBLE PRECISION type), and must have 2 columns (size 2,n). The independent variable X is used to scan the 1st column of the array ARR. This variable is shifted and reduced to a value within one period, i.e. the interpolation is performed considering MOD(X+SHIFT,PERIOD) as an input. The PERIOD and SHIFT parameters must be defined as REAL to be taken into account. The interpolation order N may be 0 (constant result over the interval), 1 (for linear interpolation) or 2 (for quadratic interpolation). It is worth mentioning that, for a steady analysis, the INTCY1 function returns an average value of the second column of the array ARR.

**Example of an INTCY1 function call**

\[ \text{QS100} = \text{INTCY1}(80000.0D0, \text{SOL100}, 1, 14400.0D0, 0D0) \]
DOUBLE PRECISION  INTCY1  (X,ARR,N,PERIOD,SHIFT)

In this example, the solar flux of node 100 (QS100) is the result of the interpolation in the user array SOL100 for the time 80000s. The solar fluxes are known only for a 14400s = 4 hour orbital period. A periodic interpolation will therefore be performed. The first column of SOL100 contains time steps and the second column contains the corresponding values for solar flux.

DOUBLE PRECISION  INTCY2  (X,Y,TAB,N,PERIOD,SHIFT)

• Input:
  • DOUBLE PRECISION X,Y: Independent variables
  • TAB: User Table array with 2 independent variables
  • INTEGER N: interpolation order
  • DOUBLE PRECISION PERIOD, SHIFT: period and shift concerning the independent variable X
• Output:
  • DOUBLE PRECISION INTCY2: Interpolation result.

Description:
This function performs a linear interpolation over a user table array, assuming a periodic problem concerning the independent variable X. The table array must be defined in the $TABLES definition paragraph, with 2 independent variables. The time must be the first variable of the table. The independent variables X and Y are used to scan the 1st and 2nd columns respectively of the TAB table. The variable X is shifted and reduced to a value within one period, i.e. the interpolation is performed considering MOD(X+SHIFT,PERIOD) as an input. The PERIOD and SHIFT parameters must be defined as REAL to be taken into account. The interpolation order N may be 0 or 1. The order 0 is however not recommended on tables not defined as matrix. The second order is not supported.

Example of an INTCY2 function call
C100 = INTCY2(TIMEN, T100, CAPTAB100, 1, PERIO1, 0FFST)
In this example, the capacitance of node 100 (C100) is the result of the interpolation in the CAPTAB100 user table which describes a capacitance depending on temperature (T100) and time (TIMEN). This concerns the modeling of a periodic time-dependent problem with a period PERIO1. The CAPTAB100 table is thus periodic with respect to time.

DOUBLE PRECISION  INTCY3  (X,Y,Z,TAB,N,PERIOD,SHIFT)

• Input:
  • DOUBLE PRECISION X,Y,Z: Independent variables
  • TAB: User Table array with 3 independent variables
  • INTEGER N: interpolation order
  • DOUBLE PRECISION PERIOD, SHIFT: period and shift concerning the independent variable X
• Output:
  • DOUBLE PRECISION INTCY3: Interpolation result.

Description:
This function performs a linear interpolation over a user table array, assuming a periodic problem concerning the independent variable X. The table array must be defined in the $TABLES definition paragraph, with 3 independent variables. The time must be the first variable of the table. The independent variables X, Y and Z are used to scan the
DOUBLE PRECISION  INTCY3  (X,Y,Z,TAB,N,PERIOD,SHIFT)

1\textsuperscript{st}, 2\textsuperscript{nd} and 3\textsuperscript{rd} columns respectively of the TAB table. The variable $X$ is shifted and reduced to a value within one period, i.e. the interpolation is performed considering $\text{MOD}(X+\text{SHIFT},\text{PERIOD})$ as an input. The \text{PERIOD} and \text{SHIFT} parameters must be defined as REAL to be taken into account. The interpolation order $N$ may be 0 or 1. The order 0 is however not recommended on tables not defined as matrix. The second order is not supported.

Example of an INTCY3 function call

$V = \text{INTCY3}(\text{TIMEN}, Y, Z, \text{TAB}, 1, \text{PERIO1}, 0\text{FFST})$

DOUBLE PRECISION  INTCYC  (X,ARRX,ARRY,N,PERIOD,SHIFT)

- **Input:**
  - DOUBLE PRECISION $X$: Independent variable
  - $\text{ARRX}$: User array of size $\{1,n\}$ (DOUBLE PRECISION values)
  - $\text{ARRY}$: User array of size $\{1,n\}$ (DOUBLE PRECISION values)
  - INTEGER $N$: interpolation order
  - DOUBLE PRECISION PERIOD, SHIFT: period and shift concerning the independent variable $X$

- **Output:**
  - DOUBLE PRECISION INTCYC: Interpolation result.

**Description:**

This function performs a linear interpolation over a user array ARRY, assuming a periodic problem concerning the independent variable $X$. The $\text{ARRX}$ and $\text{ARRY}$ arrays must be defined in the \$ARRAYS definition paragraph (in a \$REAL sub-paragraph to specify the DOUBLE PRECISION type), must have 1 column (size $1,n$) and must have the same size. The independent variable $X$ is used to scan the $\text{ARRX}$ array, while the interpolation is performed into $\text{ARRY}$ to return the output value. The variable $X$ is shifted and reduced to a value within one period, i.e. the interpolation is performed considering $\text{MOD}(X+\text{SHIFT},\text{PERIOD})$ as an input. The PERIOD and SHIFT parameters must be defined as REAL to be taken into account. The interpolation order $N$ may be 0 (constant result over the interval), 1 (for linear interpolation) or 2 (for quadratic interpolation). It is worth mentioning that, for a steady analysis, the INTCYC function returns an average value of the array ARRY.

Example of an INTCYC function call

$\text{QS100} = \text{INTCYC}(\text{TIMEN}, \text{TIMEARR}, \text{SOLFLX100ARR}, 1, 14400.0\text{D0}, 0\text{D0})$

In this example, the solar flux of node 100 ($\text{QS100}$) is the result of the interpolation in the SOLFLX100ARR user array for the time $\text{TIMEN}$. The $\text{TIMEARR}$ array contains a list of time values, and the array SOLFLX100ARR contains the values of solar flux for node 100 for the same time values. The problem is periodic, with a 14400s period.

DOUBLE PRECISION  NODFNC  (TYPE, ANAME|USRFNC[,N])

- **Input:**
### NODFNC (TYPE, ANAME|USRFNC[,N])

- **TYPE**: Integer (1: Lagrangian interpolation; 2: Function evaluation)
- **ANAME**: ARRAY name (if TYPE=1)
- **USRFNC**: Double Precision user function
- **N**: Interpolation order (if TYPE=1)

**Output:**
- DOUBLE PRECISION NODFNC: Interpolation result.

**Description:**
NODFNC provides a shorthand way of defining nodal entities which depend on temperature, such as capacitance or emissivity. This function removes the need for the user to supply the nodal temperature reference explicitly. It is not actually a function in its own right, but a reference to it in the $NODES block its translated by the pre-processor into a call to an appropriate underlying function. If TYPE=1 (interpolation mode), ANAME may be either a 2xN real array or a table array with one independent variable, containing (temperature, property) pairs. If TYPE=2 (function mode), USRFNC may be defined in the $SUBROUTINES block or may be contained in an external library. In either case the function must take a double precision input (temperature).

**Example of an NODFNC function call**

```plaintext
$NODES
D23, T=26.65, C=RHOX*NODFNC(1, SPECHT, 1);
D24, T=26.65, C=RHOX*NODFNC(2, MYFUNC);

Those declarations are equivalent to:
D23, T=26.65, C=RHOX*INTRP1(T23, SPECHT, 1);
D24, T=26.65, C=RHOX*MYFUNC(T24);
```

### CNDFNC (TYPE, ANAME|USRFNC[,N])

- **Input:**
  - ANAME: ARRAY name (if TYPE=1,3)
  - USRFNC: Double Precision user function (if TYPE=2)
  - N: Interpolation order (if TYPE=1)

**Output:**
- DOUBLE PRECISION CNDFNC: Interpolation result.

**Description:**
CNDFNC provides a shorthand way of defining couplings which depend on the average temperature of the two nodes. This function removes the need for the user to supply the nodal temperature reference explicitly. It is not actually a function in its own right but is translated by the pre-processor into a call to an appropriate underlying function.

If TYPE=1 or 3 (interpolation mode), ANAME may be either a 2xN real array or a table array with one independent variable, containing (temperature, property) pairs. If TYPE=2 (function mode), USRFNC may be defined in the $SUBROUTINES block or may be contained in an external library. In either case the function must take a double precision input (temperature).

**Example of an CNDFNC function call**

```
$NODES
D23, T=26.65, C=RHOX*CNDFNC(1, SPECHT, 1);
D24, T=26.65, C=RHOX*MYFUNC(T24);
```
$CONDUCTORS

\[
GL(2,3) = CNDFNC(1,CONDX,1) \times CSA/DX;
\]

\[
GL(2,3) = CNDFNC(2,MYFUNC) \times CSA/DX;
\]

\[
GL(2,3) = CNDFNC(3,MYARR) \times CSA/DX;
\]

Those declarations are equivalent to:

\[
GL(2,3) = INTRP1((T2+T3)/2.0,CONDX,1) \times CSA/DX;
\]

\[
GL(2,3) = MYFUNC((T2+T3)/2.0) \times CSA/DX;
\]

\[
GL(2,3) = (INTGL1(T2,T3,MYARR)/(T3-T2)) \times CSA/DX;
\]

### Mathematical routines

#### DOUBLE PRECISION INTGL1(X1,X2,ARR)

- **Input:**
  - DOUBLE PRECISION X1, X2: Limits of integration interval
  - ARR: User array of size (2,n) (DOUBLE PRECISION values)
- **Output:**
  - DOUBLE PRECISION INTGL1: Integration result
- **Description:**

This function performs an integration over a user array. The array must be defined in the $ARRAYS definition paragraph (in a $REAL sub-paragraph to specify the DOUBLE PRECISION type), and must have 2 columns (size 2,n).

The independent variable used for the integration from X1 to X2 corresponds to the 1\textsuperscript{st} column of the ARR array.

**Example of an INTGL1 function call**

TPS = INTGL1(0.0D0, 120.0D0, SOLARR)

In this example, the TPS variable is used to store the result of the integration from 0 to 120 in the SOLARR user array.

#### DOUBLE PRECISION INTEGL(X1,X2,XARR,YARR)

- **Input:**
  - DOUBLE PRECISION X1, X2: Limits of integration interval
  - XARR, YARR: User array of size (1,n) (DOUBLE PRECISION values)
- **Output:**
  - DOUBLE PRECISION INTGL: Integration result
- **Description:**

This function performs an integration over a user array. The arrays must be defined in the $ARRAYS definition paragraph (in a $REAL sub-paragraph to specify the DOUBLE PRECISION type), and must each have 1 column (size 1,n). The independent variable used for the integration from X1 to X2 corresponds to the 1\textsuperscript{st} array XARR while the values of the function are in the 2\textsuperscript{nd} array YARR.

**Example of an INTEGL1 function call**

TPS = INTEGL(0.0D0, 120.0D0, TIMES, SOLARR)

#### SUBROUTINE THRMS(T,THIGH,TLOW,STATUS)

- **Input:**

Ref: UM.000040364.AIRB © Airbus Defence and Space SAS 2023 - All rights reserved 71
DOUBLE PRECISION T: Temperature
DOUBLE PRECISION THIGH: High temperature threshold
DOUBLE PRECISION TLOW: Low temperature threshold

Output:
INTEGER STATUS: Thermostat status

Description:
This subroutine simulates a basic thermostat:
- STATUS is set to 1 if T ≤ TLOW
- STATUS is set to 0 if T ≥ THIGH
- STATUS is unchanged if TLOW < T < THIGH

Example of a THRMST subroutine call
CALL THRMST(T100,25.0D0,0.0D0,STAT)

The THRMST subroutine should be called in the $VRESULT paragraph.

DOUBLE PRECISION THSTAT(INODE,THIGH,TLOW,POWER)

Input:
NODE ID INODE: Node reference to control and apply the thermostat dissipation
DOUBLE PRECISION THIGH: High temperature threshold
DOUBLE PRECISION TLOW: Low temperature threshold
DOUBLE PRECISION POWER: Thermostat dissipation

Output:
DOUBLE PRECISION THSTAT: Current thermostat dissipation

Description:
This subroutine simulates a thermostat like the THRMST function except that it is suitable for steady-state and transient cases.
It may return a value ON or OFF (dissipation equal to zero or to POWER) but also an intermediate value in steady-state cases corresponding to the duty cycle of the heater so to maintain the average temperature of the heater.
This routine can only be used with an identical control and dissipative node (use the version THSTAT2 in other cases).
This routine can be used with only one control node (use the version THSTAT3 for 3 control nodes).

Example of a THSTAT subroutine call
QR100 = THSTAT(N100,25.0D0,0.0D0,50.0D0)

The THSTAT subroutine should be called in the $VTEMPERATURE paragraph.

A basic example below shows the interest of the THSTAT routine compared to the THRMST function. Let’s consider the following basic model:
$NODES
B 100 = ' ', T= 10.000, A= 0.000E+000, C= 0.0000E+000, ALP= 0.0000E+000, EPS= 0.0000E+000;
D 200 = ' ', T= 25.000, A= 0.000E+000, C= 1.0000E+000, ALP= 0.0000E+000, EPS= 0.0000E+000;
$CONDUCTORS
GL(100,200)= 1.D0;

Node 100 is a boundary node with a fixed temperature set to 10°C. Node 200 is a diffusive node with an initial temperature set to 25°C. These two nodes are linked by a conductive coupling. A transient analysis of this model shows that the temperature of node 200 tends to 10°C. It is possible to add a thermostat applied on node 200 to
impose a temperature value between 20°C and 25°C. This can be done using the THR MST routine or the THSTAT routine:

$VARIABLES
INTEGER* THSTS = 0;
REAL* THPOW = 30.0;
REAL* THMIN = 20.0;
REAL* THMAX = 25.0;

$VRESULTS
CALL THR MST(T200, THMAX, THMIN, THSTS)
QR200 = THSTS * THPOW

$VTEMPERATURES
QR200 = THSTAT(N200, THMAX, THMIN, THPOW)

The results obtained after a transient analysis are given in the figures below. We can see that the THSTAT routine allows to better reach the target temperature value with less dissipated power. The THR MST routine overestimates the power needed to reach the target value.
DOUBLE PRECISION THSTAT2(INODE,THIGH,TLOW,POWER,QR)

- **Input:**
  - NODE ID INODE: Node reference to control the thermostat
  - DOUBLE PRECISION THIGH: High temperature threshold
  - DOUBLE PRECISION TLOW: Low temperature threshold
  - DOUBLE PRECISION POWER: Thermostat dissipation
  - DOUBLE PRECISION QR: Dissipation currently applied

- **Output:**
  - DOUBLE PRECISION THSTAT2: Current thermostat dissipation

- **Description:**

  This subroutine simulates a thermostat like the THRMS function except that it is suitable for steady-state and transient cases.

  It may return a value ON or OFF (dissipation equal to zero or to POWER) but also an intermediate value in steady-state cases corresponding to the duty cycle of the heater so to maintain the average temperature of the heater.

  This routine can be used with only one control node (use the version THSTAT3 for 3 control nodes).

**Examples of a THSTAT2 subroutine call**

HPOW = THSTAT2(N100,25.0D0,0.0D0,50.0D0, QR110)

QR110 = THSTAT2(N100,25.0D0,0.0D0,50.0D0, QR110)

Please note that the two example are different. The first example only give the information of what should be the dissipation to applied on node 110 (only post-processing) whereas the dissipation is effectively applied on node 110 in the second example.

The THSTAT2 subroutine should be called in the $VTEMPERATURE paragraph.
DOUBLE PRECISION THSTAT3(INODE1, INODE2, INODE3, THIGH, TLOW, POWER, QR)

- **Input:**
  - NODE ID INODE1, INODE2, INODE3: Node reference to control the thermostat
  - DOUBLE PRECISION THIGH: High temperature threshold
  - DOUBLE PRECISION TLOW: Low temperature threshold
  - DOUBLE PRECISION POWER: Thermostat dissipation
  - DOUBLE PRECISION QR: Dissipation currently applied

- **Output:**
  - DOUBLE PRECISION THSTAT3: Current thermostat dissipation

**Description:**
This subroutine identifies the node (INODE1, INODE2 or INODE3) corresponding to the median temperature value and calls the THSTAT2 subroutine with the median node as input.

**Example of a THSTAT3 subroutine call**

HPOW = THSTAT3(N100,N101,N102,25.0D0,0.0D0,50.0D0, QR110)

The THSTAT3 subroutine should be called in the $VTEMPERATURE paragraph.

DOUBLE PRECISION STEADYQR(INODETM, INODETH, TOBJ, TLOW, POWER, RELAX)

- **Input:**
  - NODE ID INODETM: Node reference to control and apply the dissipation
  - NODE ID INODETH: Node reference to thermostat
  - DOUBLE PRECISION TOBJ: Objective temperature
  - DOUBLE PRECISION TLOW: Low temperature threshold
  - DOUBLE PRECISION POWER: Thermostat dissipation
  - DOUBLE PRECISION RELAX: Relaxation criteria

- **Output:**
  - DOUBLE PRECISION STEADYQR: Current thermostat dissipation

**Description:**
This subroutine simulates a thermostat like the THRMST function except that it is suitable for steady-state cases.

Note: POWER must be a high value to achieve the objective temperature.

It will return an intermediate value corresponding to the duty cycle of the heater so to maintain the average temperature of the heater.

**Example of a STEADYQR subroutine call**

QR110 = STEADYQR(N100, N110, 12.5D0, 0.0D0, 1000.0D0, 0.0001D0)

The STEADYQR subroutine should be called in the $VTEMPERATURE paragraph.

INTEGER ADIM(ARRAY)

- **Input:**
  - ARRAY: User array name

- **Output:**
  - INTEGER ADIM: User array dimension

**Description:**
This function returns the dimension of a user array: 1 or 2 dimensions.

**Example of an ADIM function call**

DIMENSION = ADIM(MYARRAY)

INTEGER ADIMVL(ARRAY,N)
Input:
- ARRAY: User array name
- INTEGER N: Dimension for which the size is requested

Output:
- INTEGER ADIMVL: Size of the Nth dimension of the array

Description:
This function returns the size of the Nth dimension for a user array.

Example of an ADIMVL function call
DIMENSION = ADIM(MYARRAY,2)

INTEGER ASIZE(ARRAY)

Input:
- ARRAY: User array name

Output:
- INTEGER ASIER: User array size

Description:
This function returns the size of the user array: total number of elements.

Example of an ASIZE function call
SIZE = ASIZE(MYARRAY)

SUBROUTINE ACONST(ARRAY,VALUE)

Input:
- ARRAY: User array name (integer or double precision)
- VALUE: Value to be set (integer or double precision)

Output:
- ARRAY: Updated user array

Description:
This subroutine sets all the elements of ARRAY to VALUE.

Example of an ACONST subroutine call
CALL ACONST(MYARRAY, 23.2D0)

SUBROUTINE AELCPY(ARRAY1,ARRAY2)

Input:
- ARRAY1: User array name (integer or double precision)

Output:
- ARRAY2: Name of the resulting user array (integer or double precision)

Description:
This subroutine copies all the elements of ARRAY1 to ARRAY2.

Example of an AELCPY subroutine call
CALL AELCPY(MYARRAY1, MYARRAY2)

SUBROUTINE AELADD(ARRAY1,ARRAY2,ARRAY3)
• **Input:**
  • ARRAY1: Name of the first user array (integer or double precision)
  • ARRAY2: Name of the second user array (integer or double precision)
• **Output:**
  • ARRAY3: Name of the user array storing the result (integer or double precision)
• **Description:**

This subroutine performs the addition \( ARRAY3 = ARRAY1 + ARRAY2 \), element by element.  
**Example of an AELADD subroutine call**  
CALL AELADD(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELSUB(ARRAY1,ARRAY2,ARRAY3)**

• **Input:**
  • ARRAY1: Name of the first user array (integer or double precision)
  • ARRAY2: Name of the second user array (integer or double precision)
• **Output:**
  • ARRAY3: Name of the user array storing the result (integer or double precision)
• **Description:**

This subroutine performs the difference \( ARRAY3 = ARRAY1 - ARRAY2 \), element by element.  
**Example of an AELSUB subroutine call**  
CALL AELSUB(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELMLT(ARRAY1,ARRAY2,ARRAY3)**

• **Input:**
  • ARRAY1: Name of the first user array (integer or double precision)
  • ARRAY2: Name of the second user array (integer or double precision)
• **Output:**
  • ARRAY3: Name of the user array storing the result (integer or double precision)
• **Description:**

This subroutine performs the multiplication \( ARRAY3 = ARRAY1 \times ARRAY2 \), element by element. This is not a classical matrix product.  
**Example of an AELMLT subroutine call**  
CALL AELMLT(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELDIV(ARRAY1,ARRAY2,ARRAY3)**

• **Input:**
  • ARRAY1: Name of the first user array (integer or double precision)
  • ARRAY2: Name of the second user array (integer or double precision)
• **Output:**
  • ARRAY3: Name of the user array storing the result (integer or double precision)
• **Description:**

This subroutine performs the division \( ARRAY3 = ARRAY1 / ARRAY2 \), element by element. This has nothing to do with the classical matrix inversion.  
**Example of an AELDIV subroutine call**  
CALL AELDIV(MYARRAY1, MYARRAY2, MYARRAY3)
**SUBROUTINE AELINV(ARRAY1,ARRAY2)**

- **Input:**
  - ARRAY1: Name of the user array (double precision)
- **Output:**
  - ARRAY2: Name of the user array storing the result (double precision)

**Description:**

This subroutine performs the inversion $\text{ARRAY2} = 1/\text{ARRAY1}$, element by element. This has nothing to do with the classical matrix inversion.

Example of an AELINV subroutine call
CALL AELINV(MYARRAY1, MYARRAY2)

**SUBROUTINE AELGT(ARRAY1,ARRAY2,ARRAY3)**

- **Input:**
  - ARRAY1: Name of the first user array (integer or double precision)
  - ARRAY2: Name of the second user array (integer or double precision)
- **Output:**
  - ARRAY3: Name of the user array storing the result (integer or double precision)

**Description:**

This subroutine performs the logical comparison $\text{ARRAY1} > \text{ARRAY2}$, element by element. If the logical comparison returns TRUE, the value of the element of ARRAY3 is set to the corresponding element of ARRAY1; if FALSE is returned, the element of ARRAY3 is set to 0.

Example of an AELGT subroutine call
CALL AELGT(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELGE(ARRAY1,ARRAY2,ARRAY3)**

- **Input:**
  - ARRAY1: Name of the first user array (integer or double precision)
  - ARRAY2: Name of the second user array (integer or double precision)
- **Output:**
  - ARRAY3: Name of the user array storing the result (integer or double precision)

**Description:**

This subroutine performs the logical comparison $\text{ARRAY1} \geq \text{ARRAY2}$, element by element. If the logical comparison returns TRUE, the value of the element of ARRAY3 is set to the corresponding element of ARRAY1; if FALSE is returned, the element of ARRAY3 is set to 0.

Example of an AELGE subroutine call
CALL AELGE(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELLT(ARRAY1,ARRAY2,ARRAY3)**

- **Input:**
  - ARRAY1: Name of the first user array (integer or double precision)
  - ARRAY2: Name of the second user array (integer or double precision)
- **Output:**
  - ARRAY3: Name of the user array storing the result (integer or double precision)

**Description:**
This subroutine performs the logical comparison \( \text{ARRAY1} < \text{ARRAY2} \), element by element. If the logical comparison returns TRUE, the value of the element of \( \text{ARRAY3} \) is set to the corresponding element of \( \text{ARRAY1} \); if FALSE is returned, the element of \( \text{ARRAY3} \) is set to 0.

**Example of an AELLT subroutine call**

CALL AELLT(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELLE(ARRAY1,ARRAY2,ARRAY3)**

- **Input:**
  - ARRAY1: Name of the first user array (integer or double precision)
  - ARRAY2: Name of the second user array (integer or double precision)
- **Output:**
  - ARRAY3: Name of the user array storing the result (integer or double precision)
- **Description:**

This subroutine performs the logical comparison \( \text{ARRAY1} \leq \text{ARRAY2} \), element by element. If the logical comparison returns TRUE, the value of the element of \( \text{ARRAY3} \) is set to the corresponding element of \( \text{ARRAY1} \); if FALSE is returned, the element of \( \text{ARRAY3} \) is set to 0.

**Example of an AELLE subroutine call**

CALL AELLE(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELEQ(ARRAY1,ARRAY2,ARRAY3)**

- **Input:**
  - ARRAY1: Name of the first user array (integer or double precision)
  - ARRAY2: Name of the second user array (integer or double precision)
- **Output:**
  - ARRAY3: Name of the user array storing the result (integer or double precision)
- **Description:**

This subroutine performs the logical test \( \text{ARRAY1} = \text{ARRAY2} \), element by element. If the logical test returns TRUE, the value of the element of \( \text{ARRAY3} \) is set to the corresponding element of \( \text{ARRAY1} \); if FALSE is returned, the element of \( \text{ARRAY3} \) is set to 0.

**Example of an AELEQ subroutine call**

CALL AELEQ(MYARRAY1, MYARRAY2, MYARRAY3)

**SUBROUTINE AELNE(ARRAY1,ARRAY2,ARRAY3)**

- **Input:**
  - ARRAY1: Name of the first user array (integer or double precision)
  - ARRAY2: Name of the second user array (integer or double precision)
- **Output:**
  - ARRAY3: Name of the user array storing the result (integer or double precision)
- **Description:**

This subroutine performs the logical test \( \text{ARRAY1} \neq \text{ARRAY2} \), element by element. If the logical test returns TRUE, the value of the element of \( \text{ARRAY3} \) is set to the corresponding element of \( \text{ARRAY1} \); if FALSE is returned, the element of \( \text{ARRAY3} \) is set to 0.

**Example of an AELNE subroutine call**

CALL AELNE(MYARRAY1, MYARRAY2, MYARRAY3)
Nodal network management routines

**SUBROUTINE STATST(ENTITY,STATUS)**

- **Input:**
  - ENTITY: Node or conductor
  - STATUS: Status to apply to the node or conductor

- **Description:**

  This subroutine is used to change the status of a node or a conductor. The possible statuses for a node are: 'B' (boundary), 'D' (diffusive) or 'X' (inactive). The possible statuses for a conductor are: 'ON' (active) or 'OFF' (inactive).

  **Example of a STATST subroutine call**

  CALL STATST('N300','X')
  CALL STATST(GL(200,1000),'OFF')

  Since version 4.3.1 the STATST routine can be replaced by a direct Fortran syntax:

  NS 300 = 'X' GLS(200,1000) = 'X'
  For couplings, the values are 'A' for active and 'X' for inactive.

**CHARACTER*3 STATRP(ENTITY)**

- **Input:**
  - ENTITY: Node, conductor or model

- **Output:**
  - CHARACTER*3 STATRP: Entity status

- **Description:**

  This function returns the status of a node, a conductor or a model. The possible statuses for a node are: 'B' (boundary), 'D' (diffusive) or 'X' (inactive). The possible statuses for a conductor are: 'ON' (active) or 'OFF' (inactive). The possible statuses for a model are: 'ON' (active) or 'OFF' (inactive).

  **Example of a STATRP function call**

  ST = STATRP('D300')
  WRITE(*,*) 'Coupling status = ', STATRP(GL(200,1000))

  Since version 4.3.1 the STATRP routine can be replaced by a direct Fortran syntax:

  NS 300 GLS(200,1000)
  For couplings, the values are 'A' for active and 'X' for inactive (and not 'ON' or 'OFF')

**SUBROUTINE MDLON(MODEL)**

- **Input:**
  - MODEL: Model to be activated

- **Output:**
  - none

- **Description:**

  This subroutine is used to activate a model, by changing the status of the model to 'ON'. The status of the nodes in this model is not changed: the boundary/diffusive/inactive nodes remain the same.

  **Example of a MDLON subroutine call**

  CALL MDLON(BATTERY)

**SUBROUTINE MDLOFF(MODEL)**

- **Input:**
• **MODEL:** Model to be deactivated  
• **Output:** none  
• **Description:**

This subroutine is used to deactivate a model, by changing the status of the model to 'OFF'. The status of the nodes in this model is not changed: the boundary/diffusive/inactive nodes remain the same.

**Example of a MDLOFF subroutine call**

CALL MDLOFF(LAUNCHER)

---

**SUBROUTINE SAVET(MODEL)**

• **Input:**
  • MODEL: Model concerned  
• **Output:** none  
• **Description:**

This subroutine is used to save the temperature of all the nodes in MODEL, in a binary file. This file is named by the solver as: MODEL.UNF, and will be loaded later by the FETCHT subroutine, possibly during another execution of the solver.

**Example of a SAVET subroutine call**

CALL SAVET(SATEM)  
CALL SAVET(SATEM:TELESCOPE)  
CALL SAVET(CURRENT)

---

**SUBROUTINE FETCHT(MODEL)**

• **Input:**
  • MODEL: Model concerned  
• **Output:** none  
• **Description:**

This subroutine is used to load the temperature of all the nodes in MODEL from a binary file. This file is saved by the solver as: MODEL.UNF, and has been already created, possibly during a previous execution of the solver.

**Example of a FETCHT subroutine call**

CALL FETCHT(SATEM)  
CALL FETCHT(SATEM:TELESCOPE)  
CALL FETCHT(CURRENT)

---

**SUBROUTINE TINIT5(H5NAME, TIME)**

• **Input:**
  • H5NAME: Character string to specify the name of the H5 file  
  • DOUBLE PRECISION TIME: time at which values will be retrieved  
• **Output:** none  
• **Description:**

This subroutine is used to initialize the temperatures from an H5 file generated by thermisol during a previous calculation. TIME=-1 is used to get the temperature values at the last point of the h5 file. This function should be called in the $INITIAL paragraph.

**Example of a TINIT5 subroutine call**

CALL TINIT5('Process.temp.h5', -1.0)  
CALL TINIT5('NewModel.temp.h5', 1286.0)
INTEGER INTNOD(MODEL,NODE)

- **Input:**
  - MODEL: Model concerned
  - INTEGER NODE: User node number
- **Output:**
  - INTEGER INTNOD: Internal node number
- **Description:**

This function returns the internal node number associated with the node which has a user node number NUMBER in MODEL.

**Example of an INTNOD function call**

\[
N = \text{INTNOD}(\text{CURRENT}, 102) \\
N = \text{INTNOD}(\text{SATEM:TELESCOPE}, 1000)
\]

This routine becomes now obsolete in MORTRAN codes with the syntax

"N: [MODEL_PATH:] NODE"

Which is strictly equivalent to "INTNOD(MODEL_PATH,NODE)"

However, with the extended MORTRAN language proposed by THERMISOL, the knowledge of the internal numbering should never be required.

INTEGER NODNUM(N)

- **Input:**
  - INTEGER N: Internal node number
- **Output:**
  - INTEGER NODNUM: User node number
- **Description:**

This function returns the user node number associated with the node which has an internal number N.

**Example of a NODNUM function call**

\[
TN = \text{NODNUM}(3)
\]

CHARACTER*256 SUBMDN(N,NAMETYPE)

- **Input:**
  - INTEGER N: Internal node number
  - NAMETYPE: Character string to specify the name format
- **Output:**
  - CHARACTER*256 SUBMDN: Model name
- **Description:**

This function returns the name of the model or submodel to which the node N belongs (N is an internal node number). The character string NAMETYPE can have the following values:

- 'ALL': The model name will be returned as a full name (e.g: 'SATEM:PLATFORM:TELESCOPE')
- 'ALL-NOMAIN': The model name will be returned as a full name but without the main model name (e.g: 'PLATFORM:TELESCOPE')
- 'ROOT': The name returned is the full name of the father model (e.g: 'SATEM:PLATFORM')
- 'ROOT-NOMAIN': The name returned is the full name of the father model without the main model name (e.g: 'PLATFORM')
- 'SUBMODEL': The name returned is the model name, without his fathers (e.g: 'TELESCOPE')

**Example of a SUBMDN function call**

\[
\text{NAME} = \text{SUBMDN}(3, 'ALL') \\
\text{NAME} = \text{SUBMDN}(30, 'ROOT') \\
\text{NAME} = \text{SUBMDN}(30, 'SUBMODEL')
\]
NAME = SUBMDN(3, 'ALL-NOMAIN')
NAME = SUBMDN(30, 'ROOT-NOMAIN')

**CHARACTER**256 SUBMOD(NAMETYPE)

- **Input:**
  - NAMETYPE: Character string to specify the name format
- **Output:**
  - CHARACTER*256 SUBMOD: Model name

**Description:**

This function returns the name of the current model. The current model is the model to which the calling subroutine belongs.

The character string NAMETYPE can have the following values:

- **'ALL':** The model name will be returned as a full name (e.g: 'SATEM:PLATFORM:TELESCOPE')
- **'ALL-NOMAIN':** The model name will be returned as a full name but without the main model name (e.g: 'PLATFORM:TELESCOPE')
- **'ROOT':** The name returned is the full name of the father model (e.g: 'SATEM:PLATFORM')
- **'ROOT-NOMAIN':** The name returned is the full name of the father model without the main model name (e.g: 'PLATFORM')
- **'SUBMODEL':** The name returned is the model name, without his fathers (e.g: 'TELESCOPE')

**Example of a SUBMOD function call**

NAME = SUBMOD('ALL')
NAME = SUBMOD('ROOT')
NAME = SUBMOD('SUBMODEL')
NAME = SUBMOD('ALL-NOMAIN')
NAME = SUBMOD('ROOT-NOMAIN')

**INTEGER INTGL(MODEL1,NODE1,MODEL2,NODE2,INDEX)**

- **Input:**
  - INTEGER MODEL1: Model of first node
  - INTEGER NODE1: User first node number
  - INTEGER MODEL2: Model of second node
  - INTEGER NODE2: User second node number
  - INTEGER INDEX: Index of the coupling
- **Output:**
  - INTEGER INTGL: Internal number of the GL coupling

**Description:**

This function returns the internal coupling number between the two nodes. If a coupling has been multi-defined, the index is used to select the coupling index to be returned by the function

**Example of an INTGL function call**

N = INTGL(CURRENT, 102, CURRENT, 103, 1)
N = INTGL(SATEM:TELESCOPE, 1000, CURRENT, 102, 1)

**INTEGER INTGR(MODEL1,NODE1,MODEL2,NODE2,INDEX)**

- **Input:**

---
This function returns the internal coupling number between the two nodes. If a coupling has been multi-defined, the index is used to select the coupling index to be returned by the function.

**Example of an INTGR function call**
N = INTGR(CURRENT, 102, CURRENT, 103, 1)
N = INTGR(SATEM:TELESCOPE, 1000, CURRENT, 102, 1)

**INTEGER INTGF(MODEL1,NODE1,MODEL2,NODE2,INDEX)**

- **Input:**
  - INTEGER MODEL1: Model of first node
  - INTEGER NODE1: User first node number
  - INTEGER MODEL2: Model of second node
  - INTEGER NODE2: User second node number
  - INTEGER INDEX: Index of the coupling
- **Output:**
  - INTEGER INTGF: Internal number of the GF coupling
- **Description:**

This function returns the internal coupling number between the two nodes. If a coupling has been multi-defined, the index is used to select the coupling index to be returned by the function.

**Example of an INTGF function call**
N = INTGF(CURRENT, 102, CURRENT, 103, 1)
N = INTGF(SATEM:TELESCOPE, 1000, CURRENT, 102, 1)

**INTEGER INTGV(MODEL1,NODE1,MODEL2,NODE2,INDEX)**

- **Input:**
  - INTEGER MODEL1: Model of first node
  - INTEGER NODE1: User first node number
  - INTEGER MODEL2: Model of second node
  - INTEGER NODE2: User second node number
  - INTEGER INDEX: Index of the coupling
- **Output:**
  - INTEGER INTGV: Internal number of the GV coupling
- **Description:**

This function returns the internal coupling number between the two nodes. If a coupling has been multi-defined, the index is used to select the coupling index to be returned by the function.

**Example of an INTGV function call**
N = INTGV(CURRENT, 102, CURRENT, 103, 1)
N = INTGV(SATEM:TELESCOPE, 1000, CURRENT, 102, 1)
**SETNDI(ZNODE,ENTITY,VALUE,MODEL)**

- **Input:**
  - ZNODE: Character string describing the nodes concerned
  - ENTITY: Character string describing the nodal entity to be valued
  - VALUE: Integer value to be affected
  - MODEL: Model
- **Output:** none
- **Description:**

  This function affects a value of a given nodal entity to all nodes specified by ZNODE from the model MODEL. Since no integer nodal entity exists by default the function can only be called on a user nodal entity.

  **Example of a SETNDI function call**
  ```
  CALL SETNDI("#300-399", "STATUS", 0, CURRENT)
  ```

  This function has been developed only for compatibility matters with ESATAN. The extended MORTRAN language of THERMISOL may be more convenient to use. The previous example can then be written:
  ```
  STATUS: '#300-399' = 0
  ```

**SETNDR(ZNODE,ENTITY,VALUE,MODEL)**

- **Input:**
  - ZNODE: Character string describing the nodes concerned
  - ENTITY: Character string describing the nodal entity to be valued
  - VALUE: Real value to be affected
  - MODEL: Model
- **Output:** none
- **Description:**

  This function affects a value of a given nodal entity to all nodes specified by ZNODE from the model MODEL. This may be applied to existing nodal entities or to user declared ones.

  **Example of a SETNDR function call**
  ```
  CALL SETNDR("#300-399", "T", 20.0D0, CURRENT)
  ```

  This function has been developed only for compatibility matters with ESATAN. The extended MORTRAN language of THERMISOL may be more convenient to use. The previous example can then be written:
  ```
  T: '#300-399' = 20.0
  ```

**SETNDZ(ZNODE,ENTITY,VALUE,MODEL)**

- **Input:**
  - ZNODE: Character string describing the nodes concerned
  - ENTITY: Character string describing the nodal entity to be valued
  - VALUE: String value to be affected
  - MODEL: Model
- **Output:** none
- **Description:**

  This function affects a value of a given nodal entity to all nodes specified by ZNODE from the model MODEL. Since no string nodal entity exists by default the function can only be called on a user nodal entity.

  **Example of a SETNDZ function call**
  ```
  CALL SETNDZ("#300-399", "STATUS", 'OFF', CURRENT)
  ```
This function has been developed only for compatibility matters with ESATAN. The extended MORTRAN language of THERMISOL may be more convenient to use. The previous example can then be written:

```plaintext
STATUS:'#300-399' = 'OFF'
```

**STORMM(ENTITY, MIN, TMIN, MAX, TMAX)**

- **Input:**
  - ENTITY: Character string describing the nodal entity to be compared
  - MIN: Character string of nodal entity to store minimums
  - TMIN: Character string of nodal entity to store times of minimums
  - MAX: Character string of nodal entity to store maximums
  - TMAX: Character string of nodal entity to store times of maximums
- **Output:** none
- **Description:**

This function is used to store the minimums, maximums and their times of a nodal entity during a transient analysis. The nodal entity referenced by the MIN, TMIN, MAX and TMAX parameters shall be user nodal entities declared by the user. This function shall be called in the $VRESULTS (or $VARIABLES2) block.

**Example of a STORMM function call**

```plaintext
$ENTITIES
  MIN;
  TMIN;
  MAX;
  TMAX;
$NODES
...
$VRESULTS
CALL STORMM('T','MIN','TMIN','MAX','TMAX')
```

**DOUBLE PRECISION GRPMIN(ZNODE, ENTITY, MODEL)**

- **Input:**
  - ZNODE: Character string describing the group of nodes
  - ENTITY: Character string describing the nodal entity to be compared
  - MODEL: Model
- **Output:**
  - GRPMIN: Minimum value
- **Description:**

This function returns the minimum value of the nodal entity specified within the group of nodes described by ZNODE.

**Example of a GRPMIN function call**

```plaintext
TMIN = GRPMIN('#300-500','T',CURRENT)
```

**DOUBLE PRECISION GRPMAX(ZNODE, ENTITY, MODEL)**

- **Input:**
  - ZNODE: Character string describing the group of nodes
  - ENTITY: Character string describing the nodal entity to be compared
  - MODEL: Model
Output:

- **GRPMAX**: Maximum value

Description:

This function returns the maximum value of the nodal entity specified within the group of nodes described by ZNODE.

**Example of a GRPMAX function call**

\[
TMAX = \text{GRPMAX}('#300-500', 'T', \text{CURRENT})
\]

---

### DOUBLE PRECISION GRPSUM(ZNODE,ENTITY,MODEL)

Input:
- **ZNODE**: Character string describing the group of nodes
- **ENTITY**: Character string describing the nodal entity to be summed
- **MODEL**: Model

Output:
- **GRPSUM**: Sum value

Description:

This function returns the sum of all values of the nodal entity specified within the group of nodes described by ZNODE.

**Example of a GRPSUM function call**

\[
\text{TOTAL\_POWER} = \text{GRPSUM}('#300-500', 'QR', \text{CURRENT})
\]

---

### DOUBLE PRECISION GRPAVE(ZNODE,ENTITY,MODEL)

Input:
- **ZNODE**: Character string describing the group of nodes
- **ENTITY**: Character string describing the nodal entity to be averaged
- **MODEL**: Model

Output:
- **GRPAVE**: Average value

Description:

This function returns the arithmetic average value (sum divided by the number of elements) of the nodal entity specified within the group of nodes described by ZNODE.

**Example of a GRPAVE function call**

\[
\text{TAVE} = \text{GRPAVE}('#300-500', 'T', \text{CURRENT})
\]

---

### DOUBLE PRECISION GRPAVE2(ZNODE,ENTITY,MODEL)

Input:
- **ZNODE**: Character string describing the group of nodes
- **ENTITY**: Character string describing the nodal entity to be averaged
- **MODEL**: Model

Output:
- **GRPAVE2**: Quadratic average value

Description:

This function returns the quadratic average value of the nodal entity specified within the group of nodes described by ZNODE.

**Example of a GRPAVE2 function call**

\[
\text{TAVE} = \text{GRPAVE2}('#300-500', 'T', \text{CURRENT})
\]
**DOUBLE PRECISION GRPAVE_W(ZNODE,ENTITY,ENTITY2,MODEL)**

- **Input:**
  - ZNODE: Character string describing the group of nodes
  - ENTITY: Character string describing the nodal entity to be averaged
  - ENTITY2: Character string describing the nodal entity to weight the average
  - MODEL: Model

- **Output:**
  - GRPAVE_W: Weighted average value

- **Description:**

  This function returns the arithmetic weighted average value of the nodal entity specified within the group of nodes described by ZNODE.

  **Example of a GRPAVE_W function call**

  ```
  .TAVE = GRPAVE_W('#300-500', 'T', 'A', CURRENT)
  ```

**DOUBLE PRECISION GRPAVE2_W(ZNODE,ENTITY,MODEL)**

- **Input:**
  - ZNODE: Character string describing the group of nodes
  - ENTITY: Character string describing the nodal entity to be averaged
  - ENTITY2: Character string describing the nodal entity to weight the average
  - MODEL: Model

- **Output:**
  - GRPAVE2_W: Quadratic weighted average value

- **Description:**

  This function returns the quadratic weighted average value of the nodal entity specified within the group of nodes described by ZNODE.

  **Example of a GRPAVE2_W function call**

  ```
  .TAVE = GRPAVE2_W('#300-500', 'T', 'A', CURRENT)
  ```

**DOUBLE PRECISION GETGL(MODEL,NODE1,NODE2)**

- **Input:**
  - MODEL: Model
  - NODE1: Node number of the first node
  - NODE2: Node number of the second node

- **Output:**
  - GETGL: Total conductive coupling between node 1 and node 2

- **Description:**

  This function returns the value of the total conductive coupling between two nodes (that is, it sum over i the GL(node 1, node 2, i)). This function is especially useful for a multi-defined coupling.

  **Example of a GETGL function call**

  ```
  GLTOT = GETGL(CURRENT, N100, N200)
  ```

**DOUBLE PRECISION GETGR(MODEL,NODE1,NODE2)**

- **Input:**
  - MODEL: Model
• **NODE1**: Node number of the first node
• **NODE2**: Node number of the second node

**Output:**
• **GETGR**: Total radiative coupling between node 1 and node 2

**Description:**
This function returns the value of the total radiative coupling between two nodes (that is, it sum over i the GR(node 1, node 2, i)). This function is especially useful for a multi-defined coupling.

**Example of a GETGR function call**

```cpp
GRTOT = GETGR(CURRENT, N100, N200)
```

**DOUBLE PRECISION GETGF(MODEL,NODE1,NODE2)**

• **Input:**
  • MODEL: Model
  • NODE1: Node number of the first node
  • NODE2: Node number of the second node

• **Output:**
  • **GETGF**: Total convective coupling between node 1 and node 2

**Description:**
This function returns the value of the total convective coupling between two nodes (that is, it sum over i the GF(node 1, node 2, i)). This function is especially useful for a multi-defined coupling.

**Example of a GETGF function call**

```cpp
GFTOT = GETGF(CURRENT, N100, N200)
```

### Other routines

**CHARACTER*17 ZDAYDT()**

• **Input**: none

• **Output:**
  • CHARACTER*17 ZDAYDT: Current date

**Description:**
This function returns the current date (information provided by the operating system).

**Example of a ZDAYDT function call**

```cpp
DATE = ZDAYDT()
```

**CHARACTER*8 ZDAYTM()**

• **Input**: none

• **Output:**
  • CHARACTER*8 ZDAYTM: Current time

**Description:**
This function returns the current time (information provided by the operating system).

**Example of a ZDAYTM function call**

```cpp
TIME = ZDAYTM()
```
Solution routines

Steady-State

SUBROUTINE SOLVIT

This subroutine computes the solution of a steady-state problem with the Newton Raphson algorithm as described in the Theoretical chapter.

Driving parameters:

- **RELXCA:** maximum temperature change for one node over one iteration. The criterion is: RELXCC < RELXCA.
- **NLOOP:** maximum number of iterations allowed. The criterion is: LOOPCT < NLOOP.
- **INBALA:** maximum power exchange between diffusive nodes and boundary nodes. The criterion is: ENBALA < INBALA.
- **INBALR:** maximum relative power exchange between diffusive nodes and boundary nodes. The criterion is: ENBALR < INBALR.
- **DAMPT:** damping factor to be applied to all the nodes at each iteration for temperature change \((T^{(n+1)} = T^{(n)} - DAMPT \cdot \Delta T)\). If DAMPT=1, the algorithm will apply some dynamic changes to ensure convergence.

Typical values are: NLOOP=1000, RELXCA=1e-4, INBALA=1e-3, INBALR=1e-5.

In a general way, do not specify any value for DAMPT (or just DAMPT=1 which is equivalent). However, you can try successive values (such as 0.9, 0.5 or 0.1) if no convergence occurs. In this case, please contact our services to enable us to improve the algorithm.

SUBROUTINE SOLVFM

This subroutine computes the solution of a steady-state problem with a Newton-Krylov algorithm as described in the Theoretical chapter.

Driving parameters:

- **RELXCA:** maximum temperature change for one node over one iteration. The criterion is: RELXCC < RELXCA.
- **NLOOP:** maximum number of iterations allowed. The criterion is: LOOPCT < NLOOP.
- **INBALA:** maximum power exchange between diffusive nodes and boundary nodes. The criterion is: ENBALA < INBALA.
- **INBALR:** maximum relative power exchange between diffusive nodes and boundary nodes. The criterion is: ENBALR < INBALR.
- **DAMPT:** damping factor to be applied to all the nodes at each iteration for temperature change \((T^{(n+1)} = T^{(n)} - DAMPT \cdot \Delta T)\). If DAMPT=1, the algorithm will apply some dynamic changes to ensure convergence.

Typical values are: NLOOP=1000, RELXCA=1e-4 (INBALA=0.1 and INBALR=1e-4 may also be used).

In a general way, do not specify any value for DAMPT (or just DAMPT=1 which is equivalent). However, you can try successive values (such as 0.9, 0.5 or 0.1) if no convergence occurs. In this case, please contact our services to enable us to improve the algorithm.
Transient

SUBROUTINE SCRANK

This subroutine computes the solution of a transient problem with the Crank Nicholson algorithm as described in the Theoretical chapter.

Driving parameters:

- **TIMEO**: start date.
- **TIMEND**: end date.
- **DTIMEI**: time step.
- **RELXCA**: maximum temperature change for one node over an iteration, at a given time. The criterion is: \( \text{RELXCC} < \text{RELXCA} \).
- **INBALT**: maximum power exchange by diffusive nodes, including capacitive power. This is a convergence criterion more physical than the RELXCA one. It has been introduced because the definition of the INBALA control variable is not suitable for transient cases. However, each diffusive node has to respect a balanced flux equation taking into account transient phenomena such as capacitive fluxes. The criterion is: \( \text{ENBALT} < \text{INBALT} \).
- **NLOOP**: maximum number of iterations allowed, at a given time. The criterion is: \( \text{LOOPCT} < \text{NLOOP} \).
- **DAMPT**: damping factor to be applied to all the nodes at each iteration of the implicit convergence. If \( \text{DAMPT}=1 \), the algorithm will apply some dynamic changes to ensure convergence.

The values for TIMEO, TIMEND and DTIMEI depend on the problem being modelled. Other typical values are: RELXCA=1e-4, NLOOP=1000 (in general, only 5 to 50 iterations are sufficient). In this routine, the DTMIN, DTMAX and DTPMAX variables are not used.

A special call of the function UPDATE_FLUX in $VRESULT allows to take into account a discontinuous phenomena that happens at a specific time by forcing the start flux of the next time-step to be recomputed and not equal to the end flux of the current time-step.

SUBROUTINE SCRANKAUTO

This subroutine computes the solution of a transient problem with the Crank Nicholson algorithm associated with an automatic time step as described in the Theoretical chapter.

Driving parameters:

- **Same as SCRANK**: TIMEO, TIMEND, RELXCA, INBLAT, NLOOP, DAMPT
- **DTIMEI**: initial time step.
- **ERRMIN**: if the maximum estimated relative error in the model is lower than ERRMIN, the time step will be increased.
- **ERRMAX**: if the maximum estimated relative error in the model is higher than ERRMAX, then the time step is decreased and the solution is recomputed.
- **DTMIN**: minimum time-step allowed.
- **DTMAX**: maximum time-step allowed.

The values for TIMEO, TIMEND and DTIMEI depend on the problem being modelled. Other typical values are: RELXCA=1e-4, NLOOP=1000 (in general, only 5 to 50 iterations are sufficient), ERRMIN=1e-4, ERRMAX=1e-4 (these last two parameters can be equal – the solution will always be as close as possible, but lower, to the desired relative error value).
Cyclic Convergence Routines

SUBROUTINE SOLCYC(SOLNAM, CVTCA, CVDTCA, PERIOD, MAXCYC, ZNODE, OUTIML)

SOLCYC is a 'meta-solver', the purpose of which is to attain a steady cyclic solution: i.e. successive cycles of a transient analysis giving the same thermal results to within user-specified criteria.

**Inputs:**
- SOLNAM : String – transient solver to be used
- CVTCA: Double – Cycle convergence criterion for temperature
- CVDTCA: Double - Cycle convergence criterion for rate of change of temperature (not used)
- PERIOD: Double – Period of cycle (seconds)
- MAXCYC: Integer – Maximum number of cycles
- ZNODE: Specifies nodes for which criteria shall be met
- OUTIML: String – defines whether normal output is required ('NONE' or 'ALL')

**Example of a SOLCYC function call**

$EXECUTION

1. Attain steady cycles
   CALL SOLCYC ('SCRANK', 0.01, 0.0, 6450.0, 10, '#100-200', 'NONE')

2. Run for one more cycle to get output
   CALL SCRANK

SUBROUTINE SCYCLE(SOLNAM, PERIOD, MAXCYC) + SUBROUTINE SCYCLE_ADDSPEC(ZNODE, CVTCA)

SCYCLE is a 'meta-solver', the purpose of which is to attain a steady cyclic solution: i.e. successive cycles of a transient analysis giving the same thermal results to within user-specified criteria.

This routine uses convergence criteria previously defined by one or several calls to the subroutine SCYCLE_ADDSPEC (limited to 32), allowing to specify different convergence levels for several group of nodes.

Moreover, it automatically performs convergence cycles excluding calls to $OUTPUTS block and storing the convergence results into a specific h5 file.

If the cyclic convergence has been successfully reached, a last cycle is performed including calls to $OUTPUTS and storing this last cycle results into the original h5 file.

A detail convergence repporting is written in the convergence control file (csv) plus a summary into the standard output file (out).

**Inputs:** (SCYCLE)
- SOLNAM : String – transient solver to be used
- PERIOD: Double – Period of cycle (seconds)
- MAXCYC: Integer – Maximum number of cycle

**Inputs:** (SCYCLE_ADDSPEC)
- ZNODE: Specifies nodes for which criteria shall be met
CVTCA: Double – Cycle convergence criterion for temperature

Example of a SCYCLE function call

$EXECUTION

1. Specification of convergence criteria
   CALL SCYCLE_ADDSPEC('#100-200', 3.0)
   CALL SCYCLE_ADDSPEC('Equipments', 0.5)

2. Run for one more cycle to get output
   CALL SCYCLE ('SCRANKAUTO', 6450.0, 10)

Wavelength dependency management routines

**DOUBLE PRECISION EPSWLBEF ()**

- **Input:**
  - none
- **Output:**
  - DOUBLE PRECISION EPSWLBEF: equivalent epsilon.

- **Description:**
  This function computes the equivalent epsilon value according to the wavelength discretization and the temperature of the thermal node. The function shall be exclusively used at the nodal description level (the value of the EPS data will then be updated during the execution according to the GENMOR behaviour). The following quantity is returned:

\[
\text{EPSWLBEF}_i = \sum_{m=1,\text{NWBLBANDS}}^{\text{FBAND}(T_i, \lambda_m, \lambda_{m+1}).EPSWL}(m)
\]

Example of a EPSWLBEF function call

D123='Radiator +X', T=20.0, ALP=0.22,
   EPS = EPSWLBEF(), EPSWLB=[0.22, 0.35, 0.45, 0.47, 0.44];

**DOUBLE PRECISION FBAND (T1,L1,L2)**

- **Input:**
  - DOUBLE PRECISION T: Temperature of thermal node
  - DOUBLE PRECISION L1: Lower bound of the wavelength band
  - DOUBLE PRECISION L2: Upper bound of the wavelength band
- **Output:**
  - DOUBLE PRECISION FBAND: fractional emissive power
**DOUBLE PRECISION**  \( \text{FBAND} (T, L1, L2) \)

- **Description:**
  This function computes the fractional emissive power for a given wavelength band. It returns the weight factor of a given wavelength band according to the thermal node temperature. The value of the FBAND function is given by:
  \[
  \text{FBAND}(T, \lambda_m, \lambda_{m+1}) = \text{BBFN}(\lambda_{m+1}, T) - \text{BBFN}(\lambda_m, T)
  \]
  where the wavelength band \( m \) is defined by the two bounds \( \lambda_m, \lambda_{m+1} \) and the BBFN function is given by Michael F. Modest, *Radiative Heat Transfer - Second Edition*, 2003, Academic Press, ISBN 0-120503163-7

**Example of a FBAND function call**

```fortran
DO IW=1, NWLBANDS
  EQUIVEPS = EQUIVEPS + FBAND(T100, WLXBANDS(WLB=IW), WLXBANDS(WLB=IW+1)) * EPSWLBEF(WLB=IW)
ENDDO
```

This example computes the equivalent epsilon value of the node 100 which can be obtained with the function EPSWLBEF().

---

**Theoretical Background**

**Mathematical Formulation of the Thermal Problem**

THERMISOL deals with a specific class of mathematical equations related to thermal analysis, especially adapted for the modeling of conduction and radiation but extendable to more complex applications.

The mathematical formulation is a system of differential equations:

\[
M \begin{cases}
  C_i \frac{dT_i}{dt} - \alpha \sum_j G_{Rij}(T_i^4 - T_j^4) + \sum_j G_{Lij}(T_i - T_j) + \sum_j G_{Fij}(T_i - T_j) + Q_{Si} + Q_{Ai} + Q_{Bi} + Q_{Ii} + Q_{Ri}
\end{cases}
\]

where:
- \( T_i \) is the temperature of node \( i \), to be computed
- \( t \) is the time
- \( C_i \) is the capacitance (M.Cp) of node \( i \)
- \( G_{Rij} \) is a symmetric radiative coupling between nodes \( i \) and \( j \)
- \( G_{Lij} \) is a symmetric linear coupling between nodes \( i \) and \( j \)
- \( G_{Fij} \) is a one-way linear coupling between nodes \( i \) and \( j \)
- \( Q_{Si} \) is the solar power absorbed by node \( i \)
- \( Q_{Ai} \) is the planet albedo power absorbed by node \( i \)
- \( Q_{Bi} \) is the planet infra-red power absorbed by node \( i \)
- \( Q_{Ii} \) is an internal power absorbed by node \( i \)
- \( Q_{Ri} \) is an additional power absorbed by node \( i \).

At several moments (at each iteration or at each time step), the user can modify any of these data, using a Fortran-like language. This can be useful to integrate the variations of external fluxes, the activation of heaters or...
dissipation of equipment, etc. It can also be used to simulate different classes of mathematical equations; e.g.,
linear couplings can be changed according to the temperature to model convective exchanges.

## Numerical Methods for Steady-State Problems

### Newton algorithm

The **Newton method** is a general method for solving non-linear systems written as the follow:

\[ f(\mathbf{x}) = 0 \]

Starting from an initial data \( \mathbf{x}_0 \), the Newton method builds a series \( \{\mathbf{x}^{(k)}\}_{k=0}^{N} \) of approximations of \( \mathbf{x} \).

Supposing that 2 successive approximations are close enough, the Taylor development of the function to the first order gives:

\[
 f(\mathbf{x}^{(k+1)}) = f(\mathbf{x}^{(k)}) + \nabla f(\mathbf{x}^{(k)}) \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} + \mathbf{o}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})
\]

We can then define the series \( \{\mathbf{x}^{(k)}\}_{k=0}^{N} \) as:

\[
\begin{cases}
\mathbf{x}^{(0)} = \mathbf{x}_0 \\
\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \nabla f(\mathbf{x}^{(k)}) \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}
\end{cases}
\]

The construction of this series stops when the convergence criterion is reached.

At each iteration \( k \), the algorithm shall compute the following steps:

1. Compute the Jacobian matrix \( \nabla f(\mathbf{x}^{(k)}) \)
2. Solve the system \( \mathbf{A} \mathbf{s}^{(k)} = -f(\mathbf{x}^{(k)}) \) then deduce the new solution \( \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{s}^{(k)} \)
3. Evaluate \( f(\mathbf{x}^{(k+1)}) \)

This method may be optimized by introducing a damping factor \( \beta \) in the evaluation of the new solution at the second step \( \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \beta \mathbf{s}^{(k)} \) in order to accelerate the convergence.

In order to avoid the "over-solving", the iteration of the Newton method may be approximated with an increasing accuracy. It is also possible to simplify the Jacobian matrix.

As it has been presented, the Newton method consists to solve a non-linear problem by successive linear system that tends to the non-linear solution. This process of convergence requires then to solve linear systems. This may be done by different approaches:

The algorithms stop when the maximum temperature change during an iteration (RELXCC) is less than the specified criterion (RELXCA):

\[
RELXCC^{(n)} = \max_i \left| T_i^{(n)} - T_i^{(n-1)} \right| \leq RELXCA
\]

Then, the global equilibrium is verified, considering incoming and outgoing fluxes (due to internal powers and exchanges with boundary nodes). This difference is called "ENBALA" and has to be less than the "INBALA" user parameter. The relative balance "ENBALR" is also evaluated and compared with the user parameter "INBALR".

\[
ENBALA = \left| \Phi_{in} - \Phi_{out} \right| \leq INBALA
\]

\[
ENBALR = \frac{ENBALA}{\max \left( \Phi_{in}, \Phi_{out} \right)} \leq INBALR
\]

\[
ENBALT = \sum_i f_i \left( T_i \right) \leq INBALT
\]

In all cases, the algorithm will be stopped if the number of iterations (LOOPCT) has reached the maximum value.
allowed by the user (NLOOP).
To deal with the convergence of the model with more information, the RELXCC, ENBALA and ENBALR outputs are signed.

**Newton Raphson algorithm (SOLVIT)**

The iterative Newton-Raphson algorithm, implemented for thermal resolution under the THERMISOL function SOLVIT consist of solving for each line i of linear system matrix the solution $x_i^{(k+1)}$ according to the data

$$
\begin{bmatrix}
  x_i^{(k+1)} \\
  x_j^{(k+1)} \\
  \vdots \\
  x_{2i}^{(k+1)}
\end{bmatrix}_{\frac{1}{2}A}
\begin{bmatrix}
  x_i^{(k)} \\
  x_j^{(k)} \\
  \vdots \\
  x_{2i}^{(k)}
\end{bmatrix}_{\frac{1}{2}A}
$$

This method is very efficient for thermal problems since all temperature variables are widely coupled to each other with radiative links. However, for more sparse problems or purely linear problems, it becomes much less efficient than a matricial method.

For each iteration, the system of N equations is considered as N independent equations of 1 variable. For each equation (i), the unknown variable is $T_i$ and all the $T_j$ (ji) are considered as fixed values:

$$f(T_i) = \sum_j a_{ij} (T_j - T_i^0) + \sum_j b_{ij} (T_j - T_i) + c_i = 0$$

Then, $T_i$ is modified as per the Newton’s method:

$$T_i^{(n+1)} = T_i^{(n)} - \frac{f(T_i^{(n)})}{f'(T_i^{(n)})}$$

1. Graphical representation of the Newton Raphson algorithm

Where: $d$ is a damping factor (DAMPT variable in the language).

If the user does not specify the value of DAMPT, the solver modifies it dynamically during the solution to allow a fast and robust convergence.

At the end of each iteration, when all $T_i^{(n)}$ have been computed, the solver calls the user subroutine $\text{SVTEMPERATURE}$ (or $\text{SVARIABLES1}$) in which the $f$ function can be updated (for couplings, fluxes or other data depending on the temperature).

**Newton-Krylov algorithm (SOLVFM)**

Originally, the SOLVFM routine was based on Cholesky decomposition. This matricial approach solves directly the linearized system.

In that case, the **inexact Newton method** allows replacing the Jacobian matrix $A$ by a symmetric matrix $A^s$ in order to use a Cholesky factorization, less time-consuming than a LU factorization.

The **Krylov** methods work by forming a basis from a matrix $A$ and a vector $b$:

$$K_r(A,b) = \left\{ b, Ab, A^2b, \ldots, A^{r-1}b \right\}$$

The approximations to the solution are then formed by minimizing the residual formed over the increasing subspace created.

Ref: UM.000040364.AIRB © Airbus Defence and Space SAS 2023 - All rights reserved 97
This method is suitable for large sparse systems that are difficult to be handled by the Cholesky decomposition. Both generalized minimum residual (GMRES) and conjugate gradient (CG) have been tested. It resulted that CG gave better results in the context of thermal or electrical problems. The **Newton-Krylov** method is the one now being implemented in the thermal resolution SOLVFM. This method requires a pre-conditioner which is given by a partial sparse Cholesky decomposition. This pre-conditioner is updated not at each iteration but when necessary. This decomposition has then became an incomplete decomposition used as a pre-conditioner for a conjugate gradient resolution.

### Numerical Methods for Transient Problems

#### Crank Nicholson algorithm

The Crank Nicholson Algorithm is an implicit method based on the evaluation of the equations in the middle of two consecutive time steps \((n+\frac{1}{2})\).

At this time \((n+\frac{1}{2})\), the temperature derivative is given by a centered evaluation:

\[
\frac{\partial T}{\partial t} \bigg|_{n+\frac{1}{2}} \approx \frac{T^{n+1} - T^n}{\Delta t}
\]

and the flux is evaluated by:

\[
f^{n+\frac{1}{2}} \approx \frac{f^{n+1} + f^n}{2}
\]

Thus, the thermal equation giving \(T^{n+1}\) as a function of \(T^n\), is the following implicit scheme:

\[
\varphi_i = C_i \left( \frac{T_i^{n+1}}{\Delta x_i^2} - \frac{2}{\Delta x_i^2} T_{i-1}^{n+1} \right) - \frac{f^{n+1}(\nabla_i T_{n+1})}{\Delta t} + f^n(\nabla_i T^n) = 0
\]

where:

\[
f^i(t) = \sum_j \alpha_j (T_j^n - T_i^n) + \sum_j \lambda_j (T_j^n - T_i^n) + \omega_i
\]

Then, the solver computes all the \(T_i^{n+1}\) values with a Newton Raphson algorithm, controlled by the NLOOP and RELXCA. These variables have the same meaning as in the steady-state problem. Arithmetic nodes (without capacitance) are solved using the Newton-Raphson algorithm (see SOLVIT).

As for steady-state cases, a criterion based on diffusive nodes equilibrium has been set, INBALT, defined by:

\[
\text{ENBALT} = \sum \varphi_i(T_i) \leq \text{INBALT}
\]

#### Crank Nicholson with automatic time step

The time stepping can be automatically changed during the solution. Since the Crank Nicholson algorithm is a 2\(^{nd}\) order scheme, the error can be accurately estimated by the 3\(^{rd}\) term of the Taylor temperature development.
Two user variables, ERRMIN and ERRMAX, are used to modify the time step, leading to 3 different cases:

- \( \_\_\text{ERRMIN } \text{err } \text{ERRMAX} \): the iteration is correct and the time step is not changed.
- \( \_\_\text{err } \text{ERRMIN} \): the iteration is correct but the time step is increased.
- \( \_\_\text{err } \text{ERRMAX} \): the iteration is cancelled and computed again with a smaller time step.

The first time-step is initially adjusted to \( 0.5 \times \text{CSGMIN} \), where CSGMIN is the minimum of all CSG coefficients.
Skeleton

THERMICA / THERMISOL link

The skeleton is mainly used to create a simple interface between THERMICA and THERMISOL. Its goal is to collect the network files coming from THERMICA and to include them into a standard skeleton of the input file dedicated to the THERMISOL temperature solver. The skeleton module can be used to create a skeleton of an input file for THERMISOL. The user specifies the options to be written, the values of the control variables and the resolution function to call. He also specifies the files to be included as reading instructions. In a second stage, the skeleton module expands the file in order to create a complete THERMISOL input.

Usage and interest

Interactive mode

In the interactive mode (i.e. into SYSTEMA), the skeleton module is used as a simple interface between THERMICA and THERMISOL giving the possibility to the user to add customized instructions from a file written by himself. The interest of such an interface is that the user can write common instructions into a single file that will be taken into account automatically in the successive calls of THERMICA/THERMISOL processes. It is only requested to select that file without needing to modify manually the THERMISOL input file.

Batch mode

The skeleton file is also very convenient to use when running the temperature solver in batch command. A standard skeleton wrote by the user can then be re-used to expand many different cases just by changing the references to the files containing the couplings or different instructions. An "easy batch" mode is also available to directly translate a ske file into a dck file. Read the batch mode section for more details.

Reading instructions

Contextual expansion

To reference the different files, the skeleton uses special reading instructions. Those instructions will expand the content of a given block from the file to read where the reading instructions are placed.

There are 5 types of reading instructions:

- #READ
- #READ-MODELNAME
- #READ-SUBMODEL
- USE
- #READ-n where n is an integer from the range 1 to 256
- #USE-n where n is an integer from the range 1 to 256
Standard READ

The #READ is used to basically expand the content of one block. When this command is found under a specific block into the skeleton file, the expansion module will replace it by the content of that block from the file to be read. If the file referenced by a #READ has no context (no blocks are written into the file), it is completely expanded at its call.

**Example**

If the file specified by the #READ does not contain any block, it is then supposed to be entirely expended where it is called.

### Modelname READ

This type of READ instruction has to be exclusively used after a $MODEL in order to get the model name from a specific input file (usually the nodal description).

This feature was used by default with THERMICA-THERMISOL v4.5. Now the model name is usually directly set from the skeleton parameters and the nodal description does not contain the main model name anymore (which avoids the use of this option).

### Sub-model READ

The sub-model READ instruction has to be placed under a $MODEL block.

Compared to a standard READ (or a USE function), it is specialized to expand sub-models with the possibility to select which blocks shall be included.
To select the blocks of a submodel expansion, the syntax is as follow:

```
#READ-SUBMODEL [$BLOCK1, $BLOCK2, ...] filename
```

In case the model is structured with sub-models, it is recommended to expand each sub-model individually and to merge them into the main model by a submodel inclusion.

**USE statement**

The `#USE` command is a new helpful command to be specified at the beginning of the skeleton file. It means that the file specified by the `#USE` instruction has to be extended in any block of the skeleton file if it matches a block of the file to be extended. In fact, it replaces repetitions of the `#READ` instructions in the skeleton file by a single statement at its beginning.

**Parametric READ or USE**

*Batch mode with a parameter file only*

The basic `#READ` or `#USE` instructions are followed by a filename. Rather than explicitly referencing a given file, it is possible to replace it by a file index between 1 and 256. In order to re-use a skeleton for different cases, it can be necessary to change the reference of the files to be expanded. This change could be done directly in the skeleton file but it may be more convenient to specify the reference of the files at an upper level so the skeleton file never has to be changed.

To call a parametric READ, the instruction is given by `#READ-n` where `n` is an integer between 1 and 256. The references shall then be given in the batch command parameter file using the keywords `READ-n`.

**Skeleton Inputs Outputs**

**Inputs**

**SKE file**

If the SKE file is specified in the Inputs (in the batch mode, it is considered as an input if the file exist, otherwise it is considered as an output), then the Skeleton program will use this skeleton file to create the input solver file (DCK file).

If no SKE file is specified (or if the SKE file specified in the batch mode does not exist), the Skeleton program will build a generic skeleton file.

**NWK files**

In the case of a skeleton generation, the nwk files specified as inputs will be added as reading instruction into the ske file.

A simple expansion of a ske file into a dck one does not require the nwk files to be specified in the inputs if they are in the result folder. Indeed, the batch mode does not usually need those files to be expressed as parameters. However, the interactive process, which is executed into a deported result folder, needs those arguments to be specified.
Outputs

SKE file
If specified as an output, a generic skeleton file will be created (except if one is specified as an input).

DCK file
If specified, this file contains the expanded skeleton, i.e. the THERMISOL temperature solver input file.
Posther

Introduction

POSTHER is a tool used to export data and to perform some analysis. The results are exported to both a XLS file (to be read in Excel® or any other similar tool) and to a text file. The data requested by POSTHER are specified in the h5 result file from THERMISOL under a "POSTHER" group. The kind of analysis performed by POSTHER can be extended to any user's need thanks to an API (available for FORTRAN and C++).

Nodes specifications

Whenever the user can select a group of nodes, the general syntax is the same one than the 'ZNODES' used in some THERMISOL output routines:

- **ALL** to select all the nodes
- **@[model path]** to select the nodes of a given sub-model and its sub-models
- **#X** to select the node X
- **#@[model path]:X** to select the node X in a specified submodel
- **#X-Y** to select the range of nodes between X and Y
- **#@[model path]:X-Y** to select the range of nodes between X and Y from a specified submodel
- **#X, Y, [model path]:Z-T** to select nodes X, Y and nodes from Z to T of a sub-model
- **[label]** to select all the nodes with the given label in their name
- **@[model path]:[label]** to select the nodes with a given label from a sub-model

A group can be the concatenation of many specification separated by ;

'#!100-199; @Equipment' means all the nodes between 100 and 199 plus the nodes having 'Equipment' in their names.

If the character '!' is placed before one specification, then it is considered as exclusion:

'ALL;#!100-199' means all the nodes except the nodes between 100 and 199.

Analyses Modules

Extract

The « extract » module creates both xls and text reports on which it is possible to choose the data to be written. First, the user can select a group of nodes to be output as well as a time range. The selected data will be classified in two tables: the first one for constant data (or data that have been stored as constant in the h5 file – using the H5_RES0 control parameter of THERMISOL), the second one for time dependant results. The data that can be selected for the module are

- Areas
- Capacitances
- Emissivities (epsilon)
- Absorptivities (alpha)
- Temperatures
- Solar Fluxes
- Albedo Fluxes
- Planet IR Fluxes
- Internal Dissipations
• Residual Fluxes

If the number of column to write in the xls file is too large then this file may be split in different files. The option EXCEL_COL can be set to define the maximum number of columns of the files.

If a time occurs more than once (for example if a steady-state has been called before the transient resolution, time 0 may be stored twice in the h5 file), only the last one is exported and taken into account.

**Node Min/Max**

The « node min/max » analysis exports, for each time stored in the h5 file and for each entity selected:

- its minimum value, on which node it is found
- its maximum value, on which node it is found
- the mean value on the selected group of node

The mean temperature is computed using the area of the nodes if they are stored in the h5 file. The means on the fluxes (which are actually powers), as well as the temperature’s mean in case no areas are found in the h5 file, are algebraic means (the sum divided by the number of summed values).

1. Node Min/Max Excel sheet
1. Time Min/Max Text sheet

**Time Min/Max**

The « time min/max » analysis exports, for each selected node and for each entity selected:

- its minimum value, at which time it is found
- its maximum value, at which time it is found
- the mean value over the time range specified

The mean values are computed taking into account the time-steps variations.
1. Time Min/Max Excel sheet

**Flux Budget**

**Exchange Flux**

The flux budget analyses all the fluxes across the frontiers between 2 groups of nodes noticed G1 and G2:

- Direct radiative flux through the radiative links between thermal nodes of each group
- Direct conductive flux through the conductive links between thermal nodes of each group
- Direct conductive flux through the common edges of the 2 groups
- Direct convective flux through the convective links between thermal nodes of each group

Each category is subdivided with:

- flux from G1 to G2 (with a negative sign)
- from G2 to G1 (with a positive sign)
- exchanged between the 2 groups (sum of the 2 previous, with a negative sign if the total radiative flux goes from G1 to G2 and a positive sign otherwise)

Except for the conductive flux through the edges for which the direct flux is physically known between the group and its frontier. In case a third group of nodes is also connected to the common frontier of G1 and G2, the third flux balancing the frontier is also given

- flux from G1 to its frontier with G2 (with a negative sign)
- flux from the frontier to G2 (with a positive sign)
- flux from the frontier to the other connected nodes G0 (sum of the 2 previous)
1. Flux analyses: group definition

The two groups of nodes are exclusives. If the group B contains nodes already defined in group A, those are removed from group B. When the model contains conductive interfaces (i.e. edge nodes), the group of nodes A and B are closed by the edges directly connected to the groups. The specification of the groups can so be done without taking into account the edge nodes belonging to a group. Another advantage of this specification is that it allows the two groups to overlap (some edges can be shared by the two groups).

1. Flux analyses: edge nodes closure

In most case the group B is defined as "the model but group A (and edges if any)". This configuration leads to study all the fluxes going in or out from group A (closed by its edges).

**Flux analyses between groups**

If the union of the two groups (closed by their edges) is not the entire model, then the definition of the fluxes between the groups is "the fluxes exchanged by a direct link or a direct contact". This definition is different from
the other possible definition which is "the equivalent fluxes between the two groups".

1. Flux analyses between two groups

The user needs to be aware of that definition because in the following example, the interpretation of the fluxes should be considered carefully.

1. Flux analyses with groups: example of a bar

The results returned from this configuration will give zero conductive flux between the groups even if there is a temperature gradient and an obvious equivalent conductive coupling between A and B. The conductive flux received by A is transmitted by the rest of the model and not the group B. If groups A and B are interconnected with other nodes, the flux exchanged between A-B is the flux directly exchanged between the 2 groups. In this case, the interconnected nodes that do not belong to group A or B have an influence on this flux (in fact the fluxes have to be considered all together). In the report, the influence of the "other
"Flux Budget Excel sheet" is also given.
Group Balance

The group balance export the budget of all powers related to that group (with eventually its enclosure in case a EDGE node group is specified).

Are then exported:

- The total absorbed, evacuated powers
- The extern powers Solar, Albedo, Planet IR, Internal dissipation and Residual powers
- The absorbed and evacuated radiative powers
- The absorbed and evacuated conductive powers
- The absorbed and evacuated convective powers

1. Group Balance Excel sheet

Radiative Budget

The radiative budget exports for the specified group of nodes:

- **Area**
  
  \[ \text{Area} = \sum_i A_i \]

- **Equivalent Emissivity**
  
  \[ \bar{\varepsilon} = \frac{\sum_i A_i \varepsilon_i}{\sum_i A_i} \]

- **GR sum**
\[ SGR = \sum_i \sum_j G_{ij} \]

- **Linear Temperature**
\[ T_{lin} = \frac{\sum_i A_i T_i^4}{\sum_i A_i} \]

- **Quadratic Temperature**
\[ T_{quad} = \sqrt{\frac{\sum_i A_i \varepsilon_i T_i^4}{\sum_i A_i \varepsilon_i}} \]

- **Black-Body Tsink (°C)**
\[ T_{Tsink} = \left( \frac{\sum_i \left( \sum_j \left( \sigma \cdot G_{R, i,j} \left( T_j^4 - T_i^4 \right) \right) \cdot Q_{S_i} + Q_{A_i} + Q_{E_i} + \varepsilon_i A_i T_j^4 \right) \cdot \varepsilon_i A_i}{\sum_i \sigma \cdot \varepsilon_i A_i} \right)^{\frac{1}{4}} \]

- **Rejection Power (W)**
\[ RP = \sum_i \left( \sum_j \left( \sigma \cdot G_{R, i,j} \left( T_j^4 - T_i^4 \right) \right) \cdot Q_{S_i} + Q_{A_i} + Q_{E_i} \right) \]

- **Rejection Flux (W/m²)**
\[ RF = \frac{RP}{\sum_i A_i} \]

---

**Radiative Budget**

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Area</th>
<th>Equivalent emittance</th>
<th>GR coeff</th>
<th>Linear Temperature</th>
<th>Quadratic Temperature</th>
<th>Black-Body Tsink (°C)</th>
<th>Rejection Power (W)</th>
<th>Rejection Flux (W/m²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.03</td>
<td>0.03</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
<tr>
<td>1</td>
<td>0.02</td>
<td>0.02</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.01</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>0.0</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>0.1</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
<tr>
<td>5</td>
<td>0.2</td>
<td>0.2</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
<tr>
<td>6</td>
<td>0.3</td>
<td>0.3</td>
<td>2.10E-14</td>
<td>-3.34E-18</td>
<td>234.74</td>
<td>23.55</td>
<td>302.44</td>
<td>0.0012</td>
</tr>
</tbody>
</table>

---

1. Radiative Budget Excel sheet

Ref: UM.000040364.AIRB © Airbus Defence and Space SAS 2023 - All rights reserved 112
**Temperature Initialization**

This mode is only available in batch mode. It exports the node temperatures into a file as a "$INITIAL" block to be integrated into a DCK file (directly or through the Skeleton module).
B-Plot

Introduction

B-Plot is a postscript document writer. It reads instruction from a file and export graphs into a ps document. As POSTHER, this module is a post-processing tool which is based on the h5 output of THERMISOL.

Principle

B-Plot is based on the following principles: a graph is described by instructions written in a text file called the B-Plot command file. Those instructions respect a language described in the next paragraph. A document is made of several graphs and each graph is made of several curves. A curve is given by a node specification and an entity (T, QS, QA, QE, QR or QI). It is possible to plot temperature and power curves into a same graphic. If this is the case, B-Plot will automatic display 2 y axes (one on the left and one on the right of the graph). The use of 1 y axe or 2 y axes can also be manually managed as well as the axes labels, factors, curve titles, graph titles… In this version of B-Plot, it is possible to choose between 2 page formats. The default format corresponds to 1 graph per page in landscape orientation. The other possibility is to specify 2 graphs per page in portrait orientation.

Limitations

B-Plot authorize the maximal number of declaration of:

- 5 input files (h5 results from THERMISOL Temperature Solver runs)
- 10 curves per graph

The numbers of graphs, the total number of nodes and the length of the time vector have no limit.

B-Plot command file

Keywords

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM</td>
<td>n</td>
<td>First page number</td>
</tr>
<tr>
<td>NGRAPH</td>
<td>n</td>
<td>Number of graphs per page (1 or 2)</td>
</tr>
<tr>
<td>F</td>
<td>n , file</td>
<td>Input File (h5 format)</td>
</tr>
<tr>
<td>T</td>
<td>n [ text ]</td>
<td>Title (or case) given to a result file</td>
</tr>
<tr>
<td>@</td>
<td>text</td>
<td>New graph – graph title</td>
</tr>
<tr>
<td>!</td>
<td>n , Xn , [ $nom ]</td>
<td>New curve – curve specifications</td>
</tr>
</tbody>
</table>
**Command sequences**

Here is an example of a B-Plot command sequence.

1. comment: a B-Plot command sequence
2. Input data
   
   | F1 ... |
   | T1 ... |
   | F2 ... |
   | T2 ... |

3. General definitions
   
   | NGRAPH... |
   | NUM... |
   | XLABEL ... |
   | XFACT... |

4. Graph 1:
   
   | @... |
   | AXEX... |
   | AXEY... |
   | YLABEL... |
   | AXEY2... |

5. Curve 1 (of graph 1)
   
   | !... |

6. Curve 2
   
   | !... |

7. Graph 2:
   
   | @... |
   | ... |
   | ... |
Keywords definitions

**AXEX**

**AXEX xmin, xmax [ , xdelta [ , xoffset ] ]**

**OPTIONAL**
Set the minimum and maximum values of the X axis

**Default values**
The range is set to the minimum and maximum values of the time reference
The offset is set to zero

**Déclaration :**

- Into the general definition (before any @ command). Set the default values to apply to each graph
- Into one graph definition (after a @ command). Locally overload the default value.

| Limitation : no |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmin</td>
<td>Real</td>
<td>Minimum value of X to plot</td>
</tr>
<tr>
<td>xmax</td>
<td>Real</td>
<td>Maximum value of X to plot</td>
</tr>
<tr>
<td>xdelta</td>
<td>Real</td>
<td>Grid spacing</td>
</tr>
<tr>
<td>xoffset</td>
<td>Real</td>
<td>Offset of X values</td>
</tr>
</tbody>
</table>

**AXEY / AXEY1 / AXEY2**

**AXEY / AXEY1 / AXEY2 ymin, ymax**

**OPTIONAL**
Set the minimum and maximum values of the Y axis

**Default values**
The range is set to the minimum and maximum values of the curves relative to that axis

**Déclaration :**

- Into the general definition (before any @ command). Set the default values to apply to each graph
- Into one graph definition (after a @ command). Locally overload the default value.

| Limitation : no |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymin</td>
<td>Real</td>
<td>Minimum value of Y to plot</td>
</tr>
<tr>
<td>ymax</td>
<td>Real</td>
<td>Maximum value of Y to plot</td>
</tr>
</tbody>
</table>
**F**

\[ F_n \text{ file} \]

**Input file declaration**

**Déclaration** :

- Into the general definition (before any @ command). |

**Limitation** : The input file has to be a h5 output from THERMISOL
5 files maximum can be declared

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td><em>1 Integer 5</em></td>
<td>File number</td>
</tr>
<tr>
<td>file</td>
<td>String</td>
<td>Filename (with its relative path)</td>
</tr>
</tbody>
</table>

**NUM**

\[ NUM \ n \]

**OPTIONAL**

First page number

**Default value**

The first page starts with number 1

**Déclaration** :

- Into the general definition (before any @ command). |

**Limitation** : The input file has to be a h5 output from THERMISOL
5 files maximum can be declared

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Integer</td>
<td>First page number</td>
</tr>
</tbody>
</table>

**NGRAPH**

\[ NGRAPH \ n \]

**OPTIONAL**

Number of graphs per page

**Default value**

One graph per page (LANDSCAPE format)

**Déclaration** :

- Into the general definition (before any @ command). |
**Limitation**: There can be only 1 or 2 graphs per page

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td><em>1 Integer 2</em></td>
<td>Number of graphs per page</td>
</tr>
</tbody>
</table>

**T**

**T n text**

Set the title of an input file to be reported into the curve legends

**Déclaration**: 

- Just after a F command

**Limitation**: See the F command

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td><em>1 Integer 5</em></td>
<td>File number</td>
</tr>
<tr>
<td>text</td>
<td>String</td>
<td>Title of the file</td>
</tr>
</tbody>
</table>

**XFACT / YFACT / Y1FACT / Y2FACT**

**XFACT / YFACT / Y1FACT / Y2FACT val**

**OPTIONAL**
Apply a factor to the values displayed from the values stored in the result file

**Default value**

1.0

**Déclaration**: 

- Into the general definition (before any @ command). Set the default values to apply to each graph
- Into one graph definition (after a @ command). Locally overload the default value.

**Limitation**: None

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Real</td>
<td>Factor to be applied on the axis</td>
</tr>
</tbody>
</table>
Remark
The time is output in seconds if XFACT is set to 1.
To convert the time into minutes, XFACT shall be set to 1.6667e-2 hours, 2.7778e-4 days, 1.1574e-5
The other default units (corresponding to a factor of 1.0) are
Temperatures Degre Celsius
Power Watt

XLABLE / YLABE / Y1LABE / Y2LABE
XLABLE / YLABE / Y1LABE / Y2LABE text

OPTIONAL
Set a title for the axis
Default value
X axis Time (seconds)
Y1 axis Temperature (Celsius) or Power (W) depending on the entities plotted
Y2 axis Power (W) if both temperatures and powers are displayed in one graph

Déclaration :

• Into the general definition (before any @ command). Set the default values to apply to each graph
• Into one graph definition (after a @ command). Locally overload the default value.

Limitation : None

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>String</td>
<td>Title of the axis</td>
</tr>
</tbody>
</table>

!
!
! n, X node, $label

Define a curve

Déclaration :

• Into one graph definition (after a @ command).

Limitation : Only 10 curves per graph can be plotted

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td><em>1 Integer 5</em></td>
<td>File number of the data</td>
</tr>
</tbody>
</table>
### New graph definition

**Déclaration**:

- After general definitions
- After a previous graph

**Limitation**: None

<table>
<thead>
<tr>
<th>Paramètre</th>
<th>Valeurs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>String</td>
<td>Graph title</td>
</tr>
</tbody>
</table>

Page left intentionally blank
Batch Mode

Introduction

THERMISOL is a package of applications which is convenient to call directly from a batch command. This batch mode should allow the user to completely define all the required computations. It should be easy to use and all the options should not necessarily be defined if they are not different from the common default values.

The batch program will execute each paragraph defined in the command file.

A paragraph is defined by the keyword "$" followed by the name of the module to be executed.

There are 4 modules in THERMISOL which are:

- $SKELETON
- $SOLVER
- $POSTHER
- $BPLOT

A command file can include as many occurrences of those modules.

Expansion of a skeleton file

Another use of the skeleton module is to expand a skeleton file into a THERMISOL input file (usually written with dck extension).

This operation may be a simple expansion not requiring any extra parameters or may use specific options which shall be described in a parameter file.

Direct expansion

The direct expansion can be called with the following syntax:

[path] ThermisolLNX –e file.ske on Linux systems

Or [path] ThermisolWIN.exe –e file.ske on Windows systems

It will automatically create the THERMISOL input file.dck.

Expansion with parameters

In case the user needs more advanced options, it is possible to use a parameter file.

In the parameter file, it is then possible to assign file names to the parametric #READ-n instructions.

The call is then:

[path] ThermisolLNX parameters.txt on Linux systems

Or [path] ThermisolWIN.exe parameters.txt on Windows systems

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># Inputs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKE</td>
<td>String</td>
<td>NEEDED</td>
<td>Name of the skeleton file</td>
</tr>
<tr>
<td>READ-n</td>
<td>String</td>
<td>&quot; &quot;</td>
<td>File reference for parametric expansion (n = 1 to 256)</td>
</tr>
</tbody>
</table>
It is also possible to complete this command with the following options:

- `--ske_inputs_dir [directory]`
  This option allows specifying a directory where the files to be expanded are located.

- `--dck_outputs_dir [directory]`
  This option specifies where to create the dck file. If the parameter file contains a second block of instructions to execute the temperature solver, it will be automatically executed in that directory. If the path specified does not exist, it will be automatically created.

### Module paragraph "SOLVER"

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCK</td>
<td>String</td>
<td>NEEDED</td>
<td>Name of the DCK file</td>
</tr>
<tr>
<td>OUT</td>
<td>String</td>
<td>&quot; &quot;</td>
<td>Name of the text result file</td>
</tr>
<tr>
<td>HDF</td>
<td>String</td>
<td>&quot; &quot;</td>
<td>Name of the h5 result file</td>
</tr>
<tr>
<td>CSV</td>
<td>String</td>
<td>&quot; &quot;</td>
<td>Name of the convergence file</td>
</tr>
<tr>
<td>LOG</td>
<td>String</td>
<td>&quot; &quot;</td>
<td>Name of the log file</td>
</tr>
<tr>
<td>TASK</td>
<td>Integer</td>
<td>0</td>
<td>Specifies the tasks to be achieved:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 – All</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 – Translate Mortran</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 – Compile Fortran</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 – Link with the solver library</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 – Execute resolution</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Flag</td>
<td>No</td>
<td>Creates extra data for debugging</td>
</tr>
<tr>
<td>IMPLICIT_NONE</td>
<td>Flag</td>
<td>No</td>
<td>Specifies Fortran routines to be &quot;IMPLICIT NONE&quot;</td>
</tr>
</tbody>
</table>
Direct batch call to the solver

As for the expansion, it is possible to call directly the solver module without specifying a parameter file. Use the following syntax; ThermisolXXX –d example.dck

Module paragraph "POSTHER"

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># Inputs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT</td>
<td>String</td>
<td>NEEDED</td>
<td>Name of the temperature results (h5 format)</td>
</tr>
<tr>
<td># Outputs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td>String</td>
<td>&quot;-&quot;</td>
<td>Name of the output file</td>
</tr>
<tr>
<td># Options</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td>String</td>
<td>&quot;EXTRACT&quot;</td>
<td>Specifies the type of post-processing: &quot;Extract&quot; &quot;Tinit&quot; &quot;NodeMinMax&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;TimeMinMax&quot; &quot;Flux&quot; &quot;Balance&quot; &quot;RadBudget&quot;</td>
</tr>
<tr>
<td>NODE_SPEC</td>
<td>String</td>
<td>&quot;ALL&quot;</td>
<td>Nodal specification</td>
</tr>
<tr>
<td>NODE_SPEC2</td>
<td>String</td>
<td>&quot;ALL&quot;</td>
<td>Nodal specification of 2nd group (for &quot;Flux&quot; analyses only)</td>
</tr>
<tr>
<td>INTERFACE_SPEC</td>
<td>String</td>
<td>&quot;EDGE&quot;</td>
<td>Nodal specification of conductive interface group (usually it is the EDGE submodel) (for &quot;Flux&quot; analyses only)</td>
</tr>
<tr>
<td>TIME1</td>
<td>Real</td>
<td>-1</td>
<td>Minimum time range specification (-1 goes automatically to the zero of the simulation)</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>----</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TIME2</td>
<td>Real</td>
<td>-1</td>
<td>Maximum time range specification (-1 goes automatically to the end of the simulation)</td>
</tr>
<tr>
<td>TIME</td>
<td>Real</td>
<td>-1</td>
<td>Time for Tinit output (-1 goes to the end time of the simulation)</td>
</tr>
<tr>
<td>A</td>
<td>Flag</td>
<td>No</td>
<td>To output the Areas</td>
</tr>
<tr>
<td>C</td>
<td>Flag</td>
<td>No</td>
<td>To output the Capacitances</td>
</tr>
<tr>
<td>T</td>
<td>Flag</td>
<td>Yes</td>
<td>To output the Temperatures</td>
</tr>
<tr>
<td>QS</td>
<td>Flag</td>
<td>No</td>
<td>To output the Solar Fluxes</td>
</tr>
<tr>
<td>QA</td>
<td>Flag</td>
<td>No</td>
<td>To output the Albedo Fluxes</td>
</tr>
<tr>
<td>QE</td>
<td>Flag</td>
<td>No</td>
<td>To output the IR planet Fluxes</td>
</tr>
<tr>
<td>QI</td>
<td>Flag</td>
<td>No</td>
<td>To output the Internal Dissipations</td>
</tr>
<tr>
<td>QR</td>
<td>Flag</td>
<td>No</td>
<td>To output the Residual Fluxes</td>
</tr>
<tr>
<td>RADIATIF</td>
<td>Flag</td>
<td>No</td>
<td>To output a detail radiative &quot;Flux&quot; budget</td>
</tr>
<tr>
<td>CONDUCTIF</td>
<td>Flag</td>
<td>No</td>
<td>To output a detail conductive &quot;Flux&quot; budget</td>
</tr>
<tr>
<td>CONVECTIF</td>
<td>Flag</td>
<td>No</td>
<td>To output a detail convective &quot;Flux&quot; budget</td>
</tr>
</tbody>
</table>

**Module paragraph "BPlot"

**Using the sequential batch mode**

Under the keyword "$B PLOT", the commands for executing B-Plot are as described in the B-Plot chapter.
Using a direct call to B-Plot

Type the command
ThermisolXXX -b bplot.dat
where bplot.dat is a parameter file conform to the specifications described in B-Plot chapter.

Example

Here is an example of THERMISOL batch sequence that was used for THERMICA validation procedures on industrial cases:
The goal of this sequence was to compare results between THERMICA v3 and v4. Let's suppose that THERMICA files for THERMISOL has been already computed
THERMICA files:

- **V3 Results:**
  - Example99N.TAN Nodal Description
  - Example99R.TAN Radiative Couplings
  - Example99H.TAN External Fluxes
- **V4 Results:**
  - Example.nod.nwk Nodal Description
  - Example.gr.nwk Radiative Couplings
  - Example.fsa.nwk Solar Fluxes
  - Example.fpa.nwk Planet Fluxes

In order to compare those results, let's compute the steady-state and transient temperatures in both case and plot some curves to have a graphical check.
First of all, we need a generic skeleton using parametric file references that we are going to use for both cases.

**Skeleton.ske**

```plaintext
#----------------------------------#
# Generic Skeleton for Thermisol #
#----------------------------------#
# Parametric files description:
# File 1: Nodal Description
# File 2: Radiative Couplings
# File 3: Planet Fluxes (v4) or External Fluxes (v3)
# File 4: Solar Fluxes (v4) or OFF (v3)
$INFOS
#READ-1
#READ-2
#READ-3
#READ-4
$MODEL #READ-1
$NODES
#READ-1
$CONDUCTORS
```
**$LOCALS**

INTEGER* NBORBIT = 2;

# inclusion of local constants defined in input files
#READ-2
#READ-3
#READ-4

**$ARRAYS**

#READ-2
#READ-3
#READ-4

**$CONTROL**

# simulation time definition
#READ-2
#READ-3
TIMEO = 0;
TIMEND = NBORBIT * PERIOD;

DTIMEI = 10;
OUTINT = PERIOD / 20;

# csv output
CSV_FREQ = 10;

# h5 output definition
1. initial data storage
   H5_RES0 = 'N5,A,C,ALP,EPS,GL,GR,GF';
   # frequency data storage
   H5_FREQ = 4;
   H5_RES1 = 'T,QS,QA,QE,QI,QR';

**$SUBROUTINES**

#READ-2
#READ-3
#READ-4

**$INITIAL**

# initialisation to average time-dependant values
#READ-2
#READ-3
#READ-4

# Storage in h5 file
CALL H5_INIT(' ')
$VTIME

# update of time-dependant radiative couplings and fluxes
#READ-2
#READ-3
#READ-4

$RESULT

# Storage in h5 file
CALL H5_DUMP

$OUTPUTS

CALL PRNDB(' ','L,C,T,QS,QA,QE,QI,QR',CURRENT)

$EXECUTION

# Steady-State case
CALL SOLVIT

# Saving to CSV files initial values
CALL DMPTHM(' ')

# Transient case
CALL SCRANK

$ENDMODEL

Now, let’s define the instruction sequence into a parameter file:

Sequence.txt

# THERMISOL Script for results comparison
#

# V3 Expansion and Execution
$SKELETON

# Input: Skeleton
SKE Skeleton.ske

# Output: DCK – THERMISOL input file
DCK v3.dck

# Parametric reading file
READ-1 Example99N.TAN
READ-2 Example99R.TAN
READ-3 Example99H.TAN
READ-4 OFF
$SOLVER

# Input: DCK
DCK v3.dck

# V4 Expansion and Execution
$SKELETON

# Input: Skeleton
SKE Skeleton.ske

# Output: DCK – THERMISOL input file
DCK v4.dck

# Parametric reading file
READ-1 Example.nod.nwk
READ-2 Example.gr.nwk
READ-3 Example.fsa.nwk
READ-4 Example.fpa.nwk

$SOLVER

# Input: DCK
DCK v4.dck

# Curve Plotting
$BPLOT

# Output: Post-Script file
OUTPUT v3-v4.ps

# Input result files
F1 v3.temp.h5
T1 v3.2.31_3

F2 v4.temp.h5
T2 v4.3.3

# X label Definition (Time)
Xlabel Seconds

# Graph 1:
@Temperatures of nodes 51 & 314
!1,T51,$Baseplate
!2,T51,$Baseplate
!1,T314,$Radiator
!2,T314,$Radiator

# Graph 2:
@Cylindrical structure
!1,T476,$Cyl7-Temperature
!2,T476,$Cyl7-Temperature
!1,QS476,$Cyl7-Solar Flux
!2,QS476,$Cyl7-Solar Flux
The following command will then perform all the tasks requested:

**ThermisolXXX Sequence.txt**

Page left intentionally blank