Master of Science Thesis

On ageing management of nuclear power plants components

Johan Leduc

Supervisors:

Henryk ANGLART

Benoit JOUAN

Division of Nuclear Reactor Technology
Royal Institute of Technology
Stockholm, Sweden, February 2014

TRITA-FYS 2014:50 ISSN 0280-316X0 ISRN KTH/FYS/-14:50-SE
## CONTENTS

1 Presentation and Context 3  
1.1 Presentation of AREVA 3  
1.1.1 AREVA 3  
1.1.2 Mechanical analysis department 4  
1.2 AREVA Fatigue Concept 4  
1.2.1 NPP safety 4  
1.2.2 FAMOS monitoring system 5  
1.2.3 Fatigue assessment 6  
1.3 Fast Fatigue Evaluation 7  
1.3.1 ANSYS® modelisation 7  
1.3.2 The ETV method 7  
1.3.3 Fatigue calculation 9  
1.4 Thesis proceedings 9  
1.4.1 Issues and objectives 9  
1.4.2 Work overview 11  

2 Requirements towards automation 13  
2.1 Structure data 13  
2.1.1 Objective 13  
2.1.2 Numerical data 13  
2.1.3 Ansys® input files and XML 14  
2.2 Perform common tasks 17  
2.2.1 Run calculations 17  
2.2.2 Manipulate data 18  
2.3 Keeping flexibility 18  
2.3.1 Issues with the classic graphic interface 18  
2.3.2 Script engine integration 19  

3 Implementation 21  
3.1 General 21  
3.1.1 Programming tools 21  
3.1.2 Graphic interface and script 21  
3.1.3 The code editor 22  
3.2 The ANSYS® module 22  
3.2.1 Editing the input file 22  
3.2.2 Run a calculation 23  
3.2.3 Process results 23
1.1 AREVA diversification .................................................. 3
1.2 ANSYS results display .................................................. 4
1.3 ANSYS workbench .......................................................... 4
1.4 Thermocouple [3] ......................................................... 5
1.5 Measurement sections [3] ............................................... 5
1.6 Life management improvement [3] ................................... 6
1.7 ANSYS code sample ...................................................... 7
1.8 Decomposition of a function [2] ....................................... 8
1.9 Unit temperature transient [2] ........................................ 9
1.11 Response reconstruction [2] .......................................... 9
1.12 Global FFE Process ....................................................... 10
1.13 FFE data workflow ..................................................... 10

2.1 Example of fatg format .................................................. 14
2.2 Switch from Ansys to XML ............................................. 16
2.3 Structure data for the software ....................................... 17

3.1 The code editor ............................................................ 22
3.2 Script engine model ....................................................... 22
3.3 The Ansys editor .......................................................... 23
3.4 The database manager .................................................. 25
3.5 Interface to edit transient .............................................. 26
3.6 my transient .............................................................. 27
3.7 new transient ............................................................. 27
3.8 Example of a 10 months transient .................................. 28
3.9 Reducing data with a threshold of 0.5 (blue: unreduced curve/value every second) 30
3.10 Response for each second ............................................. 31
3.11 Methods differences ................................................... 35
3.12 F4 and F5 comparison (green: scenario, blue: F4, yellow: F5) 38
3.13 ETV flowchart ........................................................... 39
3.14 Fatigue curve example [5] ............................................ 42
3.15 Fatigue values widget .................................................. 43
3.16 Fatigue parameters widget .......................................... 43
3.17 Fatigue algorithm flowchart ......................................... 44

4.1 Signal 1 ................................................................. 48
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Signal 2</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Basic even shape</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Data reduction algorithm</td>
<td>50</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>Task division</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Error summary</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>ETV algorithms comparison</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Maximum relative error</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Cumulative usage factor differences</td>
<td>51</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1</td>
<td>Run an Ansys calculation</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Extract data</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Get a list of files</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Add a file to a database</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Transient manipulation</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Basic algorithm</td>
<td>29</td>
</tr>
<tr>
<td>3.7</td>
<td>Version with vectors</td>
<td>29</td>
</tr>
<tr>
<td>3.8</td>
<td>Reducing raw data</td>
<td>30</td>
</tr>
<tr>
<td>3.9</td>
<td>Matrix trick</td>
<td>32</td>
</tr>
<tr>
<td>3.10</td>
<td>Algorithm for reduced data</td>
<td>32</td>
</tr>
<tr>
<td>3.11</td>
<td>Dynamic Program</td>
<td>33</td>
</tr>
<tr>
<td>3.12</td>
<td>ETV calculation</td>
<td>39</td>
</tr>
<tr>
<td>3.13</td>
<td>ETV all components</td>
<td>40</td>
</tr>
<tr>
<td>3.14</td>
<td>ETV all components</td>
<td>43</td>
</tr>
<tr>
<td>A.1</td>
<td>ETV all components</td>
<td>57</td>
</tr>
</tbody>
</table>
Ageing management of components is crucial when it comes to nuclear power plants. Indeed, throughout its life a component undergoes temperature and pressure changes causing it to suffer stress. Ultimately, the accumulation of stress over the years can lead to the component failure, with potentially severe consequences in the case of a nuclear power plant. The increasing life expectancy of the plants, both with new generation plants (up to 60 years for the EPR) or the life time extension projects of existing plants, makes today ageing management of components a particularly important aspect of nuclear power plants management.

A nuclear power plant state has to be examined thoroughly either at design phase or regularly during its life time. In practise, the fatigue state of the component and predictions of its evolution are evaluated. Maintenance operations are performed or not depending on the results.

AREVA has developed an approach towards fatigue assessment of nuclear power plants components: the AREVA Fatigue Concept. This approach consists in three levels of fatigue evaluation increasingly accurate but increasingly time consuming and thus costly. The object of this master thesis is the automation of the second level of the fatigue evaluation.

This second level is called Fast Fatigue Evaluation and relies on the Unit Transient Variation method. The process as a whole uses a bunch of different numeric tools and lacks of a global workflow approach. Solving this issue requires a good understanding of the process as a whole and a solid informatics background.
1.1 Presentation of AREVA

1.1.1 AREVA

AREVA is the world leader in the design and construction of nuclear power plants. The company also proposes additional services such as modernisation, maintenance and reparations. It provides its customers with carbon-free solutions for power generation and electricity transmission. With headquarters in Paris, France, AREVA has two main subsidiaries in Germany and the United States. AREVA can also meet the local requirements of its customers with offices in the US., Asia, central Europe and South Africa.

AREVA covers every stage of reactor design and construction, the complete fuel cycle and related other services. Since 2009, the group has also expanded considerably in the renewable energies market.

The group is currently engaged in three major Nuclear Power Plant projects of new generation (EPR) in Finland, France and China.
1.1.2 Mechanical analysis department

The department for mechanical analysis is devoted to structural analysis, component integrity, stability and functionality equations. Among its main capabilities lie all the scope of mechanical analysis: thermo-mechanical analysis, structural dynamics, integrated fatigue concept with different approaches, seismic qualification of electrical structures, . . .

The department is involved in AREVA’s major project like the EPR construction sites as well as installed base projects.

1.2 AREVA Fatigue Concept

1.2.1 NPP safety

Safety is a critical aspect of Nuclear Power Plants (NPPs). Ensuring the safety of a power plant implies the prevention of unforeseen accidents and failures. In this regard, it is necessary to analyse the state of the plant to prevent such events on a regular basis. This is particularly true with the increasing life time of nuclear power plant (up to 60 years for the last generation of reactors), operation extension projects and tightening nuclear safety regulating codes.

During the power plant life, operating conditions are constantly changing as the power generation varies. This changing environment, temperature and pressure of the medium, creates stress loads on power plant components which ultimately can lead to the component’s failure (cracks, leaks, . . .).

Ageing management of nuclear power plant components is thus a key point for safety. The fatigue of a component is an indicator of whether the state of a component is critical or not. The fatigue can be calculated from the temperature and pressure variation that the component goes through and the way it should be calculated is defined in the regulating codes, which may be different from a country to another. Fatigue evaluation of components is performed both in the design and operation phases of a NPP. The design phase is a loop:

- engineers design a plant and its operating modes
• from this design a transient catalogue can be defined (i.e. temperature and pressure variation that the power plant will go through)

• from this transient catalogue a fatigue calculation over the lifetime of the NPP is done which will be used as feedback to optimize the plant design

Fatigue evaluation are also performed during the life of the NPP as real conditions are different from the designed transient catalogue. Global data (medium temperature and pressure) are used to calculate the fatigue state of components.

1.2.2 FAMOS monitoring system

However, using global data can be an inappropriate method to assess the fatigue state of critical components. Indeed, the medium temperature can be quite different from the temperature experienced by the components. For instance, stratification phenomena can occur in horizontal pipes, quick variation of the medium temperature can be experienced differently for different components.

As early as the 70s and 80s, cracks appeared in some pipes due to unexpected local phenomena (stratification) or manufacturing flaws. Consequently, the FAMOS (FAtigue MONitoring System) system was developed by Siemens and installed in german power plants as a request from the german safety authority. The FAMOS system objective was to measure temperatures experienced by critical components and how they differ from what had been expected [3].

The FAMOS system has two components, the global (medium temperature and pressure, valves opening state) and the local monitoring consisting of on-pipe measurement units. The measurement unit consist of thermocouples installed on the outside of the pipe and isolated from the outside medium.

This system was very ahead of his time. The collected data were not only useful to detect critical state of important components but also made it possible to identify and optimise operating modes unfavourable to fatigue, to improve the catalogue transient used at the design phase and for lifetime management and lifetime extension.
A few years ago, the FAMOS system was upgraded with the development of the FAMOSi (FAMOS integrated) system. In comparison to FAMOS, the new system includes a software for an automatic treatment of the information and database management while the data were processed manually beforehand.

1.2.3 Fatigue assessment

The fatigue assessment is something that has to be carried out on a regular basis. It is thus important for the company to be able to perform those assessments in the most effective way as such calculations can be expensive both in time and money.

The fatigue assessment is required by the safety authority and shall be validated by this one when carried out. The fatigue calculation requirements are specified in the regulation codes written by the country’s safety authority, and are thus different for each country.

AREVA has developed a three-stage approach to handle fatigue evaluations: the AREVA fatigue concept [3]. The three steps are:

- the Simplified Fatigue Calculation
- the Fast Fatigue Calculation
- the Detailed Fatigue Calculation

**Simplified Fatigue Calculation** The simplified fatigue calculation is integrated in FAMOSi. This step uses the conservative equation of completely restrained thermal expansion \( \sigma = E \cdot \alpha \cdot \Delta T \) (where \( \sigma \) is the stress, \( E \) the Young's modulus and \( \alpha \) the coefficient of thermal expansion) to determined if a component is worth a more detailed analysis.

**Fast Fatigue Evaluation** This method uses the ETV method (EinheitTransient Verfahren) also known as UTV (Unit Transient Variation) to perform a quick fatigue calculation. The method will be explained in details in the AREVA document [2]. It will be explained later shortly. A usage factor \( U \) is calculated, if \( U \leq U_{\text{admissible}} \) defined by the regulation code, a detailed fatigue calculation does not have to be performed.
Detailed Fatigue Calculation To perform a detailed fatigue calculation, the transient catalogue first has to be updated. A stress response of the component for each transient is calculated with a finite element software (ANSYS®). Finally the results have to be presented to the authority. This process requires more time and money. For this reason, the Fast Fatigue Evaluation has been developed.

1.3 Fast Fatigue Evaluation

In this section, the Fast Fatigue Evaluation process is presented.

1.3.1 ANSYS® modelisation

The FFE process starts by reproducing the studied component in a numerical model. The PEEA department typically uses the ANSYS® software to model the components.

ANSYS® is a commercial software that AREVA uses to perform thermal-mechanical studies of components. The software provides a Graphical User Interface (GUI) either to create the component model or review the results of a calculation. However, AREVA may prefer to work with manual instructions for documentation and re-use of routine purposes. When creating a model, the instructions are written in a separated input file that will be read by ANSYS® to perform the calculation.

![ANSYS code sample]

The creation of a model and a calculation consist almost always of the following tasks:

- creation of the component geometry from plans
- defining materials properties
- creating a mesh for the finite element calculation
- apply a transient to the component

Calculating the response behaviour of a component with this method is accurate but very time-consuming, as data are calculated for every node of the meshing. For long transients, the calculation time can reach several days. For this reason another approach is used in the FFE.

1.3.2 The ETV method

The ETV method (Einheit Transient Variation) also know as UTV (Unit Transient Variation) is a method to calculate a stress or temperature response for FAMOSi measurements quicker
that with a finite element calculation.

**Theory**  
The method is described in details in [2] and is here summarized.

The ETV method fundamentally relies on the superposition principle. It can be applied to linear phenomena like pressure or temperature changes and only to geometry where this superposition principle is empirically verified. In general, simple geometries such as pipes verify this principle.

The idea behind this principle is the linearity of the component response function. For instance, let us call $F_R$ one component response function for a property $p$. We can say that the component response function verifies the superposition principle for the property $p$ if

$$F_R(\alpha p_1(t) + \beta p_2(t)) = \alpha F_R(p_1(t)) + \beta F_R(p_2(t)) \quad (1.1)$$

Let us note $p_{ref}$ a reference transient that represent a single variation of $\Delta P$ over a time period $\Delta t$.

$$p_{ref}(t) = \frac{\Delta P}{\Delta t} \forall t \in [0, \Delta t] \cap \Delta P \forall t > \Delta t$$

An input function $p_{inp}(t)$ can be approximated by decomposing it with the $p_{ref}$ functions:

$$p_{inp}(t) \approx \sum_k \alpha_k \cdot p_{ref}(t + k.\Delta t) \text{ where } \alpha_k = \frac{p_{inp}(k + \Delta t) - p_{inp}(k)}{\Delta P} \quad (1.2)$$

Graphically it gives

![Decomposition of the temperature profile into elementary slopes](image)

**Figure 1.8:** Decomposition of a function [2]

Knowing the response function for the reference transient $F_R(p_{ref}(T))$, the global response can be approximated using the superposition principle.

$$F_R(p_{inp}(t)) \approx \sum_k \alpha_k \cdot F_R(p_{ref}(t + k.\Delta t))$$

The idea of the ETV method is to use ANSYS® to calculate the temperature or stress response of our component over a short reference transient. The ETV algorithm is then applied to decompose the input transient, calculate the unit responses and sum everything to yield the global response.
This method, while less accurate than a finite element calculation, is considerably faster. Indeed, even if modelling the component is still required, only one finite element calculation on a short transient is necessary. This enables huge time increases when a whole transient catalogue has to be evaluated.

1.3.3 Fatigue calculation

When the stress responses have been calculated, the Cumulative Usage Factor (CUF) can be calculated following the code-dependant procedures. AREVA has developed a small software called FAM that automated this task, taking into account the regulating code used to calculate the usage factor.

The global process is represented in Figure 1.12.

1.4 Thesis proceedings

1.4.1 Issues and objectives

As detailed in the FFE description, different softwares are required during the whole FFE process as we can see in figure 1.12. Unfortunately, those softwares do not have the same input/output format and the user constantly needs to modify the data himself.

This means that the workflow of the FFE process is not direct and follows detours (which are often the most tedious tasks where mistakes are likely to be introduced). The FFE process is then fast, if one knows exactly what is wanted (which are the most stressed nodes, . . .) but becomes less attractive when it comes to makes some tests or perform it a great number of times.

On figure 1.13, the data workflow of the process is represented. For the sake of simplicity, only a linear version of the process is presented, while in reality, several Ansys calculations can be done and data have to be combined.
1.4. THESIS PROCEEDINGS

CHAPTER 1. PRESENTATION AND CONTEXT

Figure 1.12: Global FFE Process

Figure 1.13: FFE data workflow
The following points have to be improved in the current FFE process:

- the time required to perform a FFE
- the risk of introducing error while manipulating data manually
- the workflow of the FFE from top to bottom
- teamwork enhancement

1.4.2 Work overview

The first month of the thesis was dedicated to getting a good understanding of the FFE process and all its tools (ANSYS®, scilab routines, FAM, ...).

The following six weeks, numerical bridges with Ansys, the database and Scilab were written in C++. During this phase, a complete software (more than 7000 code lines) has been developed that could handle most of the FFE tasks. The problem with this software was that it did not enable automation as the approach was to perform all the tasks from a graphic interface. A graphic interface to realise the whole process from top to bottom all at once could have been implemented, but it would have been quite complex for the user and more importantly not flexible at all.

The following two weeks, the focus was on the improvement of the ETV algorithm as the previous version could not handle large transients and was not efficient.

Finally, the implementation of a second software started, relying on the experience acquired until then to make automation possible. A script engine has been integrated in the software to give the user the possibility to write script routines while accessing powerful underlying C++ functions. The user can now automate the actions by writing scripts while also being able to do it from graphic interfaces that had been reused from the first software.

Finally, new features and modules were included in the software while a complete user documentation was written to prepare for a potential deployment.

<table>
<thead>
<tr>
<th>Task Division</th>
<th>Readings</th>
</tr>
</thead>
<tbody>
<tr>
<td>September</td>
<td>Readings</td>
</tr>
<tr>
<td>October</td>
<td>Development of numerical tools</td>
</tr>
<tr>
<td>1/2 November</td>
<td></td>
</tr>
<tr>
<td>2/2 November</td>
<td>Optimisation of the ETV algorithm</td>
</tr>
<tr>
<td>December</td>
<td>Development of the software</td>
</tr>
<tr>
<td>1/2 January</td>
<td></td>
</tr>
<tr>
<td>2/2 January</td>
<td>Report writing and optimisation</td>
</tr>
<tr>
<td>February</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Task division

This thesis has brought me to cover large areas of scientific fields in a nuclear energy specific issue. I have deepened and opened my knowledge in very different, yet always interesting, horizons. From fatigue calculations defined by regulating code, innovative methods such as ETV and finite element software understanding all the way to interprocess communication and algorithm optimisation while always keeping in mind that this process will be used in the nuclear field.

The everyday work was a lot of ideas, trials, failures and successes. In this report, a summary of the results are presented following a scientific methodology (hypothesis, experience and conclusion). Firstly, the great principles of the work, the decisions we made and the reasons of those decisions will be explained. In a second part, the way those principles have been implemented will be described. Finally, the validation process of the software will be detailed.
CHAPTER 2

REQUIREMENTS TOWARDS AUTOMATION

2.1 Structure data

2.1.1 Objective

The first step, when one wants to communicate, is to define a set of rules called language or protocol. Likewise, being able to read or write external files or pass instructions to other programs suppose that the data respect a format, i.e. have been structured. *Structure the data* does not mean to rewrite it in another format, but rather to identify or set format rules in the FFE process so that the software knows how to read/write it. This step consists in setting format rules for every file represented in Figure 1.13.

2.1.2 Numerical data

As can be seen in figure 1.13, almost all files handled during the FFE process, except the Ansys input files, are numerical data. Unfortunately, those numerical data do not all have the same format. Some have a header, some only accepts two columns while other accepts more. Some are binary files while others are text files. The Ansys output files (.rst or .rth) are large binary files (typically several gigabytes) because they hold stress or temperature data for all nodes and all step loads. They can only be written or read by Ansys so that we can not use those files directly for ETV, we need to extract the data for the nodes of interest under a readable format, usually text files (.txt).

All the other numerical data are written into readable text files. However, even if this can be enough for a human to read and process those files, it is not for a machine. For example, the text files extracted from Ansys files do not have the same format as the text files used as Scilab routines input. Likewise, the FAM software, used to calculate fatigue from stress, uses a special format (.fatg) as input.
2.1. STRUCTURE DATA CHAPTER 2. REQUIREMENTS TOWARDS AUTOMATION

There are basically three text format that we will use:

- the text format extracted from Ansys binary files (always a matrix with a header)
- the text format (always a matrix without header)
- the fatg format
- the FAMOSi format (one header line, followed by a matrix with [date, value] format)

In order to perform operations on those data (plot, select, store,...) we chose to use a database. We implemented functions to import or export those format into or from a SQLITE database. Using a database seemed to be the right choice to manipulate numerical data and a good choice as a standard format. It basically plays the same role as an Excel like software but it can be easily manipulated by our software.

Using a database also has a lot of advantages. Instead of implementing a bunch of conversion functions with all the possible combinations between the different formats, we use a database as an intermediate step. This also offers more flexibility if one wants to add a new format.

2.1.3 Ansys® input files and XML

Document structure

It is interesting to give the user the possibility to structure his document for two main reasons. First, it makes the document more readable in a teamwork enhancement perspective, and second it facilitates our task to interact with the Ansys input file. For example, modifying only a part of the input or running an Ansys calculation only taking into account part of the code could be done more easily if the input file is structured. First of all, let us have a look at what an ANSYS® input file consists in.

Ansys input files are written by Ansys users. An Ansys input file contains a series of instructions that will be read and executed by Ansys. The structure of such a document can be very flexible depending on the user’s objective. Input files can be quite long in some cases (a couple of hundred lines), which make them difficult to read for anyone who has not written the code.

However, in case of a stress calculation (for example), a pattern in an ANSYS® file can be identified:

- header of the file (Ansys parameters)
- material properties definition
- geometry and meshing
CHAPTER 2. REQUIREMENTS TOWARDS AUTOMATION  

2.1. STRUCTURE DATA

- transient definition
- calculation
- post processing of results

These patterns followed by Ansys users are a great base to structure the document. In most cases, the different sections do not criss-cross each other and can be completely separated. Declaring this kind of structure explicitly would make the lecture of the document much easier for any external reader. This would also enable the machine to locate and modify part of the code.

XML languages

XML (standing for eXtensible Mark-up Languages) are used to structure data in a way that is comprehensible both by humans and machines. The introduction of XML has played a key role in the development of the internet. XML itself is only a set of standard syntax rules. When a document respect those rules, it is said valid, and can be read easily by programs.

A lot of computing languages stem from XML (ie respect the XML syntax and add new rules) especially in the web technologies. This is the case of HTML for example. The fact that so many developers use the XML syntax show how useful it is to structure data as well as its adaptability. Using this syntax will also make our work more flexible and quite easier, as there already exists a bunch of API (Application Programming Interface) implemented to read and modify XML files.

For a short introduction to the XML syntax see the program documentation.

Structure Ansys inputs with XML

Unfortunately, the XML syntax is not compatible with the Ansys syntax, in both ways. First, introducing mark-ups in the Ansys input would raise an error at the Ansys execution, as a mark-up is not a recognised Ansys command. Second, the Ansys code can contain special XML characters (such as <, >, &, ", ') that would potentially raise an error when trying to parse an XML document containing Ansys code.

For these reasons we modified a bit the XML syntax to make it compatible with Ansys code. The Ansys user can then quite simply structure its Ansys document inserting pseudo-mark-ups in its code.

The syntax to be used in the Ansys code differs from the XML syntax at the following points:
- the mark-up are !< ... > instead of < ... >. This will simply place the mark-up in the comments so that it is ignored by Ansys at file execution
- a global mark-up is not necessary, for the sake of simplicity
- special XML characters (such as <, >, &, ", ') are accepted outside mark-ups

The Ansys user can structure its document by writing XML mark-ups in comment. This document is a valid Ansys input but not a valid XML document. We implemented a function that applies minor modifications to the Ansys input to transform in a valid XML document. This function does:
- replace !< ... > by < ... >
- add all the code inside a global mark-up !<file > code </file>
2.1. STRUCTURE DATA

CHAPTER 2. REQUIREMENTS TOWARDS AUTOMATION

- replace special XML characters (<, >, &, " , ') by (&lt; &gt; &amp; &quot; &apos;)

The reverse function has also been implemented so that we can switch back from a XML file to an Ansys input file.

```
<file>
  <header>
    /batch,list
    /CLEAR,START
    /file,scheibmodell !name of the file
    /nerr,3,100000000 !number error, warnings
  </header>
  !<coord_system >
  csys
dsys !coordinate system
  </coord_system >
  !Parameters Definition
  le =50
  space =2
  ri =250
  s =30
  alp =1.5 !heat transfer coefficient
  n_event=300
  p_inn=8.5
  /title,Test Scheibmodell
</file>
```

Figure 2.2: Switch from Ansys to XML

```
<file>
  <header>
    /batch,list
    /CLEAR,START
    /file,scheibmodell !name of the file
    /nerr,3,100000000 !number error, warnings
  </header>
  <coord_system >
  csys
dsys !coordinate system
  </coord_system >
  !Parameters Definition
  le =50
  space =2
  ri =250
  s =30
  alp =1.5 !heat transfer coefficient
  n_event=300
  p_inn=8.5
  /title,Test Scheibmodell
</file>
```

It relatively easy to navigate through a valid XML document, build a model of its structure or modify its content. With this new structure of data, manipulating the file is easier for the user and reduce time and errors by automating conversion procedures.

In a nutshell, structuring the data has introduced a new layer between the original workflow and our new structure as represented in figure 2.3.
2.2 Perform common tasks

2.2.1 Run calculations

Obviously, the most important tasks in the Fast Fatigue Evaluation are the calculations. The automation objective implies that the final software should be able to command the calculations. Those calculations can be divided in three groups:

- the finite element calculations
- the ETV (unit transient method) calculations
- the fatigue calculations

There are two possibilities for our program to perform a calculation, either by relying on the existing calculation tool or reimplementing the calculation algorithms in the software. In the first case, the underlying informatic code deals with interprocess communication. The advantage of this choice is that the complex calculation algorithms are already verified but on the other hand we lose flexibility as the existing calculation tool does not fit into the automation paradigm. In the second case, the advantage is that the code will be perfectly adapted to our needs at the expense of time (both for programming and verifying the algorithm) and calculation options.

The finite element calculations are currently performed with ANSYS® in the AREVA mechanical analysis department in Erlangen. Obviously, reprogramming such a software, even only for simple geometries, is not the purpose of this thesis, and would be way too long. For this reason, our software is able to communicate with ANSYS® rather than perform finite element calculations itself. ANSYS® can be runned in two modes: the graphic mode or the batch mode. It is important to note that communicating with the ANSYS® process is only possible in batch mode. Once an input file has been written, our program can run ANSYS® in batch mode. After the calculation, the software can open ANSYS® in graphic mode to display the results. Convenience functions corresponding to frequently performed tasks have also been implemented: retrieving the result files, check if a calculation finished succesfully,…

The second group of calculations are the ETV calculations. The ETV method has been introduced in the AREVA Fatigue Concept recently. That is, given a measured temperature
and a reference transient, one can calculate quickly the temperature profile at a point in the structure. Currently, the ETV method is included in the AREVA FAMOSi software and a bunch of different versions of the algorithm have also been implemented in Scilab for testing, research or convenience use. Given that the interprocess communication possibilities with the FAMOSi software are not well documented and that implementing the ETV algorithm can be done without major difficulties, an ETV module has been integrated in our software. The methodology is described further on in the ETV module section.

Finally come the fatigue calculations. The AREVA mechanical analysis department in Germany has developed a small Fatigue module (FAM) that takes an ANSYS® fatigue file (.fatg) as input and can perform common fatigue analysis depending on the regulating code (KTA, ASME, ...). The FAM module provides a lot of flexibility on the choice of calculation parameters at the expense of a certain rigidity in the inputs. For instance, the fatigue files taken as inputs are written by ANSYS®. In this file, a certain number of events are defined, where each event is defined by a few load steps and a frequency. This kind of definition is well-suited for the design phase where transient catalogues define events with frequency in a similar way.

However, such a format is not suited to describe a real case scenario. For compatibility reasons and convenience, a function to export data into a .fatg format has been implemented. In this function, the chosen solution is to define only one event, with as many load steps as there are datapoints, and a frequency of one. Eventually, interprocess communication has not been implemented with FAM, for the same reasons that no intercommunication process has been done with FAMOSi: implementing a small fatigue module adapted to FFE was less expensive both in time and flexibility. The methodology is described further on in the fatigue module section.

2.2.2 Manipulate data

Another important part of the FFE process, and probably the one which makes our software worthy in terms of time and error reduction, is data manipulation. Indeed, as highlighted previously, the FFE process is not direct and a lot of intermediate operations have to be done between each calculation.

Such intermediate tasks include: data checking for consistency (through plotting), data linearisation (stamping time) in order to compare two signals, data reduction to keep only significant values, format conversion between different files, ...

Although those tasks are the less complicated theoretically, they are where lies the highest risk of unnoticed error introduction. In this regard, automation provides not only a gain of time but also a gain in reliability.

Our software includes all functions to perform those tasks quickly, simply and in a reliable way.

2.3 Keeping flexibility

2.3.1 Issues with the classic graphic interface

In addition to developing functions to perform tasks, it is also important to think about the frame in which they are to be used by the user, while keeping in mind the automation paradigm.

A first approach would be to create a graphic user interface. This was the first approach used to develop and test the functions while programming them. In the first part of the thesis, focus
has been set on creating numerical bridges with the underlying FFE process. Functions and graphic interfaces have been implemented to enable the user to perform FFE tasks. The next step was to make possible to automate those tasks.

With the first approach, the idea would be to create a graphic interface to tell the software what instructions should be executed and when. This would typically be an interface where the user indicates what is the Ansys input, where to find the Ansys results, what to do to process the results, launch an ETV routine and so on.

However, with it would be very complicated to provide flexibility, as the instructions would stem from the graphic interface which is inflexible itself. This particularly appeared in the attempt to implement an interface to perform ETV with scilab.

Flexibility was introduced by enabling the user to choose data either from text files or databases, which quickly led to an increase in the interface complexity. This flexibility problem became a real issue when wishing to give the possibility to the user to change the underlying scilab routine.

Implementing an automation interface for the whole FFE process appeared complex with such an approach. A certain amount of flexibility could have been incorporated but another angle has been taken to tackle the problem.

2.3.2 Script engine integration

Instead of enabling the user to interact with the implemented functions only through a graphic interface, it has been found better to do it through a script engine too. Scripts are programming languages that stem from the ECMAScript language, defined in standard ECMA-262 [1]. The characteristic of Scripts is that they are interpreted languages in opposition to compiled languages.

For instance, contrary to scripts, C++ is a compiled language. The programmer writes a code that is transformed into a binary file (.exe) by a compiler. This .exe file is directly understandable by the computer. On the opposite, scripts do not require to be transformed to be executed. A program called the script engine or interpreter reads the script file and transforms it into machine instructions at real time.

The idea with this second approach is to integrate a script engine inside our software. The implemented functions to perform FFE tasks are then loaded inside the script engine. The user can then write his own routine and launch an ANSYS® or ETV calculation in one instruction for instance.

Using a script engine has major advantages though it is less user friendly as a graphic interface for a new user.

First of all, it provides all the flexibility necessary to adapt to each user. Moreover, with a graphic interface approach, the user would be blocked if a calculation case that had not been predicted in the design of the program comes up.

Second, this approach enables to reuse previous work. The user can encapsulate his routines inside functions, save them in files and reuse them later. Future developers of the software can also load new functions inside the script engine.

Finally, the script engine provides powerful tools toward automation, particularly through the use of conditional loops. For example, let us say that a user wrote a routine that takes one input file as parameter to perform a whole calculation and save the results. The user now have the possibility to iterate over a list of input files, with a for loop for instance, to perform a bunch of calculations in a row.
3.1 General

3.1.1 Programming tools

Automation of the calculation process has been realised through the programmation of a single software able to handle all the tasks of a FFE. The software has been developed in C++ and make extensive use of the Qt library under LGPL license. The Qt library provides very convenient classes and methods to realise native Graphic User Interfaces, to manage interprocess communications as well as SQL databases management. The Qwt and QCustonPlot library have also been used for plotting purposes.

In total, the program consists in more than 40000 code lines.

As explained in the previous section, a script engine has been integrated into the program in order to make the implemented functions available to users and enable automation. The Qt project includes the Qt Script module, which enables programmers to integrate easily a script engine into C++ programs and make C++ objects or functions available to the script. Our software uses the Qt script module to this purpose.

More information about Qt and the Qt script module can be found at [6].

3.1.2 Graphic interface and script

The functions implemented in our software can be regrouped into several modules:

- the ANSYS module
- the data (storage/manipulation) module
- the ETV module
- the fatigue module

For each of those modules, the functions are accessible by the user either or both through a graphic and through the script. Obviously, the script engine itself is accessible through a graphic interface that is called the Code Editor.
3.1.3 The code editor

The interface to interact with the script engine and submit script routines to it can be seen in Figure 3.1. The widget is composed of a text editor, where instructions can be written, and of a text browser, that displays the script engine output. Script instructions can also be written in separated files (.js) and submitted to the script engine.

The text editor behaves like a console, the submitted commands cannot be edited anymore and the user cannot write anything while the script engine is evaluating. For more readability and error detection, we also implemented a syntax highlighter.

![Figure 3.1: The code editor](image)

3.2 The ANSYS® module

3.2.1 Editing the input file

At the beginning of a FFE process, an ANSYS® calculation usually has to be done to calculate a reference transient that will later be used in the ETV method. During this phase, the ANSYS® input file that will be submitted to ANSYS® is edited. To integrate this task in our software, an Ansys editor has been developed. Compared to a standard text editor, our Ansys editor features usefull options.

First, the list of all ANSYS® commands (more that 1600 in total) has been integrated to provide syntax coloration. This list also enables to have input completion while editing code: if the user types 3 letters the editor proposes a list of all the commands that start with those 3 letters.

The special mark-ups that we defined earlier to structure the ANSYS® input benefit from the syntax coloration. Moreover, the user can have a quick glance on the document structure defined by the mark-ups by parsing the document.
3.2 Run a calculation

Once the input file is ready for calculation, the user can run the code in batch mode or graphic interface mode. The software needs to know the path to the ANSYS® executable file, this path can be changed in the Settings of the software.

This option can be useful in some cases. For instance, when creating the geometry of a component, the user may want to run ANSYS® in graphical mode to check visually the results.

Finally, by setting an attribute status="unvalidated" to a code fragment delimited by mark-ups, the user can choose to ignore this code fragment when submitting the input.

3.2.3 Process results

After a calculation, ANSYS® stores data in large binary files. Those files have rst or rth extensions. They are usually very large because they hold data for all nodes and all load steps. Such files can weight up to several tens of Go. Out of the thousands of nodes, only those with the higher stress load are interesting.

However, it is not possible to read directly data in this database and we must use ANSYS® to access those data. Routines to extract data out of the binary files have been implemented. The user chooses the file (.rst or .rth), selects the interesting nodes and the directory where he wants to extract the results. From those data, the software will create a new ANSYS® input file, with the correct instructions to extract the interesting data into text files, and launch the ANSYS® calculation.
3.2.4 Related script objects

AnsysInput, AnsysOutput and AnsysProgram

Those three classes are used to perform an Ansys calculation from a script routine.

An AnsysInput object represents an ANSYS® input file. This object is used to modify the content of the file (such as apply a transient) and as input of the ANSYS® program.

An AnsysProgram object represents the ANSYS® executable. There is only one object of this class which is called ansys.

An AnsysOutput object represents the result of an ANSYS® calculation. This object is used to get the exit code (success or failure) of the Ansys calculation, to get the errors that could have happened and to retrieve the result files’ paths.

The following script runs a calculation and check if the calculation succeeded.

```
1 var input = new AnsysInput("D:/project/Ansys/model.inp");
2 var output = new AnsysOutput();
3 output = ansys.run(input);
4 if(output.exitcode == 0){
5     // success
6 }
```

Listing 3.1: Run an Ansys calculation

RSTFile and RTHFile

Those classes are used to manipulate the ANSYS® binary result files. They have methods to extract data for specified nodes into text files. The following code extracts temperature data from a .rth file into a text file.

```
1 // supposing that output is an AnsysOutput object
2 var rth = output.getRTH();
3 var dir = new Directory("D:/project/data");
4 var result_file = rth.extractNodes([1,46,78], dir, "results");
5 // result_file is a TextFile object, representing a file that has been created in the specified directory
```

Listing 3.2: Extract data

3.3 The data module

3.3.1 Database management

A graphic interface, seen on Figure 3.4, has been implemented to manipulate databases. This window appears automatically when opening a database from the software. This graphic interface features useful options that enable the user to navigate and see the content of the different tables very easily. It is also possible to rename tables and fields as well as change the format (comma or point separators) of the numbers (France, Germany, US or UK).

Data can be imported in databases from text files. A small widget for plotting data has also been implemented to enable a quick data consistency check. The user can manipulate and check all its data in the same interface whatever the format of the file.
An interface with the FAMOSi software has also been implemented. This interface allows the user to access directly signals stored in a FAMOSi database and extract the data to a local database.

3.3.2 Transients integration

While classic tasks of a FFE from a single software have been considered, new features were also introduced. An interface to design and integrate transients quickly into an Ansys input has been created. From a global perspective, a single component may be used to perform several calculations for different (reference) transients. In practice, the user will modify the part of the Ansys input defining the transient while keeping the rest (geometry, material properties, . . .).

The corresponding graphic interface enables the user to create transients very quickly just by giving time and temperature values as seen on figure 3.5. The user can also give the heat transfer coefficient, in case the transient defines the fluid temperature in the middle of the pipe, and not directly the component temperature.

When the transient has been defined, it can be translated directly into Ansys instructions. A transient can be saved as a binary file with a .trans extension. This way the user can save his transients and reuse them later. Such a transient file can be called from the Ansys editor to generate automatically the corresponding Ansys code.

Transients can be merged, summed or combined using implemented methods. For instance, the user can call a first transient on a first component and a second transient on a second component. The corresponding Ansys code will be generated automatically.
3.3.3 Related script objects

Directory

A Directory object represents a directory of the file system. This object is used to retrieve a list of files in the directory or to specify a location to write files.

The following script retrieves the list of files with the .txt extension in the specified directory.

```java
var dir = new Directory("D:/project/data");
var text_files = dir.getFiles("*.txt");
```

Listing 3.3: Get a list of files

Database and TextFile

Those two classes are used to manipulate files holding data.

A TextFile object represents a file holding text. This object is used to read or write the content of a file from the script as well as input or output object of some methods.

A Database object represents a SQLITE database. This object is used to interact with a database from the script: add, read of modify data.

The following script creates a new database and adds the content of a text file to it.

```java
//creates the database if it does not exists
var db = new Database("D:/project/data/my_database.db");
var file = new TextFile("D:/project/data/results.txt");
```
CHAPTER 3. IMPLEMENTATION

3.4 ETV Module

The ETV method principle have been presented in a previous section. This method is the core of the Fast Fatigue Evaluation process. As a prerequisite of an ETV calculation, we must have two things:

```
db.add(file);
```

Listing 3.4: Add a file to a database

Transient

A **Transient** object represents a .trans file. This class is used to create or manipulate transients from the script.

The following script creates a new transient from an existing one.

```
// creates a Transient object with data read in a file
var my_transient = new Transient("D:/project/Transients/event.trans");

// creates a transient object with (time, temperature) values equal to (0,0) and (100,0)
var pause = new Transient([0,100],[0,0]);

// appends the transient pause at the end of transient
var new_transient = my_transient.append(pause);
```

Listing 3.5: Transient manipulation

Signal and PolySignal

While data are stored in a database, it is convenient to have a class to manipulate it. The class **Signal** represents a list of time values associated with a list of values associated to a property. Several methods can be applied to a **Signal** object to linearise it or to reduce the number of values keeping only significant variations.

A **PolySignal** object holds several **Signal** objects that have the same time base. Those two objects are typically used to plot data or as inputs for the ETV and fatigue functions. A typical **PolySignal** object used as an input calculation would hold 13 signals corresponding to \[ T, S_x, S_y, S_z, S_{xy}, S_{yz}, S_{xz}, MB_x, MB_y, MB_z, MB_{xy}, MB_{yz}, MB_{xz} \].

3.4 ETV Module

The ETV method principle have been presented in a previous section. This method is the core of the Fast Fatigue Evaluation process. As a prerequisite of an ETV calculation, we must have two things:
• data of a long transient of which we want to calculate the response
• input data and response data of a shorter reference transient

The ETV algorithm then decomposes the long transient using the reference transient and calculated coefficients as explained in equation 1.2 page 8. The response of the long transient can be calculated using the previously calculated coefficients if the component verifies the superposition principle. The ETV method is described in details in [2].

A stress or temperature response calculated with the ETV method is considered less accurate than a finite element calculation. Several factors can influence the ETV method results and one of them is the shape of the reference transient. Based on previous work, we will now consider that the reference transient is always a single brutal increase in temperature from 0 to a temperature $T_{ref}$ that occurs in 1 second as represented in Figure 1.9 page 9.

The ETV method is implemented in the AREVA FAMOSi software as well as in a bunch of routines written in Scilab. As explained in section 2.2, a complete integration of the FAMOSi software in our paradigm was not realised as the FAMOSi documentation was short in inter-process communication possibilities. In a first part, the approach to include the ETV method into our software was to link our software to Scilab and enable the user to choose and run his own routines.

It is important to keep in mind that scenario transients (or FAMOSi measurements) can be very long (several months) and that each second of such a transient will be approximated by the reference transient. That is, for long transients, the previous Scilab routines just proved to be incapable of performing a ETV calculation. For this reason, we spend some time to reprogram and optimise the ETV algorithm in order to have a robust routine that can perform real case calculations.

![Figure 3.8: Example of a 10 months transient](image)

The different versions of an ETV calculation algorithm are presented in the following lines. For the sake of simplicity the algorithms are written in pseudo code. Operations and keywords are in blue, vectors or matrices are in green and scalars are in red.

### 3.4.1 Basic algorithm

**Previous version**

Let us note `scenario.txt` the file containing the time and input temperature for the long transient, and `reference.txt` the file containing the time and response temperature for the reference transient.

In this version of the algorithm, input data have to be stamped (ie. data values for every second: which is the basic step on which we decompose our transient using the reference transient). Thus
we first consider that the two text files `scenario.txt` and `reference.txt` have been previously formatted to hold stamped data. The basic algorithm then consists in:

```plaintext
create a vector <input_in_temp> from 'scenario.txt' holding input temperatures
create a vector <ref_out_temp> from 'reference.txt' holding reference output temperatures
create a large vector <results> that will hold the temperature responses for every second

foreach line i in <input_in_temp> do
    calculate the multiplication coefficient alpha
    // alpha = (<input_in_temp>(i+1) - <input_in_temp>(i)) / T_ref
    foreach line j in <ref_out_temp> do
        calculate the right response T_out = alpha * <ref_out_temp>(j)
        add T_out to <results>(i+j)
    end do
end do
return <results>
```

Listing 3.6: Basic algorithm

**Version with vectors**

The main issue with the basic version is the compound of two loops. This leads to a great number of calculations. A smarter way to perform the calculation would be to calculate directly the response vector from the reference response vector. The new algorithm is then:

```plaintext
create <input_in_temp> from 'scenario.txt' holding input temperatures
create <ref_out_temp> from 'reference.txt' holding reference output temperatures
create a large vector <results> that will hold the temperature responses for every second

foreach line i in <input_in_temp> do
    calculate the multiplication coefficient alpha
    // alpha = (<input_in_temp>(i+1) - <input_in_temp>(i)) / T_ref
    calculate the right vector response <T_out> = alpha * <ref_out_temp>
    add <T_out> to <results> at the right slots
end do
return <results>
```

Listing 3.7: Version with vectors

The increase in calculation time is noticeable even for short transients.

### 3.4.2 Skipping input data

The previous version works fine with short transients. However the studied transients are often very long in reality (several months up to a year). An ETV calculation with the previous algorithm on such a transient requires to much time to be an usefull tool in practise.
And one of the characteristic of real transients is that they show long periods of time where the variation of temperature is minimal and not very relevant as we can see on Figure 3.8 page 28. Thus one idea would be to simplify the raw data, given a threshold, keeping only $(time, temperature)$ values where the change in temperature has been significant.

Reducing data

Given a long input matrix transient holding $(time, temperature)$ values and a threshold, the reduction algorithm is as follow:

```plaintext
create <input_in_time> holding input times and <input_in_temp> holding input temperatures
create a scalar threshold
if there are enough data then
    create a vector <reduced_data>
    create a vector <temp> holding the cumulative absolute temperature differences
    //<temp>(i) = |<input_in_temp>(i)| + temp(i-1)
    foreach i in <temp> do
        if <temp>(i) > threshold then
            take the values <input_in_time>(i) and <input_in_temp>(i)
            append them to <reduced_data>
            update <temp> so that the cumulative increase in temperature starts at i
        end if
    end do
end if
return <reduced_data>
```

Listing 3.8: Reducting raw data

Figure 3.9: Reducing data with a threshold of 0.5 (blue: unreduced curve/value every second)

The matrix trick

We have now reduced our input data, we have time intervals of more than one second where the temperature gradient is constant. For example, let us say that the temperature only increases by 1 degree in 20 seconds. Before reducing raw data, we would have recalculated the local
response 20 times with the local gradients. Now we calculate it one time and we use the same
response for the next 19 seconds as seen on figure 3.9.

The next step would be to sum those responses shifting each one one second further as represented in Figure 3.10. There is however a trick to calculate this sum quicker than by iterating
over the vector list and summing each value.

Let us introduce formal notations and consider a case where we reduced data and where the change in temperature is \( \Delta T \approx \text{threshold} \) during a period \([t, t+\Delta t], \Delta t \geq 1\).

We call \( V_{ref} = (ref_1, ref_2, ..., ref_{nrv}) \) the vector containing the referent transient response for each second. This vector’s length is \( nrv \) (for Number of Reference Values). A response vector for one second is equal to \( \Delta T \Delta t \ast T_{ref} V_{ref} \).

Let us note \( A_{\Delta t,(nrv+\Delta t-1)} \) the following matrix, where each line represent the response at the
time \( t+i \). \( A \) is the matrix representation of figure 3.10.

\[
A = \begin{pmatrix}
ref_1 & ref_2 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\
0 & ref_1 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & ref_1 & \cdots & ref_{nrv}
\end{pmatrix}
\]

Figure 3.10: RESPONSE FOR EACH SECOND

Let us call \( S_A \) the vector containing the sum of the columns of \( A \), \( S_A(i) = \sum_{k=1}^{\Delta t} A(i,k) \). As we said we have to sum the responses so that the ETV response for this input interval \( \Delta t \) is equal to \( \frac{\Delta T}{\Delta t} V_{ref} S_A \).

Now, we can calculate \( S_A \) using the cumcum function, implemented in Scilab. This function returns the cumulative sum of a vector. If we have a vector \( V = (v_1, v_2, ..., v_n) \) then \( \text{cumsum}(V) = (v_1, v_1 + v_2, ..., v_1 + v_2 + ... + v_n) \).

Let us consider the matrix \( B_{(nrv+\Delta t-1),(nrv+\Delta t-1)} \).

\[
B = \begin{pmatrix}
ref_1 & ref_2 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\
0 & ref_1 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & ref_1 & \cdots & ref_{nrv} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & \cdots & 0 & 0 & ref_1
\end{pmatrix}
\]

where

\[
C_{(nrv-1),(nrv+\Delta t-1)} = \begin{pmatrix}
0 & \cdots & 0 & ref_1 & \cdots & ref_{nrv-1} \\
\vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & \cdots & 0 & ref_1
\end{pmatrix}
\]

Let us call \( S_B \) and \( S_C \) the vectors containing the sum of the columns on the same model than \( S_A \). We can see that \( S_A = S_B - S_C \) because of equation 3.1.
We notice that \( S_B = \text{cumsum}(B(1,:)) \) (cumsum of the first line of \( S_B \)) and \( S_C = \text{cumsum}(C(1,:)) \) because of the particular configuration of the matrices.

We can then calculate \( S_A \) easily with Scilab creating two vectors \( B(1,:) \) and \( C(1,:) \) and using the \( \text{cumsum} \) function.

We can then calculate \( S_A \) easily with Scilab creating two vectors \( B(1,:) \) and \( C(1,:) \) and using the \( \text{cumsum} \) function.

\[
\begin{align*}
S_B &= \text{cumsum}(B(1,:)) \\
S_C &= \text{cumsum}(C(1,:))
\end{align*}
\]

Listing 3.9: Matrix trick

Adapted algorithm

Let us now consider that scenario.txt is not linearized (i.e., we can have more than one second between the \((\text{time}, \text{temperature})\) values. We have now reduced the data. The improved algorithm is then:

\[
\begin{align*}
\text{create <input_data> from 'scenario.txt' holding input temperatures and reduce it} \\
\text{create <input_in_time> and <input_in_temp> from <input_data>} \\
\text{create <ref_out_temp> from 'reference.txt' holding reference output temperatures for each second} \\
\text{create a large vector <results> that will hold the temperature responses for every second} \\
\text{//<delta_time> = <input_in_time>(2:end) - <input_in_time>(1:end-1)} \\
\text{create <delta_temp> on the same model and <alphas> = <delta_temp> / <delta_time> / \delta_{T_{ref}}} \\
\text{foreach line i of <input_in_time> do} \\
\text{create the response vector for one second <T_out_unit> = <alphas>(i) * <ref_out_temp>} \\
\text{create the response vector <T_out> using the matrix trick} \\
\text{add <T_out> to <results> at the right slots} \\
\text{end do} \\
\text{return <results>}
\end{align*}
\]

Listing 3.10: Algorithm for reduced data

This algorithm enables to reduce considerably the number of iteration of the main loop and thus the calculation time.

### 3.4.3 Splitting vectors

Because the calculation time in Scilab seems to increase exponentially with the size of vectors, we can reduce the calculation time of a function by splitting the main vector into several smaller one, apply the function to those smaller vectors and combine the results to have the global result. This method has been applied successfully to the reduce data function.
3.4.4 Dynamic vectors

The previous algorithm made it possible to reduce considerably the input vector size and the number of iterations of the main loop. However, we still have a huge response vector which holds one data for every second. This problem reaches a limit when dealing with very long transients. Indeed, the calculation time in Scilab seems to increase exponentially with the size of vectors on which calculations are performed. Thus, longer transients imply longer vector to manipulate and exponentially longer calculation time.

More specifically the limiting factor is now the size of the response vector. Indeed, in an ETV calculation we are summing responses overlapping in time, we then have to have a common meshing of time for this summation. In our case the common meshing is the second by second time meshing of the reference transient response vector. Thus the results have to be calculated for every second, however we can find a solution to reduce the size of the vectors we manipulate.

The idea is to handle dynamically allocated vectors. We define a number maxSize which will help us controlling the size of the results vector. The results vector will be initialized with as few values as possible and its size will be incremented at each step. At the end of each step, we will check if the number of values that will not change in further calculations is above maxSize. If so, we reduce those data and store them in a new vector while stripping the results vector from its unnecessary values.

```plaintext
1 create <input_data> from 'scenario.txt' holding input temperatures and reduce it
2 create <input_in_time> and <input_in_temp> from <input_data>
3 create <ref_out_temp> from 'reference.txt' holding reference output temperatures for each second
4 create a vector <temp_out_cal> that will hold the temperature responses for every second
   //But will have a limited size, the first second of <temp_out_cal> is stored in last_time
5 create an empty vector <results> that will hold the temperature responses
6 create <delta_time> holding the time intervals between two values
   //<delta_time> = <input_in_time>(2:end) - <input_in_time>(1:end-1)
7 create <delta_temp> on the same model and <alphas> = <delta_temp> / <delta_time> / <delta_T_ref>
8 foreach line i of <input_in_time> do
9    create the response vector for one second <T_out_unit> = <alphas>(i) * <ref_out_temp>
10   create the response vector <T_out> using the matrix trick
11   add <T_out> to <temp_out_cal> at the right slots
12   calculate the number of finished values finished_values that will not change anymore
13   if finished_values > maxSize then
14      do reduce those finished values
15      append them to <results>
16      take out the finished values from <temp_out_cal>
17      update last_time
18   end if
19 end do
20 return <results>
```
3.4. ETV MODULE CHAPTER 3. IMPLEMENTATION

Listing 3.11: Dynamic Program

This algorithm enables to reduce considerably the calculation time and enables to process considerably longer transients than the basic version. The sacrifice we did in term of accuracy is minimal and can be fixed by the user by setting the threshold when reducing the input data.

3.4.5 ETV with unit variation

In the current way of calculation, we calculate the temperature response for every second no matter what is the temperature variation. Indeed, even if we reduced the data (i.e. more than one second between two input values) we still approximate a local temperature gradient.

Another possibility would be to consider that all the temperature variation occurs during the first second and that no temperature variation occurs after. This interpretation makes it possible to implement an algorithm that reduces dramatically the calculation time. However, we can ask ourselves what are the effect of such an assumption on the results compared with the classic method.

Responses

Let us consider a time interval $\Delta t$ where a change in temperature $\Delta T$ occurs. In the classic case we consider that the change in temperature occurs linearly thoughout the time interval. The temperature response for every second is equal to $\frac{\Delta T}{\Delta t \cdot T_{ref}} * S_A$, the vector holding the sum of the columns of $A_{\Delta t,(nrv+\Delta t-1)}$ where:

$$A = \begin{pmatrix} ref_1 & ref_2 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\ 0 & ref_1 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ 0 & \cdots & 0 & ref_1 & \cdots & ref_{nrv} \end{pmatrix}$$

In a second case we consider that the change in temperature occurs during the first second. In this case, the temperature response is equal to the vector $V_{resp}$ where:

$$V_{resp} = \frac{\Delta T}{T_{ref}} (ref_1, ref_2, \ldots, ref_{nrv}, \ldots, ref_{nrv})$$

We can notice that $V_{resp}$ is equal to $\frac{\Delta T}{\Delta t \cdot T_{ref}} * S_B$ where:

$$B_{\Delta t,(nrv+\Delta t-1)} = \begin{pmatrix} ref_1 & ref_2 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ ref_1 & ref_2 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \end{pmatrix}$$

Cumulative error calculation

The cumulative error is equal to the area between the two curves on Figure 3.11.

$$\sum_{columns} (V_{resp} - S_A) = \frac{\Delta T}{\Delta t \cdot T_{ref}} \sum_{columns} (S_B - S_A) = \frac{\Delta T}{\Delta t \cdot T_{ref}} \sum_{allmembers} (B - A)$$
Figure 3.11: Methods differences

To help us calculate this sum we can use a matrix $A'$ so that $\Sigma(A) = \Sigma(A')$ where

$$A' = \begin{pmatrix} ref_1 & ref_2 & \cdots & ref_{nrv} & \cdots & ref_{nrv} \\ ref_1 & \cdots & ref_{nrv} & \cdots & ref_{nrv} & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ ref_1 & \cdots & ref_{nrv} & 0 & \cdots & 0 \end{pmatrix}$$

and then

$$B - A' = \begin{pmatrix} 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & ref_{nrv} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & ref_{nrv} & ref_{nrv} \end{pmatrix}$$

We then have

$$\Sigma(B - A) = \Sigma(B - A') = \sum_{i=1}^{\Delta t-1} \sum_{j=1}^{i} (ref_{nrv})$$

$$= ref_{nrv} \cdot \sum_{i=1}^{\Delta t-1} i$$

$$= ref_{nrv} \left( \frac{(\Delta t - 1) \Delta t}{2} \right)$$

$$= \frac{ref_{nrv}}{2} \left[ \Delta t^2 - \Delta t \right]$$

Finally we have the total cumulative error $\epsilon_{tot}$ and the average error $\epsilon_{av}$.

$$\epsilon_{tot} = \frac{\Delta T * ref_{nrv}}{2T_{ref}} [\Delta t - 1]$$
\[ \epsilon_{av} = \frac{\epsilon_{tot}}{[\Delta t - 1 + nv]} = \frac{\Delta T}{2T_{ref}} \star \frac{ref_{nv}}{1 + \frac{nv}{\Delta t - 1}} \]

**Conclusion about cumulative error**

We can make several comments:

- we can notice that the cumulative error is null when \( \Delta t = 1 \), which agrees with the model.
- in very long time intervals (\( \Delta t >> nv \)) we have \( \epsilon_{av} \approx \frac{\Delta T \cdot ref_{nv}}{2T_{ref}} \).
- for a stress calculation the reference response tends to zero (i.e. \( ref_{nv} = 0 \)) so the cumulative error is null \( \epsilon_{tot} = 0 \) (i.e. the risk of accumulating a stress load is null).

if the reference data are a temperature response then \( ref_{nv} \approx T_{ref} \) and \( \epsilon_{tot} \approx \frac{\Delta T}{2}[\Delta t - 1] \)

For a stress calculation we cannot accumulate a stress load over time but we have to question the peak stress error. For a temperature calculation we can have a cumulative error, this error is not really relevant as it is more the temperature variation that matters. However, this method can affect the rapidity of a temperature variation, though only on small temperature steps.

**Peak error calculation**

The calculation of the peak error depends on the shape of the reference vector response \( V = (ref_{1}, ref_{2}, ..., ref_{nv}) \).

For a temperature reference vector, we can suppose that \( V \) has increasing values. Thus \( ref_{nv} \) is the maximum. In both cases the maximum value is then the same \( \frac{\Delta T}{T_{ref}} ref_{nv} \). There is no peak value error.

For a stress reference, \( V \) starts at zero, grows until a maximum value \( ref_{max} \) and decreases until zero again. In the unit variation method the peak stress is always \( \frac{\Delta T}{T_{ref}} ref_{max} \). In the cumulative sum the peak value is \( \frac{\Delta T}{T_{ref}} \sum_{i=1}^{nv} (ref_{i}) \) when \( \Delta t > nv \). The error is then

\[ \epsilon_{peak} = \frac{\Delta T}{T_{ref}} \sum_{i=1}^{nv} \frac{ref_{max}}{nv} - \frac{ref_{i}}{\Delta t} \]

In the case of a very long transient the error is maximal and equal to \( \frac{\Delta T}{T_{ref}} ref_{max} \). In practise, \( \frac{\Delta T}{T_{ref}} \approx 1\% \) so the maximum peak value error is equal to 1\% of \( ref_{max} \). The error is moreover always conservative.

**Conclusion**

In conclusion we can sum up what we calculated in a table.

For a temperature calculation we have a non null average error, however this error is not really relevant as we have no peak value error. Thus the temperature variations are not changed and this is what matters in a stress and then fatigue calculation.

For a stress calculation we have a null cumulative error, which means that we are not accumulating a stress load error due to this method. Concerning the peak stress, we have an error but it can be accepted as it is small in amplitude and conservative.
3.4.6 Comparison of algorithms

We will compare our algorithms with the one implemented in the AREVA FAMOSi software. The results will be compared in terms of computation time and accuracy. We will study the case of three different scenarios: one long, one medium and one short.

In terms of accuracy, the two first algorithms without data reduction are strictly equivalent. The third and fourth ones, with data reduction, are slightly less accurate, but this actually smoothen the response curve and shall not play any role in the fatigue assessment. The fifth and sixth algorithms correspond to the same algorithm (unit variation over long periods) but with different implementations. Those last algorithms, though faster do not match perfectly the response curve calculated with FAMOSi.

In terms of accuracy the results obtained with the fourth version are very similar to those obtained with the FAMOSi software. We can also clearly see the difference between the fourth and the fifth version of the ETV algorithm on figure 3.12.
The computational time saved by our hypothesis of the unit variation is not very significant so that it is not worth using this version as it introduces small errors in the response.

In order to save more calculation time and in the perspective of more integration of the ETV calculation in our project, we implemented the fourth version of the algorithm in C++. This version calculates a response very close to the FAMOSi calculated response. Moreover, it is approximately 10 times faster to compute than FAMOSi. This is not negligible as calculating the 13 ETV responses (12 stress responses and the temperature) can take more than 10 minutes with FAMOSi. The flowchart of the C++ algorithm is represented on figure 3.13.
3.4.7 Related script functions

Our software features no graphic interface to interact with the ETV related functions. Two script functions `etv_temp` and `etv_stress` can be used to perform ETV calculations.

They are typically used as follows:

```java
//Process ETV
var db = new Database("results.db");
var ref = new Signal();
var sce = new Signal();
var res = new Signal();
ref.setTime(db.vector("reference_famssi", 1));
ref.setValue(db.vector("reference_famssi", 4));
sce.setTime(db.vector("FAMSi_ME6T1", 2));
sce.setValue(db.vector("FAMSi_ME6T1", 3));
//res = etv_temp(ref, sce);
```
3.5 Fatigue module

Fatigue calculation usually refers to the calculation of the cumulative usage factor (CUF). Each regulating code defines the way the usage factor has to be calculated. According to the ASME and KTA code, the usage factor is to be calculated as follows:

- count the stress cycles for each main stress component
- calculate the alternative stress for each stress cycle
- calculate the partial usage factor associated with this cycle
• sum the partial usage factors to get the cumulative usage factor for this main stress component

3.5.1 Cycle counting

There are several methods to count stress cycles. The extreme counting method pairs the maximum and minimum stress peaks. This method introduces a lot of conservatism and is not very realistic as it can pair together stress values very spaced in time.

The rainflow counting method is a recognised method to identify fatigue relevant stress loads. The AREVA FAM module uses the rainflow counting method by means of the Hysteresis Counting Method (HCM) as explained in [4]. Our software includes a C++ version of the algorithm used in FAM.

3.5.2 Alternating stress calculation

The equivalent stress for a stress cycle can be calculated either with the Von Mises or Tresca equation depending on the regulating code.

The Von Mises equivalent stress is

\[ \sigma = \sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2 - \sigma_x \sigma_y - \sigma_y \sigma_z - \sigma_z \sigma_x + 3(\sigma_{xy}^2 + \sigma_{xz}^2 + \sigma_{yz}^2)} \]

The Tresca stress equivalent is equal to

\[ \sigma = \max(\sigma_I - \sigma_{II}, |\sigma_{II} - \sigma_{III}|, |\sigma_I - \sigma_{III}|) \]

where \( \sigma_I \), \( \sigma_{II} \) and \( \sigma_{III} \) are the main stress components.

Calculating the main stress components is done by diagonalising the following matrix.

\[
\begin{pmatrix}
\sigma_x & \sigma_{xy} & \sigma_{xz} \\
\sigma_{xy} & \sigma_y & \sigma_{yz} \\
\sigma_{xz} & \sigma_{yz} & \sigma_z
\end{pmatrix}
\]

This matrix is diagonalisable because it is symmetric. The main stress component are then the characteristic values of this matrix. Our algorithm finds the characteristic values by developing the third degree characteristic polynomial of this matrix and finding its roots using the Cardan method.

Correcting factors should also be calculated, the first one is the Young modulus variation with the temperature \( \frac{E}{E(T)} \) as the material gain elasticity with temperature increase and the second one is a \( K_e \) coefficient to model the plastic-elastic behavior of the material.

The corresponding alternating stress is calculated with the formula \( S_{alt} = \frac{E}{E(T)} \frac{\sigma K_e}{2} \) [4].

3.5.3 Usage factor

The allowed number of cycles is obtained from the previously calculated alternating stress the fatigue curve given by the code. The inverse of this allowed number of cycles yields the partial usage factor for this cycle. The cumulative usage factor is the sum of all the partial usage factors for all the counted cycles.
3.5.4 User interface

A fatigue calculation can be done through a graphic interface. This interface consists in three widgets. In the first widget the user can enter the stress values. In the second widget, the user shall input miscellaneous parameters such as the fatigue curve definition, the Young’s modulus temperature variation, material parameters and so on. In the last widget, the user can launch the calculation and view the results. The results can also be exported to a MS Excel file.

3.5.5 Related script function

The script function fatigue can be used to perform a fatigue calculation from the script. This function takes a polysignal with thirteen components, a template file and a path to a MS Excel result file as parameters. The template file format is the same as in FAMOSi.
The fatigue function is typically used as follows:

```javascript
var db = new Database("D:/Project/Database/fatigue.db");
var tb_name = "scenario";
var poly = new PolySignal("values");
var sig = new Signal();
sig.setTime(db.vector(tb_name, 1));
for(var i=0; i<fields.length; i++){
    sig.setName(fields[i]);
    sig.setValue(db.vector(tb_name, i+2));
poly.addSignal(sig);
}
var template = new TextFile("D:/Project/Fatigue/Loc_3_Plug_flow_M6T1.txt"); // FAMOSi file holding parameters
var result = "D:/Project/Fatigue/results_script.xls";
fatigue(poly, template, result);
```

Listing 3.14: ETV all components

The above code retrieves data from the database and stores them in a polysignal. A text file representing the FAMOSi template is also created as well as a string holding the path to a MS Excel path.

At the end of execution, a MS Excel file has been created with the fatigue calculation details and results.

The flowchart of the fatigue function is represented on figure 3.17.
3.6 Functionalities summary

Some of the major functionalities of the program are listed below:

- ANSYS® editor (syntax coloration, auto-completion)
- Structuration of an ansys input, selection/unselection of code parts
- Extracting data from RST and RTH files
- Automatic generation of ANSYS® code for transients
- Easy SQLITE database management
- Drag and drop data manipulation for a direct workflow
• Fast ETV algorithms
• Integrated fatigue module
4.1 ANSYS module

4.1.1 Extract data from results files

The functions to extract nodes data stored in ANSYS® binary files have also been tested. A calculation has been performed and results extracted both manually and with using the implemented functions. The data match despite a negligible difference that can be attributed to rounding errors.

4.1.2 XML features

A second test consists in the verification of the XML related features. An ANSYS® input file has been structured using XML mark-ups. The different parts have been saved in separated files using the implemented functions. A similar input has then been reconstructed using those parts. A calculation has been runned and results compared with the initial calculation.

The results are exactly the same thus proving the exactitude of the input file manipulations through the XML implemented functions.

4.2 Data module

To validate the data module, the exactitude of the operations performed to manipulate the data has to be verified. This verification will be done systematically with two different sets of data. The data yielded by our software will be compared with the same calculation done with MS Excel.

The two sets of data have the following shape.
4.2.1 Importing the data in the database

The function to import data into our software from text files has been verified. The two data sets have been copied and pasted from MS Excel into text files and the text files have been imported into a database created with our software. The imported data have then been copied back to MS Excel and compared with the original data.

For both data sets the matrix size remains unchanged. For the first data set, small differences are observed and no differences for the second. The differences can be explained by small rounding mistakes as the first data set values have a lot of decimals. The second data set shows no differences at all.

4.2.2 Linearisation

The Signal linearisation function has also been verified. Linearise is setting a constant time step between two values, missing values are linearly interpolated. The two data sets have been linearised with a MS Excel macro. The same operation has been done through our software.

For both data sets the matrix size remains unchanged. For both data sets, negligible differences are observed.

4.2.3 Operations

The various operations on Signal objects have also been verified. The results for the operations add, factor, power and times (multiplication of two Signal objects) show no significant difference between the data calculated with our software and the data calculated with MS Excel.

4.2.4 PolySignal

The consistency of the PolySignal object has also been verified. A PolySignal object encapsulates several Signal objects and ensures that all the signals have the same time values. When adding a Signal to a PolySignal, the time vectors of the PolySignal and of the new Signal are merged, missing values are linearly interpolated.

These operations have been verified with a PolySignal holding the two data sets. The PolySignal has been added to the database and then recopied in Excel. No significant error has been introduced by the process.
4.2.5 Transient

The transient integration operations have been verified. A basic transient representing a random event has been creating using the graphical interface. Its shape is as seen on figure 4.3.

The transient has been saved then imported in the database without any error. Basic operations have then been performed on the transient: append, add and factor. No error has been introduced.

Finally the conversion function to translate a transient into ANSYS® code has been tested and both the generated code and the results have been checked successfully.

4.3 ETV module

4.3.1 Algorithm verification

A calculation has been performed both through our implemented functions and through FAMOSi. The two data sets of the previous calculation have been reused. The results show no significant difference between our results and the data calculated with FAMOSi. However, small gaps can be attributed to post processing data reduction method differences. Indeed, after the ETV response is calculated, the data are reduced in order to keep only significant values rather than entries for every seconds.

The way the data is reduced in our algorithm is showed on figure 4.4.
This way of reducing data can cause small deviations with the original data set. However, those small gaps are not very significant in a perspective of a further fatigue analysis as they do not alter the stress cycles counting algorithm (peak values do not change in practice).

4.4 Fatigue module

4.4.1 Implemented functions

Several functions have been implemented and need to be verified. Particularly, the rainflow cycle counting algorithm, the Von Mises and Tresca calculation and the fatigue curve logarithmic interpolation functions need to be thoroughly checked.

A complete fatigue calculation has been performed. The input data were taken from FAMOSi, the calculation has been performed both with FAMOSi and our software (for both the Von Mises and Tresca cases) and results have been compared.

Only negligible differences have been observed, they are summarised in the table below. One can also note that the counting algorithms do not store the cycles in the same order, even if the identified cycles are exactly the same.
<table>
<thead>
<tr>
<th></th>
<th>$S_x$</th>
<th>$S_y$</th>
<th>$S_z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tresca</td>
<td>0</td>
<td>$10^{-18}$</td>
<td>$10^{-10}$</td>
</tr>
<tr>
<td>Von Mises</td>
<td>$10^{-19}$</td>
<td>0</td>
<td>$10^{-19}$</td>
</tr>
</tbody>
</table>

Table 4.2: Cumulative usage factor differences
The Fast Fatigue Evaluation (FFE) is a medium accuracy process for fatigue evaluation. It relies on the Unit Transient Variation method to calculate stress loads in a component rather than the finite element method, which enables to spare a lot of calculation time. If the so calculated fatigue coefficient is small enough no further fatigue calculation is required. Otherwise, a detailed fatigue calculation has to be carried out.

The FFE process does not rely on a single platform but on several softwares and algorithms. The process consists in three main steps: the finite element calculation on a reference transient, the Unit Variation Transient calculation and the fatigue calculation. From each of those steps stem substeps, mainly lower tasks of data conversions or informations processing. The user has to convert data manually which makes the whole process slower and prone to error introduction.

A good understanding of the process as a whole has been necessary to tackle this issue. After having acquired a good insight, we started modelling the process and try to shape a global structure and great lines towards automation. It appeared that structuring and manipulating data, perform common tasks (either with other programs or by reimplementation) and setting a correct user interaction framework were necessary steps to reach automation.

A complete software has been developed in C++ to enforce those principles to the FFE process. Its features can be divided in four modules: the ANSYS® module, the data module, the ETV module and the fatigue module.

The implemented functions and routines can be used by the user through the graphical user interface as well as through a script engine. The script engine enables the user to combine the implemented functions at will, so that a whole FFE calculation can be automated from top to bottom while the program remains flexible to adapt to other kind of calculations.

Finally, we sometimes priviledged interprocess communication but we also reimplemented some FFE (ETV, fatigue functions) and new (transient integration, data manipulation) features. Those functions have been verified to be correct and code conforming.

A complete documentation on the software has also been written in the course of this thesis.


# include "etv_algorithm.h"

/****************************
* Algorithm returning the absolute value of a double
* *
* ****************************

double abs_val(double par)
{
    if(par>=0)
        return par;
    return (-par);
}

/****************************
* Matrix trick algorithm
* *
* cumsum: the reference vector cumsum
* last_ref: the last value of the reference vector
* coef: the multiplication coefficient (Delta_temp)/(Delta_time)/temp_ref
* delta_t: Delta_time
* *
* return value: a vector with nrv+Delta_time-i values holding the local response
* ****************************

std::vector<double> matrix_trick(std::vector<double> &cumsum, double last_ref, double coef, int delta_t)
{
    std::vector<double> response(cumsum);
    double alpha = cumsum.back();
    for(int i=1; i<delta_t; i++){
        response.push_back(alpha+i*last_ref);
    }
    int size = response.size();
    for(int i=0; i<size; i++){
        if(i>=delta_t){
            response[i] = coef*(response[i] - cumsum[i-delta_t]);
        }else{
            response[i] = coef*response[i];
        }
    }
    return response;
}

/****************************
* ETV direct algorithm for temperature
* ****************************
APPENDIX A. ETV C++ ALGORITHM

* ref_temp: the vector holding reference temperatures for each second
* sce_time, sce_temp: lists holding time and temperature values
* for the long transient
* return value: a list holding time and temperature results
  * even indices are times
  * odd indices are temperatures
  *
  *******************************************************************************/
std::list<double> etv_temp_alg(std::vector<double>& ref_temp,
std::list<int>& sce_time,
std::list<double>& sce_temp)
{
  std::list<double> results;

  // Parameters definition
  // *******************************************************************************/
  double temp_ref = ref_temp.back() / 100; // temperature of the reference transient Â°C
  int buffer_size = 10000;
  double sensitivity = 0.25; // a negative sensitivity will consider all values

  // Check conditions
  // *******************************************************************************/
  if (sce_time.size() != sce_temp.size())
    return results;
  if (sce_time.size() < 2)
    return results;

  // Initialisation of miscellaneous variables
  // *******************************************************************************/
  std::vector<double> cb(buffer_size, sce_temp.front());
  int nrv = ref_temp.size(); // Number of Reference Values
  std::vector<double> cumsum(nrv);
  for (int i = 0; i < nrv; i++)
  {
    if (i == 0)
      cumsum[i] = ref_temp[i];
    else
      cumsum[i] = ref_temp[i] + cumsum[i - 1];
  }
  std::list<int>::iterator it_time = sce_time.begin();
  std::list<double>::iterator it_temp = sce_temp.begin();

  double variation_inp = sensitivity + 1, variation_out = sensitivity + 1;
  int last_time = *it_time, current_time = *it_time;
  double last_temp = *it_temp, current_temp = *it_temp, coef;
  it_time++;
  it_temp++;

  double last_ref = ref_temp.back(); // Last temperature value
  std::vector<double> response;
  double last_value = *it_temp, previous_temp = *it_temp;

  // Iterating over input values
  // *******************************************************************************/
  while (it_time != sce_time.end()){
    // Checking variation
    current_temp = *it_temp;
    variation_inp += abs_val(current_temp - previous_temp);

    // If the variation has been significant
    if (variation_inp >= sensitivity){
      current_time = *it_time;

      // Check that we have been forward in time (to avoid input inconsistencies)
      if ((current_time - last_time) > 0){
        // Reset variation
      }

      previous_temp = current_temp;
      variation_inp = sensitivity;
    }
    else {
      // Reset variation
    }
    previous_time = current_time;
    variation_out = sensitivity;
    variation_inp = sensitivity;
  }
  return results;
}
APPENDIX A. ETV C++ ALGORITHM

```cpp
variation_inp = 0;

// Creating a plateau for last values
for (int i = last_time + nrv - 2; i < (current_time + nrv); i++) {
    cb[i % buffer_size] = last_value;
}

// Calculating response
coeff = (current_temp - last_temp) / (current_time - last_time) / temp_ref;
response = matrix_trick(cumsum, last_ref, coeff, current_time - last_time);

// Adding local response to global response
for (int i = last_time; i < (current_time + nrv - 1); i++) {
    cb[i % buffer_size] += response[i - last_time];
}

// Taking last ultimate value for the next plateau
last_value = cb[(current_time + nrv - 2) % buffer_size];

// Saving calculated results between last_time and current_time
for (int i = last_time; i < current_time; i++) {
    variation_out += abs_val(cb[i % buffer_size] - cb[(i - 1) % buffer_size]);
    if (variation_out >= sensitivity) {
        variation_out = 0; // reset variation_out
        results.push_back((double)(i));
        results.push_back(cb[i % buffer_size]);
    }
}

// Updating last values
last_time = current_time;
last_temp = current_temp;
}

// ETV direct algorithm for stress

ETV direct algorithm for stress

ref_stress: the vector holding reference stress for each second
sce_time, sce_temp: lists holding time and temperature values
for the long transient
return value: a list holding time and stress results
even indices are times
odd indices are stress

ETV stress algo(
    std::vector<double>& ref_stress,
    std::list<int>& sce_time,
    std::list<double>& sce_temp)
{
    std::list<double> results;

    // Parameters definition
    double temp_ref = 100; // temperature of the reference transient °C
    int buffer_size = 10000;
    double sensitivity = 0.25; // a negative sensitivity will consider all values

    // Check conditions
    if (sce_time.size() == sce_temp.size())
        return results;
    if (sce_time.size() < 2)
        return results;
```
APPENDIX A. ETV C++ ALGORITHM

//Initialisation of miscellaneous variables

//**************************************************************************************************
std::vector<double> cb(buffer_size, 0); //initialises at 0, no stress initially

int nrv = ref_stress.size(); //Number of Reference Values

//Cumsum vector, calculated only once
std::vector<double> cumsum(nrv);

for (int i = 0; i < nrn; i++){
    if (i == 0)
        cumsum[i] = ref_stress[i];
    else
        cumsum[i] = ref_stress[i] + cumsum[i - 1];
}

std::list<int>::iterator it_time = sce_time.begin();
std::list<double>::iterator it_temp = sce_temp.begin();

double variation_inp = sensitivity+1, variation_out = sensitivity+1;

int last_time = *it_time, current_time = *it_time;

double last_temp = *it_temp, current_temp = *it_temp, coef;

it_time++;

it_temp++;

double last_ref = ref_stress.back(); //Last temperature value

std::vector<double> response;

double last_value = 0, previous_temp = *it_temp;

//Iterating over input values

while (it_time != sce_time.end()){

//Checking variation
    current_temp = *it_temp;
    variation_inp += abs_val(current_temp - previous_temp);

//If the variation has been significant
    if (variation_inp >= sensitivity)
        current_time = *it_time;

//Check that we have been forward in time (to avoid input inconsistencies)
    if ((current_time - last_time) > 0){

//Reset variation
        variation_inp = 0;

//Creating a plateau for last values
        for (int i = last_time + nrn - 2; i <= (current_time + nrn - 1); i++){
            cb[i % buffer_size] = last_value;
        }

//Calculating response
    coef = (current_temp - last_temp) / (current_time - last_time) / temp_ref;
    response = matrix_trick(cumsum, last_ref, coef, current_time - last_time);

//Adding local response to global response
        for (int i = last_time; i <= (current_time + nrn - 1); i++){
            cb[i % buffer_size] += response[i - last_time];
        }

//Taking last ultimate value for the next plateau
    last_value = cb[(current_time + nrn - 2) % buffer_size];

//Saving calculated results between last_time and current_time
        for (int i = last_time; i < current_time; i++){
            variation_out += abs_val(cb[i % buffer_size] - cb[(i - 1) % buffer_size]);

            if (variation_out >= sensitivity){
                variation_out = 0; //reset variation_out
                results.push_back((double)(i));
                results.push_back(cb[i % buffer_size]);
            }
        }
    }
}
APPENDIX A. ETV C++ ALGORITHM

```cpp
// Updating last values
last_time = current_time;
last_temp = current_temp;
}
}
prevus_temp = current_temp;
it_time++;
it_temp++;
}
return results;
}

/**
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Function called from QtScript: Uses Qt
 *
 * This function takes two arguments: reference and scenario
 * Those argument can be either a path to a file holding data or a Signal
 * return value: the response signal
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

QScriptValue etv_script(QScriptContext* context, QScriptEngine* engine)
{
    // Checking that there are 3 arguments: reference, scenario, results
    if(context->argumentCount() != 2)
        context->throwError("Wrong number of arguments for ETV." +QString::number(context->argumentCount()));

    // Script values
    QScriptValue ref_val = context->argument(0), sce_val = context->argument(1);

    // Reads the reference
    std::vector<double> ref_temp;
    if(ref_val.property("class").toString() == "Signal"){
        QObject *object = ref_val.toQObject();
        Signal* sig = static_cast<Signal*>(object);
        sig = sig->linearise();
        ref_temp = sig->Value().toVector().toListVector();
    } else{
        QStringList rows = ref_text.split("\n");
        QStringList fields;
        bool ok;
        double val;
        for(int i = 0; i < rows.size(); i++){
            fields = rows.at(i).split("\t");
            if(fields.size() > 1){
                val = fields.last().toDouble(&ok);
                if(ok)
                    ref_temp.push_back(val);
            }
        }

    // Reads the scenario
    std::list<int> sce_time;
    std::list<double> sce_temp;
    if(sce_val.property("class").toString() == "Signal"){
        QObject *object = sce_val.toQObject();
        Signal* sig = static_cast<Signal*>(object);
        sig = sig->linearise();
        QList<double> time_val = sig->Time();
        QList<double>::iterator it_time_val = time_val.begin();
```
while (it_time_val != time_val.end()) {
    sce_time.push_back((int)(*it_time_val));
    it_time_val++;
}

sce_temp = sig->Value().toString();
if (sce_file.open(QIODevice::ReadOnly)) {
    sce_text = String(sce_file.readAll());
    sce_file.close();
}

QStringList rows;
QStringList fields;
for (int i = 0; i < rows.size(); i++) {
    fields = rows.at(i).split("\t");
    if (fields.size() == 2) {
        sce_time.push_back(fields.first().toInt());
        sce_temp.push_back(fields.last().toDouble());
    }
}

QStringList time, val;
while (it != results.end()) {
    time << *it;
    it++;
    val << *it;
    it++;
}

Signal *response_sig = new Signal(engine);
response_sig->set_Time(time);
response_sig->set_Value(val);

QString Value return_val = engine->newQObject(const_cast<Signal*>(response_sig),
    QScriptEngine::QObjectOwnership);
return return_val;
1 Overview
   1.1 What is this? ................................................................. 1
   1.2 Documentation ............................................................ 1
   1.3 Getting Started ........................................................... 1

2 Introduction
   2.1 Main Window interface ................................................... 3
      2.1.1 Description .......................................................... 3
      2.1.2 Menu ................................................................. 3
   2.2 Ansys Input ............................................................... 3
      2.2.1 Description .......................................................... 3
      2.2.2 Structuring the data .............................................. 4
      2.2.3 Short introduction to XML languages ......................... 4
      2.2.4 Use in an Ansys input .......................................... 5
      2.2.5 Possible actions .................................................. 6
   2.3 Transient ................................................................. 6
      2.3.1 Description .......................................................... 6
      2.3.2 The transient mark-up ......................................... 7
   2.4 Databases ............................................................... 8
      2.4.1 Description .......................................................... 9
   2.5 Code Editor .............................................................. 9
      2.5.1 A short introduction to Script codes ....................... 9
      2.5.2 Custom types ..................................................... 10

3 Tutorials
   3.1 Script Tutorials ........................................................ 11
      3.1.1 What is Script? ................................................... 11
      3.1.2 Why using Script? ............................................... 11
      3.1.3 Where to write Script? ........................................ 12
      3.1.4 The basis of Script ....................................... 12
         3.1.4.1 Write a script ............................................. 12
         3.1.4.2 Script Syntax ............................................. 12
         3.1.4.3 Variables .................................................. 13
         3.1.4.4 Basic Structures ......................................... 14
         3.1.4.5 Functions ................................................ 16
         3.1.4.6 Objects .................................................... 16
   3.2 Ansys Tutorials ........................................................ 18
      3.2.1 Open the Ansys editor ........................................ 18
      3.2.2 Create an AnsysInput object .................................. 18
      3.2.3 Structure an Ansys Input ..................................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4 Modify an Ansys Input from the script</td>
<td>18</td>
</tr>
<tr>
<td>3.2.4.1 Set a mark-up text</td>
<td>18</td>
</tr>
<tr>
<td>3.2.4.2 Apply a Transient</td>
<td>19</td>
</tr>
<tr>
<td>3.2.5 Run Ansys from the Script</td>
<td>19</td>
</tr>
<tr>
<td>3.2.6 Extract data from a RST file</td>
<td>20</td>
</tr>
<tr>
<td>3.2.7 Extract data from a RTH file</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Transients Tutorials</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1 Create a Transient</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1.1 Create a transient file</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1.2 Create a Transient Object</td>
<td>21</td>
</tr>
<tr>
<td>3.3.2 Combine Transients</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2.1 Append Transients</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2.2 Add Transients</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2.3 Scale a transient</td>
<td>23</td>
</tr>
<tr>
<td>3.3.3 Write Transients into text files</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Database Tutorials</td>
<td>23</td>
</tr>
<tr>
<td>3.4.1 Create a Database</td>
<td>23</td>
</tr>
<tr>
<td>3.4.2 Add Data to the database</td>
<td>24</td>
</tr>
<tr>
<td>3.4.3 Extract data from the database</td>
<td>25</td>
</tr>
<tr>
<td>3.5 Scilab Tutorials</td>
<td>26</td>
</tr>
<tr>
<td>3.5.1 Start Scilab</td>
<td>26</td>
</tr>
<tr>
<td>3.5.2 Execute a Scilab instruction</td>
<td>26</td>
</tr>
<tr>
<td>3.5.3 Execute the content of a .sce file</td>
<td>26</td>
</tr>
<tr>
<td>3.5.4 Stop scilab</td>
<td>27</td>
</tr>
<tr>
<td>3.6 Other Tutorials</td>
<td>27</td>
</tr>
<tr>
<td>4 Examples</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Step by step Fatigue Calculation</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1 Presentation</td>
<td>29</td>
</tr>
<tr>
<td>4.1.2 Parameter definition</td>
<td>29</td>
</tr>
<tr>
<td>4.1.3 Ansys Calculation</td>
<td>30</td>
</tr>
<tr>
<td>4.1.4 Extract Data</td>
<td>31</td>
</tr>
<tr>
<td>4.1.5 ETV calculation</td>
<td>31</td>
</tr>
<tr>
<td>4.1.6 Process data</td>
<td>32</td>
</tr>
<tr>
<td>5 Class Index</td>
<td>33</td>
</tr>
<tr>
<td>5.1 Class List</td>
<td>33</td>
</tr>
<tr>
<td>6 Class Documentation</td>
<td>35</td>
</tr>
<tr>
<td>6.1 AnsysInput Class Reference</td>
<td>35</td>
</tr>
<tr>
<td>6.1.1 Detailed Description</td>
<td>35</td>
</tr>
<tr>
<td>6.1.2 Member Function Documentation</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.1 applyTransient</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.2 childElements</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.3 getElementText</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.4 open</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.5 setElementAttribute</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.6 setElementText</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2.7 setTransientComp</td>
<td>37</td>
</tr>
<tr>
<td>6.1.2.8 setTransientURL</td>
<td>37</td>
</tr>
<tr>
<td>6.1.3 Property Documentation</td>
<td>37</td>
</tr>
<tr>
<td>6.1.3.1 resultFiles</td>
<td>37</td>
</tr>
<tr>
<td>6.2 AnsysOutput Class Reference</td>
<td>37</td>
</tr>
<tr>
<td>6.2.1 Detailed Description</td>
<td>38</td>
</tr>
<tr>
<td>6.2.2 Member Function Documentation</td>
<td>38</td>
</tr>
<tr>
<td>6.2.2.1 getRST</td>
<td>38</td>
</tr>
<tr>
<td>6.2.2.2 getRTH</td>
<td>38</td>
</tr>
</tbody>
</table>
1.1 What is this?

The AFC Editor software aims to provide a tool for automated fatigue calculation. It gives tools to edit Ansys Inputs, process results, to store and plot data, run ETV routines and others. You can get a glance at those tools in the Introduction section. The core of the software is the script editor. Using the script makes it possible to go from a tool to another automatically while giving you the possibility to create your own routines.

1.2 Documentation

This documentation aims to introduce the reader to the software developed for automated fatigue calculations. In the Introduction section you will find a description of the software graphic interface. In the Examples section you will find detailed and commented examples, that will help you to understand the automation logic. In the Tutorials section you will find short tutorials on how to perform some tasks.

1.3 Getting Started

If you are just getting started, we suggest that you start by reading the Script tutorial section to get a glance at the overall objective of the software.

Once you are familiar with the concept you can check the Step by step Fatigue Calculation to understand the automation logic.

At any moment, you can check the Tutorials if you ask yourself how to do something. You can also check the classes description for a more detailed description.
2.1 Main Window interface

2.1.1 Description

When you open the program a graphical user interface (GUI) window appears.

It is divided in several parts:

- The system file tree provides a quick overview on the system files. The user can open a file by double-clicking on it. A drag-drop action on one item will result in copying the file’s absolute path at the drop location.

- The code editor is a console-like user input area. The user can enter javascript commands and runs them by clicking on the "Run Script" button. Standard outputs and errors are displayed in the lower part of the editor.

- The table view on the right side displays the script environment state. It shows the variables currently stored in the script environment. For instance, if the user enters the command `var x = 2;`, a variable `x` of type `number` will be displayed.

A more detailed description of the code editor will come at the end of this introduction.

2.1.2 Menu

<table>
<thead>
<tr>
<th>File Menu</th>
<th>Option Menu</th>
</tr>
</thead>
</table>

In the menu, the user will find common options to create, open or save files. The **New Project** action, will simply create subfolders such as **ANSYS, Scripts, ETV, Database,** ... in the given folder in order for the user to organize his data.

The **Settings** option enables the user to define the paths to the ANSYS and Scilab executables. **If the program does not find those executables at start-up it will ask to modify the paths.**

2.2 Ansys Input

2.2.1 Description

When opening an Ansys Input file (*.inp) with FFC Editor, a new tab is opened.
2.2. ANSYS INPUT

It is divided in several parts:

- The central part is a classic text editor, with syntax coloration and completion matching ANSYS’ commands.
- The right part shows the structure of the Ansys input as defined by the XML markups. More detail about this later.
- The Run Ansys button will launch Ansys in graphical mode, after having executed the code marked as validated (green icons) in the input.

2.2.2 Structuring the data

Structuring a data is very useful when wanting to communicate either with humans or machines as it enables teamwork and automatisation of tasks. It is thus very important to try to structure as much as possible our inputs.

A very effective way to do this is the use of XML (eXtensible Mark-up Languages) who are now recognize world-wide and have been necessary in the development of the internet. Because they are now a reference, a lot of documentation and APIs are available.

2.2.3 Short introduction to XML languages

XML languages are used to structure data in a way that is comprehensible both by humans and machines. An XML code is structured by mark-ups. A mark-up can take to forms: opening/closing or single.

<table>
<thead>
<tr>
<th>Single markup</th>
<th>Opening/closing</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;myMarkup /&gt;</td>
<td>&lt;otherMarkup &gt; random text &lt;/otherMarkup &gt;</td>
</tr>
</tbody>
</table>

Mark-ups have a name (myMarkup or singleMarkup in the examples above). It is good to know that several mark-ups can have the same name and mark-up names do not accept spaces nor special characters.

Mark-up can also take attributes, to describe their contents or characteristics.

To define an attribute do as follow: <anotherMarkup attr = "something good to know">
You can put what you want as an attribute name or value. The value is always between quotes. One use of attributes can be to identify mark-ups. For example, if you have several mark-ups with the same name, you can define an attribute id in each mark-up to identify it uniquely.

According to the rules defined for XML language, all the mark-ups should be included in a global mark-up, called the root of the document, to have a valid XML document.

This knowledge of the XML language should be sufficient to start structuring our Ansys input.

### 2.2.4 Use in an Ansys input

Of course, if you introduce mark-ups such as `<header />` in an Ansys input you will get an Ansys error when you try to run the Ansys calculation, as it is not valid Ansys code.

So instead of writing mark-ups as `< ... >` you should write your mark-ups as `! < ... >` in the Ansys code.

For example you can write:

```xml
<!header status="validated" >
<!-- batch, list input written in the output file -->
<clear, result>
<!-- ANSYS transient icone of the file -->
<!-- incr, LOرحم00000; include error, warnings -->
<!-- coor, coordinate system expr:1 cartesian on expr:1 cylinder -->
<!-- disp, display coordinate system -->
<!-- title, Test Submodel1 -->
<!header>
<!parameters definition
10 = 50 label elementi over word element numbers
 sperm = 2 spacingfabo model outer inner
rl = 0.311 innerradius in an inner radius
```

Figure 2.2: Mark-up example

Such a code is valid in Ansys as the mark-ups are in comments, but it does not form a valid XML document (all the text in the document is not in a global mark-up). In order, to have a valid XML document a function has been implemented in the program. This function will transform our Ansys input into a valid XML document.

You can call this function by clicking on the Parse Document button. If it succeeded the structure of the document will appear as a tree. If an error occurred (a mark-up was not closed, invalid mark-up name,...) the line of the error will be indicated.

When the document has been parsed, you can now see the structure of the document. And also start doing interesting things.

Example of document structure:

Figure 2.3: Parsed document

### 2.2.5 Possible actions

When the user right-click on one item, a menu pops and proposes to validate or unvalidate the item. The effect of this action is to give the value validated or unvalidated to the attribute status in the mark-up as seen in the code example above.

The user can see if a mark-up and its content are validated or unvalidated with the small icons.
When clicking on the Run Ansys button, Ansys will be launch in the graphical mode only considering the validated code. This can be useful for debugging purposes, when the user wants to check the geometry without launching a whole calculation for example.

2.3 Transient

2.3.1 Description

In FFC Editor we define another type of file, namely transient files (*.trans). You can create a new transient file by going in File->New File and giving the .trans extension to your file.

When you open a transient file a new tab is opened:

![Transient Widget](image)

It is divided in several parts:

- The first part is where you enter the transient’s data. You can enter the time, temperature and alpha values.

- This part gives the user a graphical overview of the transient. You can rename the transient by double-clicking on the title and save it by clicking on the save button. You can also define long time periods by right-clicking on a section of the transient where the temperature is constant. This will be translated as a static calculation in Ansys (timi, off).

- This part gives an overview of the Ansys code corresponding to the transient.

This tool help you create and modify transient quite quickly and will allow the program to generate Ansys code for the transient automatically.
2.3.2 The transient mark-up

To include a transient in an Ansys input you can use a transient mark-up. This mark-up has this syntax: `!<transient url=" " comp=" "/>`. To insert such a mark-up in the text you can right-click in the Ansys editor and click on Insert Transient or write it yourself.

![Figure 2.5: Insert transient](image)

And the result:

```
<transient comp="" url="/">
```

![Figure 2.6: Transient inserted](image)

You can then fill in the attribute url which should be the absolute path to your transient file (for example `url="D:/My Documents/Project/Transient/transient.trans"`). To get the absolute path quicker you can use the drag-drop function from the system file view at the left of the screen.

The comp attribute should contain the name of the component you want to apply this transient to.

If you want to apply merge several transients together you should separate the urls and comps by a comma. For example if you want to apply transient1 to comp1 and transient2 at comp2 you should write: `!<transient url="url_transient1,url_transient2" comp="comp1,comp2"/>
```

To translate this mark-up into Ansys code you can parse the document, right-click on the transient mark-up in the document structure view and click on Update transient.

![Figure 2.7: Update transient](image)

If you put an invalid url, nothing will happen.
2.4 Databases

2.4.1 Description

When the user opens a database created with the software SQLITE (*.db) a new tab opens.

The use of this page is quite intuitive. The user can see and select the tables using the combo box at the top-left of the page. The user can rename the current table using the appropriate field and rename the data fields by double-clicking on the header.

The user can also delete a table or add new text files to the database. Text files that can be added to the database should have a matrix format with eventually a few header lines.

2.5 Code Editor

The core of the software is the Code editor that allows a powerful automatisation of calculation tasks and other features.

The code to be written in the Editor is Script code similar to Javascript.

2.5.1 A short introduction to Script codes

Script codes (such as Javascript) are polyvalent languages used in particular to provide dynamic features to websites. These codes are interpreted languages, i.e., they do not require to be compiled (i.e., translated to machine language) to be executed (C++ is not an interpreted language as it needs to be compiled into a *.exe to be executed). The script code is interpreted by a script engine. Script engines are present in web browsers for example.

FFC Editor uses the QtScript module provided in Qt to provide the user the ability to write his own calculation routines. Underlying C++ classes have been created to define new types in the script engine.
2.5.2 Custom types

FFC Editor makes several custom types available to the user in the script engine. These types are:

- AnsysInput
- AnsysOutput
- AnsysProgram
- TextFile
- RSTFile
- RTHFile
- Directory
- Transient
- Database
- ScilabProgram
This section regroups a bunch of small tutorials sorted by categories. If you are new you should start by reading the Scripts tutorials as it is a prerequisite to understand the other sections.

- Script Tutorials
- Ansys Tutorials
- Transients Tutorials
- Database Tutorials
- Scilab Tutorials
- Other Tutorials

3.1 Script Tutorials

3.1.1 What is Script?

Scripts languages are programming languages that stem from the ECMAScript language, defined in standard ECMA-262. The characteristic of Script languages is that they are interpreted languages in opposition with compiled languages.

C++ is a compiled language. You write a code that is transformed into a binary file (.exe) by a compiler. This .exe file is directly understandable by the computer. On the opposite, scripts do not require to be transformed to be executed. A program called the scriptengine reads the script file and transforms it into instructions at real time.

Due to this flexibility, Script languages such as Javascript are used extensively by web technologies.

3.1.2 Why using Script?

The fact that script languages are interpreted languages makes it possible for the user to create its own routine while using the functions defined in the software. This functions are very usefull to store data, manipulate Ansys file, extract data from RST or RTH files and so on.

Those functions makes it possible to perform a whole FFE calculation only from script code (apply a transient to an Ansys input file, run Ansys, extract the data for desired nodes, store data in a database, run scilab for ETV calculation,...). The strength of this approach is that once a routine has been written, it can be reused.

3.1.3 Where to write Script?

You can write script either directly in the Code Editor tab or in a seperate file. Usually, script files take the extension .js.
3.1.4 The basis of Script

3.1.4.1 Write a script

In this section we will teach you to write your first script. First of all, launch the AFC Editor software, you should see a window appearing that looks like below.

The zone 2 is called the code editor. It is a console-like widget. You can see the instructions that you have already written but you cannot edit the text that is above the prompt cursor.

Click in the code editor zone and write

```javascript
var a = 2;
```

Then click on the Run Script button.

The text browser under the code editor displays the output of the script engine. Normally, it should now be written `undefined` in this zone.

If you succeeded in doing this then you have just written and executed a script command.

3.1.4.2 Script Syntax

3.1.4.2.1 Instructions In general, script instructions are separated by a ; character as follow:

```javascript
instruction_1;
instruction_2;
instruction_3;
```

However, the ; character is not necessary if you have a new line between two instructions. The following code is correct

```javascript
instruction_1; instruction_2
instruction_3
```

3.1.4.2.2 Spaces The script is not sensitive to spaces. The following code is correct.

```javascript
instruction_1;
    instruction_1_1;
    instruction_1_2;
instruction_2; instruction_3;
```

3.1.4.2.3 Comments You can write comments in a script. You can write single line comments and multiple line comments.

```javascript
/* This script has three instructions
 - instruction_1
 - instruction_2
 - instruction_3 */
instruction_1;
instruction_2;
instruction_3; // End of the script
```

3.1.4.3 Variables

3.1.4.3.1 Declare a variable Declare a variable means allocate a memory space. Variable names can be alphanumeric characters it means letters [a-z] [A-Z], numbers [0-9] the dollar $ character and the underscore _.

Declare a variable is done as follow:

```javascript
var myVariable;
```

Variable names are case sensitive, with the following code you will declare three different variables:

```javascript
var myVariable;
var myvariable;
var MYVARIABLE;
```

Once a variable has been declared you can assign a value to it:

```javascript
var myVariable;
myVariable = 2;
```

You can do both in one instruction:

```javascript
var myVariable = 5.5;
```
3.1.4.3.2 Variable Types  Script languages are dynamically typed. This means that to declare a variable you only need to use the keyword `var`. A variable can also change its type once it has been assigned with a certain value.

- The *number* type: numbers
- The *string* type: characters strings
- The *boolean* type: `true` or `false`
- The *undefined* type: this is the equivalent of a `null` variable

The following code is correct:

```javascript
var myVariable;
myVariable = 2; // number type
myVariable = "a string variable"; // string type
myVariable = true; // a boolean variable
```

Variables can also be more complicated types like array or Objects. But we will see this later.

3.1.4.3.3 Operation  You can perform operations on variables.

- For numbers: `+,-,*,/,%`

The following code is correct:

```javascript
var a = 10, b = 2, c;
c = a+b; // c=12
c = a-b; // c=8
c = a*b; // c=20
c = a/b; // c=5
```

- For strings: `+`

The following code is correct:

```javascript
var a = "hello", b = "world", c;
c = a+" "+b+"!"; // c = "hello world!"
```

It is interesting to note that when you sum a number and a string the result is a string:

```javascript
var a = "I am ", b = 20, c;
c = a+b; // c = "I am 20"
```

- For booleans: `||` (or), `&&` (and), `!` (not)

The following code is correct:

```javascript
var a = true, b = false, c;
c = a||b; // c = true
```

3.1.4.3.4 Comparison operators  There are several comparison operators:

- `==` means equals to
- `!=` means different from
- `>` means superior to
- `>=` means superior or equals to
- `<` means inferior to
- `<=` means inferior or equals to

The result of a comparison is a boolean. For example:

```javascript
var a = 0, b = 1, c;
c = (a == b); // c = false
```

```javascript
var a = 0, b = 1, c;
c = (a == b); // c = true
```

```javascript
var a = 0, b = 1, c;
c = (a > b); // c = false
```
3.1.4.4 Basic Structures

3.1.4.4.1 if ... else You can use a if...else structure to execute a segment of code conditionally. What you need to write is:

```javascript
//condition is a boolean
if(condition)
{
    instructions;
}
```

If `condition` is `true` then the first group of instructions is executed, otherwise it is not executed. You can also use an `else` structure to specify the code to be executed in case `condition` is `false`.

```javascript
//condition is a boolean
if(condition)
{
    instructions;
}
else{ 
    instructions;
}
```

Finally, you can also join several conditions in a row like this:

```javascript
//condition is a boolean
if(condition)
{
    instructions;
} else if(condition)
{
    instructions;
} else{
    instructions
}
```

3.1.4.4.2 switch Sometimes instead of joining a lot of if...else structures, you may prefer to use a switch structure. This enables to probe the content of a variable and execute instructions depending on the content.

```javascript
var type = "temperature";
switch (type) {
    case "temperature":
        instructions;
        break;
    case "stress":
        instructions;
        break;
    case "linearised stress":
        instructions;
        break;
    default:
        instructions;
}
```

The `default` case contains the instructions to be executed in case no test matched.

3.1.4.4.3 while... loop The while loop can be used to execute a fragment of code several times, while a condition is verified. The syntax is the following

```javascript
while (condition) {
    instruction_1;
    instruction_2;
    instruction_3;
}
```

When the condition becomes `false` the loop stops and the execution of the script instructions goes further. When using this loop, one must pay attention that the contition will eventually be `false`. Otherwise you enter in to an infinite loop that will freeze the execution of the script.

Example:

```javascript
var number = 0;
while (number < 10) {
    number++; //this instruction is equivalent to number = number + 1;
}
```
3.1.4.4.4 for...loop  The for loop is very useful to execute a fragment of code several times. The syntax of the for loop is the following:

```javascript
for (initialisation; condition; incrementation)
{
    instruction_1;
    instruction_2;
    instruction_3;
}
```

This loop is optimised for incrementation. You declare a variable that will be incremented with each iteration. The classical way of using this loop is as follow:

```javascript
for (var i=0; i<10; i++)
{
    instruction_1;
    instruction_2;
    instruction_3;
}
```

Note: The variables declared inside a loop are not accessible outside the loop. Moreover, for performance reasons you should avoid declaring variables inside a loop.

```javascript
//Bad code
for (var i=0; i<10; i++)
{
    var a = i;
    a = 3; //this line will raise an error as the variable a is not declared in this scope
}
```

```javascript
//Good code
var a;
for (var i=0; i<10; i++)
{
    a = i;
}
```

### 3.1.4.5 Functions

Creating functions is very useful when you want to reuse or integrate a code fragment into another routine. To declare a function do as follow:

```javascript
function myFunction(arguments) {
    // Instructions that will be executed when the function is called
}
```

For example we can create a simple function:

```javascript
function sayHello() {
    print("Hello!"); //print itself is a function defined in the script engine.
}
```

```javascript
sayHello();
```

In the code above, we declare a function named `sayHello` that takes zero argument. We then call this function and the effect is that it displays the message "Hello!" in the output zone.

Functions can take an unlimited number of arguments. They can also return a value, to return a value one should use the keyword `return`. When a `return` instruction is encountered inside the function, be aware that the instructions that follow will not be executed.

```javascript
function add(number1, number2)
{
    return number1+number2;
    print("Function finished"); //this instruction will not be executed as it is after the return instruction
}
```

```javascript
var a = add(1,5); // a = 6
```

Note: The variables declared inside a function are not accessible outside the function. Example:

```javascript
function add(number1, number2){
    var result = number1+number2
    return result;
}
```

```javascript
add(1,5);
```

```javascript
print(result); // This instruction will raise an error as result is not declared in this scope
```
3.1.4.6 Objects

An Object is a concept. An Object has a structure that enables it to interact with other objects. In the script language there are native objects, defined in the script engine. We have already met such objects: a number, a string or a boolean. In fact, variables contain Objects.

An object contains three kind of things:

- A constructor: a list of instruction to execute when creating the object
- Properties: variables inside the object
- Methods: functions that can be applied to the object and its content

3.1.4.6.1 Array

Another native object is the array. An array can be declared as follow:

```javascript
var myArray = [42, 12, 6, 3];
```

//Alternative way
```javascript
var myArray = new Array(42, 12, 6, 3);
```

Arrays can contains other objects, that can be of different types. You can also access the elements of an array using [] and indexes. The indexes start at 0.

```javascript
var myArray_a = [42, 12, 6, 3];
var myArray_b = [42, "test", 6, true];
var x = myArray_a[0]; // a = 42
var y = myArray_a[2]; // a = 6
```

As an object, the array also has native methods: push, shift and pop.

```javascript
var myArray = [42, 12, 6, 3];
```

//push adds a value at the end of the array
```javascript
myArray.push(58); //myArray = [42, 12, 6, 3, 58]
```

//shift removes the first item of the array
```javascript
myArray.shift(); //removes 42
```

//pop removes the last item of the array
```javascript
myArray.pop(); //removes 58
```

//myArray = [12, 6, 3]

You can use a for loop to iterate over the values in an array:

```javascript
var myArray = [42, 12, 6, 3];
for(var i=0; i<myArray.length; i++){
    print(myArray[i]);
}
```

3.1.4.6.2 Custom Objects

Finally, you can define you own objects. It is not really useful for you now as useful custom types and their methods have already been created and exposed to the script engine. Those custom types are:

- AnsysInput
- AnsysOutput
- AnsysProgram
- TextFile
- RSTFile
- RTHFile
- Directory
- Transient
• Database

• ScilabProgram

To declare a new custom object you should use the keyword `new`:

```javascript
var ansys_input = new AnsysInput("D:/project/Ansys/model.inp");
```

See the individual class documentation or corresponding tutorials to get more information about the custom objects and their methods.

### 3.2 Ansys Tutorials

#### 3.2.1 Open the Ansys editor

To open the Ansys editor just open a .inp file. You can do this either in the File->Open menu or by double-clicking a .inp file in the system file view at the left of the main window.

#### 3.2.2 Create an AnsysInput object

AnsysInput objects make it possible to manipulate Ansys input files directly from the script code. There are two ways to instanciate an AnsysInput object. Either you declare it without a file path and then set the file path or declare immediately with a file path.

```javascript
var my_input = new AnsysInput("D:/project/Ansys/model.inp");
var my_other_input = new AnsysInput();
my_other_input.setFilePath("D:/project/Ansys/model.inp");
```

Remarks

The file paths have to be absolute

#### 3.2.3 Structure an Ansys Input

Structuring Ansys input is important as one can understand the structure of the file at first glance. Moreover, some parts can be changed directly from the script, which is helpful in an automation perspective. You can visualize the structure of the document by clicking on the button `Parse Document`. This will analyse the document and create a model of its structure. If an error has been encountered it will tell you at which line.

To structure a document you need to insert mark-ups in the file. Mark-ups are either single mark-ups `!<test />` or opening/closing mark-ups `!<test > .... !</test >`. In practice, the mark-up name describes its content. For example you can create an opening/closing mark-up `geometry` that will contain the Ansys code for the geometry.

Below is an example of an opening closing mark-up named `header` that contains the header ansys code.

You can embed mark-ups in other mark-ups and place single mark-ups where you want. Ansys code does have necessarily to be in mark-up in order for the analysis to work. All the mark-ups will be included in a global mark-up called `file`. For instance, below is a simple example of a structured ansys input and the corresponding model.

Remember, structuring the data will be usefull to manipulate it from the script in an automation perspective.

#### 3.2.4 Modify an Ansys Input from the script

You can modify an Ansys input file directly from the script. The first thing to do is declare an AnsysInput object with a correct file path to a *.inp file. Manipulating the content of the input file is don based on the structure you previously defined.
3.2.4.1 Set a mark-up text

You can use the `AnsysInput::setElementText` method to set the content of a mark-up in an input file. The first argument is the mark-up name while the second argument is the new text. The method returns `true` if success; `false` otherwise.

![Figure 3.1: before](image1.png)

```javascript
var myInput = new AnsysInput("D:/project/Ansys/example.inp");
myInput.setElementText("meshing", "new ansys code");
```

![Figure 3.2: after](image2.png)

```javascript
var myInput = new AnsysInput("D:/project/Ansys/example.inp");
myInput.setElementText("meshing", "new ansys code");
```

3.2.4.2 Apply a Transient

If you defined a transient mark-up in the Ansys input you can insert an Ansys code representing this transient directly from the script with the help of a transient file. This is done in three steps:

- Give the transient url (`AnsysInput::setTransientURL`)
- Give the name of the component to which we should apply the transient (`AnsysInput::setTransientComp`)
- Translate the transient file into Ansys code (`AnsysInput::applyTransient`)

```javascript
var myInput = new AnsysInput("D:/project/Ansys/example.inp");
myInput.setTransientUrl("D:/project/Transient/my_transient.trans");
myInput.setTransientComp("comp");
myInput.applyTransient();
```

Remarks

If you defined several transient mark-up, only the first will be modified.

3.2.5 Run Ansys from the Script

You can run an Ansys calculation in batch mode directly from the script. The first thing to do is declare an `AnsysInput` object with a correct file path to a *.inp file. You can drag and drop from the file system view to get the absolute file path. You then need to call the `run` method of the `AnsysProgram` object, giving it the `AnsysInput` object as argument.

```javascript
var ansysInput = new AnsysInput("D:/project/Ansys/model.inp");
var ansysOutput = ansys.run(ansysInput);
// ansysOutput is an AnsysOutput object
```
3.2.6 Extract data from a RST file

A RST file is a binary file created by an Ansys calculation that contains stress and temperature data for all nodes and for all load steps. You can extract those data for some nodes from a RST file into readable text files using the \texttt{RSTFile::extractNodes} method.

First you need to create get \texttt{RSTFile} object. There are two ways to do it: use \texttt{AnsyzOutput::getRST} or instantiate a new one with a path to a .rst file.

\begin{verbatim}
//First way, supposing that ans_output is an AnsyzOutput object
var rst = ans_output.getRST();

//Second way
var rst = new RSTFile("D:/project/Ansyz/stress.rst");
\end{verbatim}

Once you have the \texttt{RSTFile} object you can call the \texttt{RSTFile::extractNodes} method on it. This method takes 3 arguments:

- a list of arrays containing the interesting nodes
- a \texttt{Directory} object where the text files will be put
- (optional) a string used as a prefix for the text files names

This method returns an array of \texttt{TextFile} objects.

Remarks

For each node you have to give two numbers. The first number is the actual node number, the second is the node used to calculate linearised stress. If you do not want to calculate linearised stress, set the second node to zero.

\begin{verbatim}
//This code extracts temperature and non-linearised stress data for node 1 and node 4
var nodes = [1,0,4,0];
var dir = new Directory("D:/project/files");
var files = rst.extractNodes(nodes, dir);
\end{verbatim}

See Also

\texttt{RSTFile::extractNodes} \texttt{AnsyzOutput::getRST}

3.2.7 Extract data from a RTH file

A RTH file is a binary file created by an Ansys calculation that contains temperature data for all nodes and for all load steps. You can extract those data for some nodes from a RTH file into a readable text file using the \texttt{RTHFile::extractNodes} method.

First you need to create get \texttt{RTHFile} object. There are two ways to do it: use \texttt{AnsyzOutput::getRTH} or instantiate a new one with a path to a .rth file.

\begin{verbatim}
//First way, supposing that ans_output is an AnsyzOutput object
var rth = ans_output.getRTH();

//Second way
var rth = new RTHFile("D:/project/Ansyz/temp.rth");
\end{verbatim}

Once you have the \texttt{RTHFile} object you can call the \texttt{RTHFile::extractNodes} method on it. This method takes 3 arguments:

- a list of arrays containing the interesting nodes
- a \texttt{Directory} object where the text files will be put
- (optional) a string used as a prefix for the text file name

This method returns a \texttt{TextFile} object.

\begin{verbatim}
//This code extracts temperature data for node 1 and node 4
var nodes = [1,4];
var dir = new Directory("D:/project/files");
var file = rth.extractNodes(nodes, dir);
\end{verbatim}
3.3 Transients Tutorials

3.3.1 Create a Transient

3.3.1.1 Create a transient file

To create a transient file you can go in the File->New File menu. Then create a file with a .trans extension.

You can also create a transient file from a Transient object. You can do it by calling the method Transient::saveAs

```javascript
//Supposing that my_transient is a Transient object
my_transient.saveAs("D:/project/Transient/new_event.trans");
```

This code creates the file new_event.trans in the directory D:/project/Transient/ with the corresponding (time,temperature) values.

3.3.1.2 Create a Transient Object

You can instantiate a Transient object in the script by typing:

```javascript
var my_transient = new Transient();
```

This code will create an empty transient. A transient object holds vectors containing (time, temperature) values, static points, and heat transfer coefficients. When attributing a file path to a transient object, the file will be read and values attributed. The following code create a Transient object with values read in a file.

```javascript
//Supposing that the file event.trans defines a transient
var my_transient = new Transient("D:/project/Transient/event.trans");
```

//Alternative
```javascript
var my_transient = new Transient();
my_transient.setFilePath("D:/project/Transient/event.trans");
```

You can also instantiate a Transient object by giving the (time, temperature) values as parameters.

```javascript
var times = [0,1,100];
var temperatures = [0,100,100];
var my_transient = new Transient(times, temperatures);
```

This code creates a Transient object with the following (time, temperature) values: (0,0) (1,100) (100,100).

Remarks

The time is in seconds and temperature in degrees celsius.

3.3.2 Combine Transients

You can create new Transient objects by combining existing ones.

3.3.2.1 Append Transients

A useful way to create a new Transient object is to append one at the end of another. You can do this by calling the Transient::append method.

```javascript
var trans1 = ..., trans2 = ...;
var trans3 = trans1.append(trans2);
```
Remarks

The last temperature of \textit{trans1} and the first temperature of \textit{trans2} have to be equal. Otherwise the function raises an error.

3.3.2.2 Add Transients

Adding two transients is another way of combining them. Adding two transients is done by calling the \texttt{Transient::add} method.

\begin{verbatim}
var trans1 = ..., trans2 = ...;
var trans3 = trans1.add(trans2);
\end{verbatim}

3.3.2.3 Scale a transient

Finally, you can create a new \texttt{Transient} object by scaling an existing one (ie multiply its temperature values by a given factor). You can do this by calling the method \texttt{Transient::factor}.

\begin{verbatim}
var trans1 = ...;
var trans3 = trans1.factor(2);
\end{verbatim}

3.3.3 Write Transients into text files

The \texttt{Transient} widget makes it possible to create signals easily. One thing you might want to do is to write those transient in a matrix form in a text file. To do this you can use the \texttt{Transient::toMatrix} method. This method returns a string value containing the transient values as a matrix. The time step given as a parameter indicates how much time you wish to have between values.

Let us consider a simple example.
3.4 Database Tutorials

3.4.1 Create a Database

The Database object is an interface to interact with a real SQLITE database. To create a Database object do as follow:

```javascript
var db = new Database("D:/project/Database/my_database.db");
```

Remarks

- Database files take the extension .db
- If the path given in parameter is correct then creating the Database object will also create a database file. Having a Database object without a correct path is useless.

3.4.2 Add Data to the database

You can add data in the database from TextFile or FAMOSi (.txpn) files. You can do it either from the Database Widget or from the Code Editor. This will add a new table in the database filled with the values of the text file.

- In the Database Widget click on Add text files in database and choose the file you want to add.
- In the Code Editor with the method Database::add. The Database::add method takes a second optional parameter that indicates the table name.

```javascript
var db = new Database("D:/project/Database/my_database.db");
var file = new TextFile("D:/project/Files/signal.txpn");
db.add(file); //returns true on success; false otherwise
//Also possible
db.add(file, "table_name");
```

You can finally write this string in a text file using the TextFile::setContent method.

```javascript
var file = new TextFile("D:/project/Data/event.txt");
file.setContent(matrx1);
```

Figure 3.5: transient to matrix
Remarks
For this operation to succeed, the Text file should have a correct format. This means that the file should be of the form Header followed by a matrix. By Matrix we mean a succession of rows with the same number of columns. Columns are separated by space or tabulation characters.

Figure 3.6: Text File Structure

The header can be empty. The program considers that the matrix starts when it encounters two consecutive lines with the same number of columns. Example:

```
This is a data file    //5 columns
This is the header     //4 columns
random things         //2 columns
TIME TEMP S X         //3 columns : the matrix starts at this line
0 0 0 0               //3 columns
1 0.111 0.001         //3 columns
...
```

3.4.3 Extract data from the database

You can extract data from a database into a string variable. You can then write this string in a TextFile. There are two ways to extract data from the Database:

- Use the `Database::extract` method

- Use the `Database::extractColumns` method

Let us consider a database file `my_database.db` that has a table named `results_node_1`. 
In case you know the fields names use the `Database::extract` method. The `Database::extract` method takes two arguments. The table name from which you want to extract data and an array of string holding the fields to be extracted.

```javascript
var db = new Database("D:/project/Database/my\ndatabase.db");
var matrx = db.extract("results\nnode\n1", ["TIME", "S\nX", "S\nY"]);
```

If you do not know the fields names you can use the columns numbers instead with the `Database::extractColumns` method. The following code is equivalent to the previous code.

```javascript
var db = new Database("D:/project/Database/my\ndatabase.db");
var matrx_bis = db.extractColumns("results\nnode\n1", [1, 3, 4]);
```

Remarks

The column numbers start at 1 (this is because we hide the 0th column which is for the field ID).

3.5 Scilab Tutorials

3.5.1 Start Scilab

To start Scilab simply type:

```javascript
scilab.start();
```

This will start a Scilab process and open a new window displaying the Scilab process output.

The `ScilabProgram::start` function can also take one argument: the path of the Scilab process working directory. This argument can be really useful if you want to include Scilab routine files. In further commands, Scilab will try to resolve relative paths taking the working directory as origin.

For example, let’s say that we have a bunch of Scilab routines in the directory D:/project/scilab. Let’s say that some of those routines are interdependent, for example routine1.sce needs routine2.sce to work. In routine1.sce there is the instruction:

```javascript
exec('routine2.sce');
```

When we execute the statement `exec('routine2.sce');` Scilab look for the file `routine2.sce` in its current working directory. Example:
3.5.2 Execute a Scilab instruction
To execute a scilab instruction you first need to start scilab. You can then submit commands using the Scilab-Program::submit method.

scilab.start();
scilab.submit("a = 2");

3.5.3 Execute the content of a .sce file
You can execute the content of a file using the ScilabProgram::exec method. You first need to start scilab.

scilab.start("D:/project/scilab");
scilab.exec("routine1.sce");

3.5.4 Stop scilab
To stop scilab write:

scilab.stop();

3.6 Other Tutorials
In this sections you will find detailed and complete examples to fully understand the logic of the automation.

- **Step by step Fatigue Calculation**

## 4.1 Step by step Fatigue Calculation

### 4.1.1 Presentation

In this example an example of an automated fast fatigue calculation is documented. We will first calculate a reference transient response for a given geometry with Ansys. We will then use the reference response to calculate the response for a long transient using the ETV technique executed by scilab routines.

The commands are written in javascript and use the types defined in FFC Editor.

The first thing to do is to Open FFC Editor and create a new JavaScript file (*.js). You can execute the content of the Javascript file by pressing CRTL+R in the editor.

In order for our script code to be reusable, it would be better either to define a function or to get all the parameters at the top of the code in order to be easily accessible.

To define a new Function you can write:

```javascript
function fatigue_calculation(arguments) {
    //Code of the function
}
```

See Also

**Functions**

### 4.1.2 Parameter definition

Here we define several parameters that we will use later:

```javascript
// Fast Fatigue Calculation example
var path_to_project_dir = "D:/Johan/Automatisation_of_calculation_tools/6_Programs/Test";
//Defining interesting variables
//You only need to change these variables and the calculation will be done
var ansys_input_path = path_to_project_dir + "/ANSYS/Scheibmode1l.inp";
var transient_path = path_to_project_dir + "/Transients/transient.trans";
var comp = "n_inn"; /*composent where the transient should be applied*/

//To store results
var directory_path = path_to_project_dir + "Databases"; //Does not need to exist
var db_path = directory_path + "my_database.db";

var scenario_path = path_to_project_dir + "ETV/scenario.txt";
var reference_path = path_to_project_dir + "ETV/reference.txt";
var results_path = path_to_project_dir + "ETV/results_evt.txt";
```
4.1.3 Ansys Calculation

We will now instantiate a new `AnsysInput` object that we will use to manipulate our input file.

```javascript
// ex: var input = new AnsysInput("absolute_file_path");
var input = new AnsysInput();
input.setFilePath(ansys_input_path);
```

Let’s suppose that we have a transient markup in out .inp file, ie somewhere in the code is written: `<transient />`. The program is able to detect such a markup and insert some ansys code at this location.

To do this we need to define to more things:

- a path to a .trans file that defines our transient
- the component name to which the transient is to be applied

Finally we call the function `AnsysInput::applyTransient` to translate the markup into ansys code.

```javascript
input.setTransientURL(transient_path);
/*Set the attribute comp to "comp" in the first transient mark-up in the input*/
input.setTransientComp(comp);
/*Translate the mark-up to ansys code*/
input.applyTransient();
```

Once the transient is applied we can run Ansys.

```javascript
print("Running Ansys");
/*Performs the Ansys calculation in batch mode*/
var ans_output = ansys.run(input);
```

We stored the `AnsysOutput` object in a variable named `ans_output`. We will use this object to check if the calculation ended correctly and to get the result files.

```javascript
if(ans_output.exitcode == 0 || ans_output.exitcode == 8){
    //The variable output is of type AnsysOutput
    var rst = ans_output.getRST();
} else {
    //Print a message
    print("The Ansys calculation has not finished properly", ans_output.exitcode);
    //Open the output file in a new tab
    ans_output.open();
}
```

See Also

Ansys Tutorials
4.1.4 Extract Data

If the ansys calculation finished correctly we can continue our fatigue calculation. We stored the RSTFile object in the variable `rst`. We can apply the RSTFile::extractNodes function to retrieve stress data from nodes of interest.

The RSTFile::extractNodes function requires at least two arguments.

- an array containing the nodes numbers
- a Directory object where the result text files will be written

For each node of interest, you should give two numbers. The first number is the node of interest and the second number is the node used to calculate linearised stress. If you don’t want to calculate linearised stress or if you are not sure what to do then SET THE SECOND NODE TO ZERO.

```javascript
var nodes = interesting_nodes;
var dir = new Directory(directory_path);
//Extract the stress and temperature data from the rst files for the given nodes
//This will create text files, in the given directory, with the given prefix
//Here it will be "results_1.txt" and "results_2.txt"
//NB: If no prefix is given, the default prefix is the name of the rst file
print('Extracting results');
var result_files = rst.extractNodes(nodes, dir, "results");
```

The variable result_files contains a list of TextFile objects. We can add TextFile objects to Database object. We instantiate a Database object and add our result files to the database.

If the database does not exist it will be created automatically.

```javascript
var db = new Database(db_path);
//The variable result_files is of type Array[TextFile] and contains the result files
for(var i=0; i<result_files.length; i++)
{
  //Adds the result files to the database
  //NB: The table name will be the file name, but you can also indicate a custom table name
  //ex: db.add(result_files[i], "custom_table_name");
  var table_name = "results_node"+nodes[2*i];
  db.deleteTable(table_name);
  db.add(result_files[i], table_name);
}
```

4.1.5 ETV calculation

We have now extracted our stress data for the reference transient. We can now perform the ETV calculation. Our scilab routine takes three arguments in addition to the calculation parameters.

- the path to a text file containing the long transient data
- the path to a text file containing the reference data
- the path to a text file where the results will be written

We prepare the file containing the reference data by extracting data from the database.

```javascript
var reference = new TextFile(reference_path);
var tables = db.tables();
//Sets the content of the reference file with the data contained in the field "TIME", "TEMP" and "SAX"
of the first table of db
//NB: we could also put the number of the columns
//ex: reference.setContent(db.extract(tables[0], [1, 2, 3]));
```

To perform the ETV calculation we need to start scilab and to submit it our commands.
4.1.6 Process data

We can now store the results in the database and plot them.

db.add(result_etv);
result_etv.plot();
5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

- **AnsysInput**
  - *.inp file
  - Page 35
- **AnsysOutput**
  - Output of an ANSYS calculation
  - Page 37
- **AnsysProgram**
  - Enables to interact with ANSYS
  - Page 39
- **Database**
  - *.db file
  - Page 40
- **Directory**
  - Page 41
- **RSTFile**
  - Page 42
- **RTHFile**
  - Page 43
- **ScilabProgram**
  - The ScilabProgram enables to start, stop and run scilab routines
  - Page 44
- **Signal**
  - List of time values associated with a list of other values
  - Page 45
- **TextFile**
  - Represent a text file
  - Page 47
- **Transient**
  - Represent a transient signal
  - Page 48
6.1 AnsysInput Class Reference

The AnsysInput class represents a *.inp file.

Inherits QObject, and QScriptable.

Public Slots

- void open()
- bool setTransientURL (QString url, QString id=QString())
- bool setTransientComp (QString comp, QString id=QString())
- bool applyTransient (QString id=QString())
- QString getElementText (QString element)
- bool setElementText (QString element, QString elem_text)
- bool setElementAttribute (QString element_name, QString attribute, QString value)
- QStringList childElements (QString element_name=QString())

Properties

- QStringList resultFiles
  the absolute paths to the result files.
- QString filePath
  the path to the *.inp file.

6.1.1 Detailed Description

The AnsysInput class represents a *.inp file.

This class provides functions to modify and get information about ansys inputs. You can create a new AnsysInput object in the script console with the command:

```javascript
var inp = new AnsysInput("path_to_file");
```

When setting the filePath of an AnsysInput object, the *.inp file is opened and analysed automatically. The program looks into the ansys code for the commands file and /output. From the parameters of those commands, it builds the absolute paths to the potential result files.

Functions have been implemented to switch from a valid Ansys input file to a valid XML document. The manipulation of the file is possible through the script.

The AnsysInput class is typically used like this.
6.1. ANSYSINPUT CLASS REFERENCE

var path = "D:/directory/example.inp";
var input = new AnsysInput(path);

var transient_path = "D:/directory/transient.trans";
input.transientURL(transient_path);
input.transientComp("comp");
input.applyTransient();
ansys.run(input);

6.1.2 Member Function Documentation

6.1.2.1 bool AnsysInput::applyTransient ( QString id = QString() ) [slot]
Opens the *.inp file and look for a <transient> mark-up. You can use the parameter id to identify more precisely a mark-up. If the parameter id is not indicated the function will take the first <transient> mark-up of the file.
Sets the text of the transient mark-up to ansys code automatically generated.
<transient url="..." comp="..." /> is transformed to <transient url="..." comp="..." > [ansys code] </transient >.
Returns true on success; false otherwise.

6.1.2.2 QStringList AnsysInput::childElements ( QString element_name = QString() ) [slot]
Returns the childs elements of the element named element_name.
If element_name is empty, returns the root elements of the file (same as AnsysInput::childElements("file")).

6.1.2.3 QString AnsysInput::getElementText ( QString element ) [slot]
Returns the text contained in the mark-up named element.

6.1.2.4 void AnsysInput::open ( ) [slot]
Opens the AnsysInput in a new tab. The tab contains an AnsysWidget object.

6.1.2.5 bool AnsysInput::setElementAttribute ( QString element_name, QString attribute, QString value ) [slot]
Sets the text attribute attribute in the mark-up named element to value.
Returns true on success; false otherwise.

6.1.2.6 bool AnsysInput::setElementText ( QString element, QString elem_text ) [slot]
Sets the text contained in the mark-up named element to elem_text.
Returns true on success; false otherwise.

6.1.2.7 bool AnsysInput::setTransientComp ( QString comp, QString id = QString() ) [slot]
Opens the *.inp file and look for a <transient> mark-up. You can use the parameter id to identify more precisely a mark-up. If the parameter id is not indicated the function will take the first <transient> mark-up of the file.
Sets the attribute comp of the mark-up to the value given by the parameter comp. <transient comp="..." >.
Returns true on success; false otherwise.

6.1.2.8 bool AnsysInput::setTransientURL ( QString url, QString id = QString() ) [slot]
Opens the *.inp file and look for a <transient> mark-up. You can use the parameter id to identify more precisely a mark-up. If the parameter id is not indicated the function will take the first <transient> mark-up of the file.
Sets the attribute url of the mark-up to the value given by the parameter url. <transient url="..." >.
Returns true on success; false otherwise.

34
6.1.3 Property Documentation

6.1.3.1 QStringList AnsysInput::resultFiles [read], [write]

the absolute paths to the result files.

When setting the filePath of an AnsysInput object, the *.inp file is opened and analysed automatically. The program looks into the ansys code for the commands file and /output. From the parameters of those commands, it builds the absolute paths to the potential result files.

The documentation for this class was generated from the following files:

- script_classes/ansysinput.h
- script_classes/ansysinput.cpp

6.2 AnsysOutput Class Reference

The AnsysOutput class represents the output of an ANSYS calculation.

Inherits QObject, and QScriptable.

Public Slots

- QScriptValue getRST () const
- QScriptValue getRTH () const
- void open ()

Properties

- int exitcode
  the exit code of the ansys calculation.
- QString filePath
  the path to the *.out file.
- QStringList resultFiles
  the absolute paths to the result files (*.rst or *.rth).

6.2.1 Detailed Description

The AnsysOutput class represents the output of an ANSYS calculation.

In general you dont need to instanciate an AnsysOutput object with new. An AnsysOutput object is created as a result of an Ansys calculation.

```cpp
//if input is an AnsysInput object
var ansys_output = ansys.run(input);
```

This object allows to have a quick overview of the output of a calculation and to extract effectively the results.

In particular, you can retrieve the results file through the functions AnsysOutput::getRST() and AnsysOutput::getRTH().

```cpp
//if input is an AnsysInput object
var ansys_output = ansys.run(input);
if(ansys_output.exitcode == 0 || ansys_output.exitcode == 8){
  //calculation was successful: We continue the processing of the rst file
  var rst = ansys_output.getRST();
} else{
  //calculation failed: We print a message and display the *.out file
  print('The Ansys calculation finished with exitcode ', ansys_output.exitcode);
  ansys_output.open();
}
```

35
6.2.2 Member Function Documentation

6.2.2.1 QScriptValue AnsysOutput::getRST ( ) const [slot]
Returns a RSTFile* object generated by the calculation if it exists; returns an empty ScriptValue otherwise.

6.2.2.2 QScriptValue AnsysOutput::getRTH ( ) const [slot]
Returns a RTHFile* object generated by the calculation if it exists; returns an empty ScriptValue otherwise.

6.2.2.3 void AnsysOutput::open ( ) [slot]
This function opens the *.out file and displays it in a new tab.

6.2.3 Property Documentation

6.2.3.1 int AnsysOutput::exitcode [read], [write]
the exit code of the ansys calculation.
This property is useful to detect if a calculation was successful or not.

6.2.3.2 QStringList AnsysOutput::resultFiles [read], [write]
the absolute paths to the result files (*.rst or *.rth).
The AnsysOutput inherits this property from the AnsysInput object. All the files in this list exists.
The documentation for this class was generated from the following files:
- script_classes/ansysoutput.h
- script_classes/ansysoutput.cpp

6.3 AnsysProgram Class Reference

The AnsysProgram class enables to interact with ANSYS.
Inherits QObject, and QScriptable.

Public Slots
- QScriptValue run (const AnsysInput *input, QScriptEngine *pengine=0)

Properties
- QString filePath
  the path to the ANSYSXXX.exe file where XXX is the version of Ansys.

6.3.1 Detailed Description

The AnsysProgram class enables to interact with ANSYS.
There is only one object of this class in the script engine global object. This object is named ansys.
To change the path to the ansys executable file, go in the menu Options>Settings. To check the path to the Ansys executable file you can use the command
ansys.filePath

The ansys object is typically used like this.

// supposing that ansys_input is an AnsysInput object
var ansys_output = ansys.run(ansys_input);
// where ansys_output is an AnsysOutput object
6.3.2 Member Function Documentation

6.3.2.1 QScriptValue AnsysProgram::run ( const AnsysInput * input, QScriptEngine * pengine = 0 ) [slot]
Launch Ansys and run a calculation in batch mode with the given input.
Returns an AnsysOutput value.

6.3.3 Property Documentation

6.3.3.1 QString AnsysProgram::filePath [read], [write]
the path to the ANSYSXXX.exe file where XXX is the version of Ansys.
This file path is saved in the settings of the program. If the program does not find this file, it will tell it at start up. You can then change the file path under the menu Options>Settings.
The documentation for this class was generated from the following files:
• script_classes/ansysprogram.h
• script_classes/ansysprogram.cpp

6.4 Database Class Reference

The Database class represents a *.db file.
Inherits QObject, and QScriptable.

Public Slots
• bool add (TextFile *, QString table_name=QString(""))
• QStringList tables ()
• QString extract (QString table_name, QStringList fields=QStringList())
• QString extractColumns (QString table_name, QList<int> fields=QList<int>())
• bool deleteTable (QString table_name)
• bool plot (QString table_name, QStringList fields=QStringList())
• void open ()

Properties
• QString filePath
  the absolute path to the *.db file.

6.4.1 Detailed Description

The Database class represents a *.db file.
This class is used to create and manipulate databases. You can create a Database object with the script code:
var data = new Database("D:/directory/my database.db");

If the database file does not exists it will be created.
You can add, retrieve or delete data from the database using the functions Database::add, Database::extract and Database::deleteTable

The Database class is typically used like this:
var data = new Database("D:/directory/my database.db");
var text_files = rst.extractNodes(nodes, dir);

//Add the text files to the database
for(var i=0; i<text_files.length; i++){
  data.add(text_files[i]);
}

//Get the tables' names
var tables = data.tables();
for(var i=0; i<tables.length; i++){
    var values = data.extract(tables[i], ['TIME', 'TEMP']);
    //Continue processing the data
}

6.4.2 Member Function Documentation

6.4.2.1 bool Database::add ( TextFile * text_file, QString table_name = QString("") ) [slot]
This function adds the given TextFile text_file to the database as a new table named table_name. If table_name is empty, the new table will have the name of the TextFile. Returns true on success; false otherwise.

Remarks
For this operation to succeed, the Text file should have a correct format. This means that the file should be of the form Header followed by a matrix. By Matrix we mean a succession of rows with the same number of columns. Columns are separated by space or tabulation characters.

6.4.2.2 bool Database::deleteTable ( QString table_name ) [slot]
Deletes the table table_name. Returns true on success; false otherwise.

6.4.2.3 QString Database::extract ( QString table_name, QStringList fields = QStringList() ) [slot]
Returns a String containing the columns fields of the table table_name.

6.4.2.4 QString Database::extractColumns ( QString table_name, QList< int > columns = QList< int >() ) [slot]
Returns a String containing the columns columns of the table table_name.

Warning
The columns numbers start at 1.

6.4.2.5 void Database::open ( ) [slot]
Opens the database file in a new tab.

6.4.2.6 bool Database::plot ( QString table_name, QStringList fields = QStringList() ) [slot]
This function reads the database, writes a .graph file from the table table_name and the given fields and displays it. If the parameter fields is empty the all the fields will be displayed.

The X values are always the first field given, while the Y values are the other fields.

6.4.2.7 QStringList Database::tables ( ) [slot]
Returns the list of the database's tables.

The documentation for this class was generated from the following files:

• script_classes/database.h
• script_classes/database.cpp

6.5 Directory Class Reference

The Directory class represents a directory.
Inherits QObject, and QScriptable.
Public Slots

- QStringList getFiles (QString ext=QString())

Properties

- QString filePath

  the absolute path to the directory.

6.5.1 Detailed Description

The Directory class represents a directory.

6.5.2 Member Function Documentation

6.5.2.1 QStringList Directory::getFiles ( QString ext = QString() ) [slot]

This function returns the list of absolute file paths, for files in the directory that have the extension ext.

If ext is empty the function returns all the files in the directory.

The documentation for this class was generated from the following files:

- script_classes/directory.h
- script_classes/directory.cpp

6.6 RSTFile Class Reference

The RSTFile class represents a RSTFile.

Inherits QObject, and QScriptable.

Public Slots

- QScriptValue extractNodes (QList<int>, Directory*, QString)

Properties

- QString filePath

  the absolute path to the *.rst file.

6.6.1 Detailed Description

The RSTFile class represents a RSTFile.

This class is used to manipulate rst files. You can use the RSTFile::extractNodes function to retrieve data. It is typically used like this:

```javascript
var rst = ansys_output.getRST();
var nodes = [1,0,4,2];
var dir = new Directory("D:/directory");
var prefix = "results";

// Create two text files in the directory dir for nodes 1 and 4
// containing the stress data for those two nodes.
// The data for node 4 are linearised using the node 2
var result_files = rst.extractNodes(nodes, dir, prefix);
```
6.6.2 Member Function Documentation

6.6.2.1 QScriptValue RSTFile::extractNodes ( QList< int > nodes, Directory * destination, QString prefix ) [slot]

This function can be used to extract temperature and stress data from the RST file. The parameter nodes holds the list of interesting nodes. For one node you should enter two values, the first value is the node and the second the node is used to calculate the linearised stress. If you don't want to calculate linearised stress then the second value should be 0.

For example, if you want to extract the data for the node 1 and 56, while linearising the stress along the path to node 2 for node 1 then nodes = [1, 2, 56, 0].

The function will launch an Ansys routine to extract the relevant data:

- TIME
- TEMPERATURE
- S_X, S_Y, S_Z
- S_XY, S_YZ, S_XZ
- S_INTENSITY, S_EQUIVALENCE
- (MEMBRANE+BENDING)_X_Y_Z (if linearised)
- (MEMBRANE+BENDING)_XY_YZ_XZ (if linearised)

The extracted data will be written in text files in the directory destination. The text files will be named prefix followed by the node number.

For the example above, if prefix = "results" then the text files will be named "results_1_path_2.txt" and "results_56.txt".

The function returns a list of TextFile objects.

The documentation for this class was generated from the following files:

- script_classes/rstfile.h
- script_classes/rstfile.cpp

6.7 RTHFile Class Reference

The RTHFile class represents a RTHFile.

InheritsQObject, and QScriptable.

Public Slots

- QScriptValue extractNodes (QList< int >, Directory *, QString)

Properties

- QString filePath

  the absolute path to the *.rth file.

6.7.1 Detailed Description

The RTHFile class represents a RTHFile.

This class is used to manipulate rth files. You can use the RTHFile::extractNodes function to retrieve data. It is typically used like this:
var rth = ansys.output.getRTH();
var nodes = [1,4,2];
var dir = new Directory("D:/directory");
var prefix = "results";
// Create one text file in the directory dir for nodes 1, 4 and 2
// containing the temperature data for those three nodes.
var result_files = rth.extractNodes(nodes, dir, prefix);

6.7.2 Member Function Documentation

6.7.2.1 QScriptValue RTHFile::extractNodes ( QList<int> nodes, Directory * destination, QString prefix ) [slot]

This function can be used to extract temperature data from the RTH file. The parameter nodes holds the list of
interesting nodes.

The function will launch an Ansys routine to extract the relevant data:

- TIME
- TEMPERATURE for the different nodes
  The extracted data will be written in a text file in the directory destination. The text file will be named
  prefix.

The function returns a TextFile object.

The documentation for this class was generated from the following files:

- script_classes/rthfile.h
- script_classes/rthfile.cpp

6.8 ScilabProgram Class Reference

The ScilabProgram enables to start, stop and run scilab routines.
Inherits QObject, and QScriptable.

Public Slots

- void stop ()
- bool start (QString workingDir=QString())
- void submit (QString)
- void exec (QString)

Properties

- QString filePath
  the path to the Scilex.exe file.

6.8.1 Detailed Description

The ScilabProgram enables to start, stop and run scilab routines.

6.8.2 Member Function Documentation

6.8.2.1 void ScilabProgram::exec ( QString filePath ) [slot]

This function reads a file whose absolute path is filePath and submits its content to scilab.
For example you can write, 'scilab.exec("D:/my_directory/routines/my_routine.sce");'
6.8.2.2 bool ScilabProgram::start (QString workingDir = QString()) [slot]

This function starts scilab in with the given workingDir as working directory.
If workingDir is empty then the working directory of the scilab process will be the current directory.
Indicating a working directory for scilab is useful for link resolution.
For example, if you start scilab in the directory my_directory you can then indicate relative paths from this
directory to scilab.
If we now submit the command "exec('routines/my_routine.sce');" to scilab, scilab will look for the file my_routine.sce in the directory my_directory/routines.
This function returns true on success; false otherwise.

6.8.2.3 void ScilabProgram::stop () [slot]

This function stops the scilab process. Does nothing if scilab is not started.

6.8.2.4 void ScilabProgram::submit (QString command) [slot]

This function submits command to scilab. Does nothing if scilab is not started.
For example you can write, 'scilab.submit("a = 2");'
The documentation for this class was generated from the following files:
• script_classes/scilabprogram.h
• script_classes/scilabprogram.cpp

6.9 Signal Class Reference

The Signal class represents a list of time values associated with a list of other values.
Inherits QObject, and QScriptable.

Public Slots
• QScriptValue getTime () const
• QScriptValue getValue () const
• void setTime (QScriptValue vect)
• void setValue (QScriptValue vect)
• QString toMatrix (double time_step=-1)
• Signal * factor (double coefficient)
• Signal * add (Signal *trans)
• Signal * addTimeValues (QScriptValue new_values)
• Signal * linearise (double time_step=1)
• Signal * reduce (double sensitivity)
• QString size ()

6.9.1 Detailed Description

The Signal class represents a list of time values associated with a list of other values.
The Signal class has two inner lists of doubles, one representing time values and the other representing the
associated values. Interesting can be performed on Signals such as linearisation, adding two signals, multiplying
signals. The time and value lists have to have to same size in order for the operations on Signal objects to succeed.

6.9.2 Member Function Documentation

6.9.2.1 Signal * Signal::add (Signal * sig) [slot]

This function is used to add one signal to another. The twoSignal object used as parameter are not modified. The
missing time values are linearly interpolated.
6.9.2.2 Signal * Signal::addTimeValues ( QScriptValue new_values ) [slot]
This function is used to add time values to the Signal. This function takes an array of numbers in parameter, the corresponding values will be interpolated from existing ones.

6.9.2.3 Signal * Signal::factor ( double coefficient ) [slot]
This function is used to multiply a Signal by a coefficient. The Signal used as parameter is not modified.

6.9.2.4 QScriptValue Signal::getTime ( ) const [slot]
This function returns the array of numbers holding the time values.

6.9.2.5 QScriptValue Signal::getValue ( ) const [slot]
This function returns the array of numbers holding the values.

6.9.2.6 Signal * Signal::linearise ( double time_step = 1 ) [slot]
This function is used to linearise a Signal, ie have stamped times values. The parameter is used to specify the step separating time values. The Signal object used as a parameter is not modified.

6.9.2.7 Signal * Signal::reduce ( double sensitivity ) [slot]
This function is used to reduce the number of values of the Signal. The Signal used as a parameter is not modified.

6.9.2.8 void Signal::setTime ( QScriptValue vect ) [slot]
This function takes an array of number as a parameter and assigns it as the time values of the Signal.

6.9.2.9 void Signal::setValue ( QScriptValue vect ) [slot]
This function takes an array of number as a parameter and assigns it as the values of the Signal.

6.9.2.10 QString Signal::size ( ) [slot]
This function returns a string giving the size of the Signal time and value lists.

6.9.2.11 QString Signal::toMatrix ( double time_step = -1 ) [slot]
This function returns a string form of the Signal. The parameter time_step specifies the desired step between time values.

The documentation for this class was generated from the following files:
- script_classes/signal.h
- script_classes/signal.cpp

6.10 TextFile Class Reference

The TextFile class represent a text file.
Inherits QObject, and QScriptable.

Public Slots
- bool hasHeader ( ) const
- void open ( )
- QString getContent ( ) const
- void setContent (QString)
- bool plot ( )
Properties

- QString filePath
  
  the absolute path to the file.

6.10.1 Detailed Description

The TextFile class represents a text file.

6.10.2 Member Function Documentation

6.10.2.1 QString TextFile::getContent () const [slot]

This function reads the text file and returns its content as a string.

6.10.2.2 bool TextFile::hasHeader () const [slot]

This function returns whether the TextFile has a header or not.

6.10.2.3 void TextFile::open () [slot]

This function reads the TextFile and displays its content in a new tab.

6.10.2.4 bool TextFile::plot () [slot]

This function plots the content of the text file in a new tab.

   The text file should contain a matrix in order for the operation to succeed. The first column is taken as X values and the other columns as Y values.

6.10.2.5 void TextFile::setContent (QString text) [slot]

This function opens the text file and replaces its content by text.

The documentation for this class was generated from the following files:

- script_classes/textfile.h
- script_classes/textfile.cpp

6.11 Transient Class Reference

The Transient class represents a transient signal.

Inherits QObject, and QScriptable.

Public Slots

- Transient * append (Transient *trans)

  This function appends two transients and returns a new Transient object.

- Transient * factor (double coefficient)

  This function creates a new Transient objects which temperature values are those of the Transient passed as parameter multiplied by the coefficient.

- Transient * add (Transient *trans)

  This function adds two Transient objects and returns a new Transient object.

- void open ()

- bool saveAs (QString filePath)

  This function creates a new .trans file with the values of the Transient object passed as parameter.
Properties

- QString filePath
  
  the absolute path to the *.trans file.

6.11.1 Detailed Description

The Transient class represent a transient signal. This class is used to create, manipulate and store transients. The Transient object consist basically of 2 vectors containing values for time and temperature. It can also store values defining long points (static calculation in ANSYS) and heat transfert coefficients.

The class.

6.11.2 Member Function Documentation

6.11.2.1 Transient * Transient::add ( Transient * trans ) [slot]

This function adds two Transient objects and returns a new Transient object.

The Transient objects passed as parameters are not modified. This function is useful to create new Transient objects from existing ones.

See Also

Combine Transients

6.11.2.2 Transient * Transient::append ( Transient * trans ) [slot]

This function appends two transients and returns a new Transient object.

The two transients passed as parameter are not modified. This function is useful to create new Transient objects from existing ones.

See Also

Combine Transients

6.11.2.3 Transient * Transient::factor ( double coefficient ) [slot]

This function creates a new Transient objects which temperature values are those of the Transient passed as parameter multiplied by the coefficient.

The Transient passed as parameter is not modified.

See Also

Combine Transients

6.11.2.4 void Transient::open ( ) [slot]

This function opens and displays the Transient in a new tab.

The documentation for this class was generated from the following files:

- script_classes/transient.h
- script_classes/transient.cpp
add
  Database, 40
  Signal, 46
 Transient, 49
addTimeValues
  Signal, 46
AnsysInput, 35
  applyTransient, 36
  childElements, 36
  getElementText, 36
  open, 36
  resultFiles, 37
  setElementAttribute, 36
  setElementText, 36
  setTransientComp, 36
  setTransientURL, 37
AnsysOutput, 37
  exitcode, 38
  getRST, 38
  getRTH, 38
  open, 38
  resultFiles, 38
AnsysProgram, 39
  filePath, 39
  run, 39
append
  Transient, 49
applyTransient
  AnsysInput, 36
childElements
  AnsysInput, 36
Database, 40
  add, 40
  deleteTable, 41
  extract, 41
  extractColumns, 41
  open, 41
  plot, 41
  tables, 41
deleteTable
  Database, 41
Directory, 41
  getFiles, 42
exec
  ScilabProgram, 45
exitcode
  AnsysOutput, 38
extract
  Database, 41
  extractColumns
    Database, 41
  extractNodes
    RSTFile, 43
    RTHFile, 44
factor
  Signal, 46
  Transient, 49
filePath
  AnsysProgram, 39
getContent
  TextFile, 47
element
  AnsysInput, 36
getFiles
  Directory, 42
getRST
  AnsysOutput, 38
getRTH
  AnsysOutput, 38
getTime
  Signal, 46
ggetValue
  Signal, 46
hasHeader
  TextFile, 47
linearise
  Signal, 46
open
  AnsysInput, 36
AnsysOutput, 38
Database, 41
TextFile, 47
Transient, 49

plot
Database, 41
TextFile, 48

RSTFile, 42
extractNodes, 43

RTHFile, 43
extractNodes, 44

reduce
Signal, 46

resultFiles
AnsysInput, 37
AnsysOutput, 38

run
AnsysProgram, 39

ScilabProgram, 44
exec, 45
start, 45
stop, 45
submit, 45

setContent
TextFile, 48

setParameterAttribute
AnsysInput, 36

setParameterText
AnsysInput, 36

setTime
Signal, 46

setTransientComp
AnsysInput, 36

setTransientURL
AnsysInput, 37

setValue
Signal, 46

Signal, 45
add, 46
addTimeValues, 46
factor, 46
gmtime, 46
getValue, 46
linearise, 46
reduce, 46
setTime, 46
setValue, 46
size, 47
toMatrix, 47

size
Signal, 47

start
ScilabProgram, 45

stop
ScilabProgram, 45

submit
ScilabProgram, 45

tables
Database, 41
TextFile, 47
getContent, 47
hasHeader, 47
open, 47
plot, 48
setContent, 48
toMatrix
Signal, 47

Transient, 48
add, 49
append, 49
factor, 49
open, 49