AMPS Model Data Base Interface Reference Manual

Revision 1.31, October 2005

Table of Contents

REVISION HISTORY: **2**

CHAPTER 1: INTRODUCTION **3**

CHAPTER 2: CREATING AMPS MDB **4**

CHAPTER 3: ACCESSING AMPS MDB **6**

CHAPTER 4: MDB DATA ENTRY REFERENCE **9**

4.1 NAMING CONVENTION 9
4.2 MDB DATA ENTRY DESCRIPTION 9
4.2.1 System Controls Data *10*
4.2.2 Elements, Nodes, and Geometry Information *13*
4.2.3 Nodal Local Coordinate System *15*
4.2.4 Material Property Definitions *15*
4.2.5 Boundary Conditions *20*

CHAPTER 5: CREATING RESULT FILE FOR AMPS POST-PROCESSING **25**

5.1 CREATING NODAL-BASED RESULTS 25
5.2 CREATING ELEMENT-BASED RESULTS 27

APPENDIX: NAMING AND NUMBERING SCHEME IN AMPS SYSTEM **30**

Revision History:

Revission 1.4 October 14, 2005
Add the following MDB items
EXPLICIT[3.05]

Revision 1.3: May 10, 2005
Add the following MDB items
FIELD_LORENTZ_FORCE[2.2], PROP_USER_PLASTICITY[2.3], PROP_USER_INITDATA[2.3], NODE_MASS_LUMPED[2..4], UNIT_LENGTH_SCALING[2.5], UNIT_MASS_SCALING[2.5], UNIT_TEMP_SCALING[2.5], UNIT_TEMP_SCALING[2.5], PROP_MULTIPLIER_DATA[2.6]

Revision 1.2: October 5 2004
Add the following MDB entries:
PROP_ELECTRIC_CONDUCTIVITY[1.762], PROP_ELECTRIC_PERMITIVITY[1.762], PROP_MAGNETIC_PERMEABILITY[1.762], FIELD_EPOTENTIAL[1.762], FIELD_EFLUX[1.762], FIELD_MPOTENTIAL[1.762], FIELD_MFLUX[1.762], OPTIONS_ELECTRRIC_DIELECTRIC[1.763], ROP_THERMAL_VOLUMEFLUX[1.763], PROP_ELECTRIC_VOLUMECHARGE[1.763], PROP_ELECTRIC_JOULETOTHERMAL[1.763], Revised OPT_RESTART[1.763], PROP_VISCO_POWERCREEP[1.764], PROP_USER_MATLAW[1.766], BC_USER_DISTRIBUTED[1.762], BC_USER_NODAL[1.762], OPT_MAXMEMORY[1.768], PROP_THERMAL_ANISOTROPIC_CONDUCTIVITY[2.01], PROP_ELASTIC_ANISOTROPIC_EXPANSION[2.01], BC_OVERSET[2.1], OVERSET_NODE[2.1], OVERSET_ELEMENT[2.1]

Revision 1.1: March 15 2003
Add the following MDB entries:
NODE_LOCAL_COORD[1.751], NODE_LOCAL_COORD_DATA[1.751], OPT_ROTATINGFRAME[1.751], OPT_ ROTATINGFRAME _TIMETABLE[1.751], BC_CONSTRAINT[1.76], SYS_TIMESTEP_OUTPUT[1.76].

Chapter 1: Introduction

AMPS (Advanced Multi-Physics Simulation) package contains three components:

AMPSolid: the solid modeling and FE mesh generation program,
AMPView: FE pre-processor and post-processor, and
AMPSol: The finite element program.

All these three packages utilize a common database to communicate with the others. As such, for an external application program to use AMPS as a tool for pre-/post-processing, the direct interface is the AMPS MDB (Model Database).

AMPS MDB evolves from the data structure of YAP (Yet Another Parser) and CSV (Comma Separated Variables) storage scheme from the 90's. It was developed based on the need to have a portable database in either ASCII or binary format. It is designed to be multi-access thread-safe, and has very efficient and flexible storage expansion ability. It is also tailored specifically for finite element application due to the need for fast and efficient access.

Currently, the MDB contains three ASCII images in three separate files:

FE file: this is the file with extension ".fe". It contains the bulk FE data, viz. a header information, the finite element nodes and element connectivity information.
GEO file: this is the file with extension ".geo". It contains the geometric associations of the finite element discrete model to the solid model, and describes how elements and nodes represent solid volumes, surfaces, edges and vertices.
DAT file: this is the file with extension ".dat". It contains all the other pieces of information that is necessary to complete the model, viz. the finite element header, analysis controls, material definitions, boundary conditions, and various miscellaneous data needed for a successful analysis.

The concept of MDB is to eliminate the need for application programs to read and parse the information scattered around in these files. These files are merely an ASCII image of the MDB information, and they possibly will change into binary format for efficiency reason as the model size gets larger. The application program only needs to know how to extract the needed information from the MDB through a defined access function. As such, it will shield the application program from any changes or enhancements made in the AMPS system and will allow more flexibility in development.

The AMPS programs are developed with Microsoft Visual C++, version 6.0, SP 6. The following assumes that the user will use the same development system for using the MDB interface.

After this chapter, the reference manual is structured as follows: Chapter Two describes how to create AMPS MDB from FE, GEO and DAT files. Chapter Three steps into details of the general MDB access interface utility functions. Chapter Four itemizes each MDB database entry and the details of each item's usage. Chapter Five describes how to create an output file for AMPS AMPView to display the results. Finally, Appendix section illustrates the numbering and naming convention in element, face, side and other details used in AMPS MDB.

Chapter 2: Creating AMPS MDB

The MDB database does not exist at the beginning of the application program execution; instead, the user will create it by calling some MDB utilities. During the creation process, the utility routines parse the input files based on the key words of MDB. These key words are referred to as "verbs" and "nouns", and are used to construct the database entries, and they are listed in the header file "mdbverb.h" with the external names described in the function mdb_naming() in the supplied code mdb.cpp. We will step into more details of these verbs and nouns as we discuss them in the subsequent chapters.

The application developers should install the supplied the libraries and the utility routines' header in their preferred directory on the development system. Currently, only MSVC 6.0 SP6 compiled version is available, other versions will be available on request. All AMPS MDB library functions have been compiled using the multithread option since they have been designed as multi-access capable with internal thread-safe locking.

The file mdb.cpp is supplied as a reference as to how the low level input part of MDB works, and it is not necessary to use it, unless the developer desires to change the low level file access routines, or to rename some key words. Another file print.cpp defines all MDB output functions. Sometimes, it is easier for the application program to replace these generic console output functions and replace them with their own equivalent codes. This is necessary since most Windows's applications generally do not support generic console messaging routine, or the operation system re-directs certain ASCII console output.

Although MDB data format should be backward compatible with the earlier version, it is important to distinguish the MDB database version stored in the files as compared with the current MDB library version.. To access the data_version, include the mdb.h header file in the project. Couple key system data are also included in this "mdb.h" header since these data are usually the required for a MDB application project to allocate memory and problem dimension size related preparation. This information includes:

field_dimension : problem dimension
data_version : MDB data version
data_version_in : the MDB version used to created the MDB data files opened
data_title : the short description of the input MDB data file title.
data_file_name: the name of the current opened MDB ".dat" file name as stored in the MDB system

It is much easier to understand how to create MDB from the simple code below:

```
#include "stdafx.h"
#include <string.h>
#include <fstream.h>
#include "mdbverb.h"
#include "mdb.h"
#include "utils.h"
#include "print.h"

int main(int argc, char* argv[])
{
    long int i=0, fn_len=0, fn_ext=0, result=0;
    if (argc >= 2)
        strcpy( data_file, argv[1] );
    else {
        prints("Missing input data file specification !\n");
        return result;
    }

    try { //any MDB access error will throw an exception
        fname_ext(data_file_name, iobuffer, data_file);//call MDB util to get the filename and extension
        strcpy(data_file, data_file_name);
        strcat(data_file,".dat" ); //required to be *.dat

        if (mdb_input() > 0) { //now open MDB database
```

```
            //fe_solve(); //user program application
            result = 1;
            prints("Successfully finished!");
        }
    }
    catch (char *msg) {
        print_s("\nException Thrown...", msg);
    }

    mdb_exit(); //clean up and release mdb allocated resource, if any
    return result;
}

long int mdb_input()
{
    //return
    // 0 file read error
        // 1 success,
        //-1 fe or geo file read error
        //-2 data format error
        long int result = 0;
        if (!mdb_initization()) goto EXIT;
        if (!input_fe()) goto EXIT;
        if (!input_geo()) goto EXIT;
        return input_dat();
EXIT:
        return result;
}
```

The above example calls the MDB fname_ext() utility function to extract the filename of the input model. It then calls the mdb_input() function to read the FE, GEO and DAT files into the MDB database form for access. Finally, it shuts down the MDB by calling mdb_exit() so as to release used resource.

You must link the program with the proper ampsmdb.lib library pending on whether you are using the DLL version or the static link version.

Chapter 3: Accessing AMPS MDB

Once the MDB database is available, the application program has the direct access to all the stored information. All MDB integer data are stored as "long int" type (4 bytes), and all floating-point data are stored as "double" type (8 bytes).

Currently, accessing the MDB is done through the following routines (as detailed in the header file mdb.h)

double *mdb_double(long int data, long int index, long int &length);
double *mdb_mixed(long int data, long int index, long int &length);
long int *mdb_integer(long int data, long int index, long int &length);

The application program uses these routines to retrieve either double, mixed, or integer type data from MDB. It returns the pointer to the storage location of the data or data array. The variable **data** and **index** specify the MDB data number and the array index desired. The size of the data stored at the index location is returned in the variable **length**, if the data and index exist. Otherwise, it returns a NULL pointer with the length size specified as zero. Note that some data may not have indexed array storage, i.e., only one item stored. Also, for each data number, different index may have different data length.

Example:
Retrieve the nodal coordinates of a node:

```
double *coords, x,y=0.0,z=0.0;
long int node_index=1, length=0; // extract cords for node 1
coords = mdb_double(NODE,node_index,length);
x = coords[0];
if (field_dimension>1 && length>1) y = coords[1];
if (field_dimension>2 && length>2) z = coords[2];
```

long int mdb_string_number(char *str);

This routine is generally used when the application program wants to find out the data number associated with a certain string.

Example:

```
long int data_number=mdb_string_number("dispx");
```

char *mdb_data_name(long int data);

This routine is generally used when the application program wants to find out the associated name for a data number. It is usually unnecessary since the supplied function mdb_naming() contains all naming. Usually it is used to inform the user about a certain data number.

Example:

```
printf("The deformation result %s is %d", mdb_data_name(DISPX), 0.5);
```

long int mdb_data_type(long int data);

This routine is generally used to identify the data type associated with a data number. MDB uses three different data types:

INT_TYPE: Integer type,
DBL_TYPE: double type, and
MIXED_TYPE: mixed type.

The mixed type can store mixed integer and double data in the database array. When the mixed type data are stored, the integer data in the data array are converted into double before storing. For efficiency reason, the mixed type storage is generally not used for frequently accessed information.

This routine is usually not necessary since the application program usually already knows what data number they want to access, and already know the type from this manual.

Also, it is not unusual to store both integer and double type data to a DBL_TYPE data number for grouping convenience. Part of the reason is due to the backward compatibility issue in MDB version.

long int mdb_index_active(long int data, long int index);

The application program generally will use this function to check the existence of a data number with a specific index before accessing them. For example, some model may have non-contiguous nodes and element, so it is easier to code the element access as

```
for (long int elem=0;  elem<=mdb_index_max(ELEMENT);  elem++) {
    If (mdb_index_active(ELEMENT, elem) {
        //now use the element
        el_nodes = mdb_integer(ELEMENT,  elem, length);
    }
}
```

long int mdb_index_type(long int data);

The function mdb_index_type(data_number) returns the data type associated with "data_number". MDB database has three different types of storage modes, and they are:

INDEX_ARRAYDATA,
INDEX_DATA, and
INDEX_SWITCH.

The INDEX_ARRAYDATA indicates that an array data are stored at each index entry, and a sub-index can be used to retrieve the array data stored in each index entry. Many data in MDB are of this type, and the array data stored at each index could be of the same or different length. For example, ELEMENT is an INDEX_ARRAYDATA since for each ELEMENT index entry, it stores an array that contains the element connectivity nodes along with the element type.

The INDEX_DATA type tells the MDB that there is only one single entry without any index, but it may contain one or more data stored in that array, and can be addressed using array sub-index. The data array could have any size, but will be at least more than one. Data of this type are usually FE control data such as SYS_TIMESTEP that stores the beginning and the end of the solution time, and the step increment desired.

The INDEX_SWITCH type is a special database entry that requires no additional data storage (in some case, it may have one datum). This type of data is usually used to turn on/off an analysis option. For example, the existence of OPT_AXISYMMETRIC will signal the need for an axisymmetric analysis mode, and it has no actual data storage.

It is usually not necessary to use this function since the for data type of each data number can be found in this reference manual, and the application usually already knows what type of data is trying to extract from MDB.

For compatibility with lots of legacy Finite Element programs, the element and node INDEX_ARRAYDATA indexes should start from one. Other than these two special data (ELEMENT and NODE), all MDB data index could potentially start from index zero.

long int mdb_index_size(long int data, long int index);

This function returns the size of a single data index storage. For example,

```
nodes_in_elem = mdb_index_size(ELEMENT,100);
```

The example tells that ELEMENT index 100 contains a data storage of size 'nodes_in_elem'.

long int mdb_index_max(long int idat);

This function returns the maximum existing index associated with a data number. For example,

```
max_node_index = mdb_index_max(NODE);
```

It shows that the maximum index of NODE data number is 'max_node_index'.

long int is_mdbcoded(long int data);
long int mdb_data_number(long int data);
long int mdb_encode(long int data);

These three routines are used to identify whether an integer number is associated with a specific database number, or just a plain integer. The need for this can be demonstrated from the following thermal-stress DAT file entry:

```
material_type 0 continuum_field thermal_field
material_type 1 continuum_field
```

The above statements specify that material_type index zero contains two analysis fields (continuum_field and thermal_field) while material_type one contains only one continuum field. When they are read into MDB, the input routine created MATERIAL_TYPE index 0 and 1, and stored these two data arrays into the index storage. In the input data parsing process, "material_type" is considered as a "verb" since it will trigger a new storage associated with MATERIAL_TYPE. The alphanumeric input "continuum_field" and "thermal_field" are converted into integers using mdb_encode(CONTINUUM_FIELD) and mdb_encode(THERMAL_FIELD). The special encoded integer tells the user that these two integers are MDB data number, and can not be treated as regular integer. These two words are considered as nouns since they are part of a verb phrase and do not trigger any new MDB data index. An application program then retrieves this MATERIAL_TYPE integer data as follows:

```
long int *materialtype, length=0, n, field_type;
materialtype = mdb_integer(MATERIAL_TYPE,0,length);
for(n=0; n<length;n++) {
    if (is_mdbcoded(mateialtype[n]) {
        field_type = mdb_data_number(mateialtype[n]);
        if (field_type == CONTINUUM_FIELD) {
            // process stress continuum material
        } else if (field_type == FLOW_FIELD) {
            // process flow material type
        } else if (field_type == THERMAL_FIELD) {
            // process thermal material type
        }
    }
}
```

The application developer should carefully review each data number as documented in the reference manual to check whether any portion of the integer data contains this "encoded" MDB data number. Most of the MDB data numbers are fairly straightforward and do not need this conversion.

The internal identification to determine whether an integer is a MDB number or not is achieved by turning on the bit next to the sign bit of the integer, and the function is_mdbcoded(intdat) or mdb_data_number(intdat) basically checks this bit to determine whether it is an encoded MDB number or not, and recovers the proper data number for access.

Chapter 4: MDB Data Entry Reference

4.1 Naming Convention

In this chapter, we will be discussing all AMPS MDB data items in details. These items usually correspond to certain AMPS front-end user dialog box controls, menu item settings check boxes, etc. The MDB database is created from the DAT, GEO and FE files, and you can probably find the corresponding information in these files. After the AMPS MDB is created, the application program can retrieve the model data in any order, at any time. If, in the future, AMPS system changes the FE, DAT or GEO file into binary format for compactness, it should not affect the MDB data format as described in this chapter.

We will indicate whether a data number is an INT_TYPE, DBL_TYPE, or MIXED_TYPE by using (INT), (DBL), or (MIXED) symbols. This will only apply to data numbers that have actual storage, that is, they are of VERB type. If it is a NOUN type, they will not have actual storage, but will be used as a descriptive noun for other MDB data entry (e.g. DISPX, VELX). For NOUN type data numbers, since there will be no database storage, there will be no additional storage symbols attached.

For VERBs that are of INDEX_ARRAYDATA type, we will also append a parenthesis "[]" after the data type symbol. If they are either INDEX_DATA or INDEX_SWITCH type, then there will be no "[]" array symbol. For example, ELEMENT(INT[]) will identify the data number ELEMENT is to be accessed as an INT_TYPE index array.

For some MDB data entries, the data may contain encoded MDB nouns (so as to distinguish from regular integer information). For such entry, an '&' sign will be appended after the type symbol, and for the item that is encoded, we will also append the "&" symbol to it to remind the developer to decode it before using it. For example, BC_TYPE(INT&[]) means the data entry for BC_TYPE is an index array, and each array storage, accessible by the sub-index, may contain number that need to be decoded using mdb_data_number() entry. For details of this encoding function, please refer to Chapter Three.

Finally, to access the data, the application will use mdb_integer(datanumber,index,size) or mdb_double(datanumber,index,size) or mdb_mixed(datanumber,index,size) to retrieve the desired data stored at the index location. If the data entry is of type INDEX_DATA and there is no index associated, the retrieving index should be set to zero to avoid an MDB access error. If the returning point is NULL, or the array size returned is zero, that means there is no active data stored at the requested index location.

It is also important to check the MDB version (variable data_version as described in mdb.h header file) against the input data file MDB version (variable data_version_in). If the input version (data_version_in) is newer than the MDB library format (data_version), it's possible that it contains more MDB data items. In such case, the developer should get an update to the MDB library, or warn the customer that there is a possibility that some data may not be supported.

Whenever possible, an example of how to access and use the data entry is given to clarify the usage.

4.2 MDB Data Entry Description

Following are details of each MDB data number description. We have classified the MDB data entries into different categories for easy reference. The application program should also check the existence and the data length before using them because it is possible that the controls as specified do not exist as in some default cases.

Due to the enhancement in the AMPS system, these MDB data format sometimes will change. To maintain backward compatibility, the database version number 'data_version" can be used to distinguish the difference. The numerical superscript next to a MDB entry refers to the MDB data version when the data entry was first added after the original MDB database creation.

4.2.1 System Controls Data

SYS_ANALYSIS_TYPE (INT&) analysis_type&
This data number specifies the type of the analysis. Currently, the available analysis types are:
STATIC, DYNAMIC, MODAL, MODALSTIFFENED, MODALINSITU, MODALTHERMAL, BUCKLING, BUCKLINGSTIFFENED, BUCKLINGINSITU, FREQUENCYDOMAIN, EXPLICIT

STATIC
This specifies a static or a steady-state solution is desired.

DYNAMIC
This specifies an analysis by step-by-step integration in time.

Followings are the available eigen analysis types:

MODAL
This performs a standard modal analysis to extract the desired modes of vibrations. The generalized eigen problem is K -λM = 0, where K is the system stiffness and M is the consistent mass matrices.

MODALSTIFFENED Modal Vibrations Analysis with Stress Stiffening
This performs a standard modal analysis to extract the desired modes of vibrations. Before the modal analysis, a static analysis is performed and the stress stiffening effect is included for the modal analysis. The generalized eigen problem is (K+Ks) -lM = 0, where Ks is the additional stress stiffness matrix computed during the initial static analysis.

MODALTHERMAL Thermal Harmonic Modal
This calculates the thermal harmonic modes of temperature patterns. The generalized eigen problem is Kc -λMt = 0, where Kc is the system conductivity matrix and Mt is the consistent thermal mass matrix formed by scaling the consistent thermal mass matrix by the thermal heat capacity

BUCKLING Euler Instability Buckling
This calculates the linear Euler buckling modes of the system. The generalized eigen problem is K -λKs = 0, where K is the system stiffness and Ks is the stress stiffness matrix.

BUCKLINGSTIFFENED Linearized Instability Buckling with Geometric Stiffness
This is a linearized predicting of the nonlinear buckling load calculation. The initial geometric stiffness Kg is computed based on the given loading, then the generalized eigen problem of (K+Kg) -λ (Ks+ Ks ') = 0 is solved. Ks' is the 2nd order stress stiffness matrix based on the geometric deformation.

BUCKLINGINSITU In-Situ Eigen Analysis
After either a successful static or dynamic analysis, the user can specify to perform either a modal or an instability buckling analysis in the same analysis. Although it is possible to restart the problem and then specify an eigen analysis, this is a convenient method of performing "in-situ" eigen analysis. In such way, the "in-situ" stiffness and mass matrices (or the stiffness and the stress stiffness matrices) represent the state at the end of the analysis, and the eigen solution give the "in-situ" prediction of the eigen problem.

For the in-situ modal analysis, the generalized eigen problem is Kt -λM = 0, where Kt is the system tangent stiffness matrix. For the in-situ instability buckling analysis, the generalized eigen problem is Kt -λKs = 0, where Ks is the stress stiffness matrix based on the finished state.

FREQUENCYDOMAIN Dynamic Frequency Response/High Frequency Analysis
This specifies a frequency domain steady state solution is desired.

EXPLICIT Transient Explicit Dynamic Analysis
An explicit time integration method is used to compute the model response.

SYS_ANALYSIS_FIELD (INT&[]) physics1&, pphysics2&…
This data number specifies the physics involved in the multi-physics analysis model. It must contain all active fields (CONTINUUM_FIELD, THERMAL_FIELD, etc.) used in the analysis. The application program should use this data number to determine what type(s) of analysis is/are specified. The active fields of the analysis also determine the amount of field variables (DISPX, VELX, TEMP, etc.) that will be available in the BC_TYPE, PROP_MULTIPLIER, etc. Also, the actual material data for each field should be described at least once in the MATERIAL_TYPE data.

SYS_TIMESTEP (DBL) analysis_time_start analysis_time_end time1 step_size1 time2 step_size2…
This data number specifies the time/step stepping controls. The variable analysis_time_start indicates the time/step value the analysis starts (if this is a restart analysis, this value will be overwritten by the actual time/step value from the restart file). The variable analysis_time_end indicates the desired finishing time/step value. The analysis should start the time/step increment by interpolating the values stored in the subsequent pairs of data (time_n, step_size_n) that can be linearly interpolated. This is to achieve, in certain situation, a finer/coarser time/step increment at certain time/step value.

SYS_TIMESTEP_OUTPUT[1.76] (MIXED) time1 output_interval1 time2 output_interval2 …
This data number specifies the time/step result output interval controls. This is usually for long time/step analysis applications. In such cases, it is desirable to only output the analysis result only after certain time/step intervals. The variable 'time1' refers to the time/step value, and the integer value 'output_interval1' refers to the number of step/time that has to finish before an analysis will be written to the output file.

SYS_TIMESTEP_SOLVER (MIXED&) initial_solver_choice& equilbrium_solver_choice& iterative_solver_method& iter_solver_error_norm, drop_tol
This data number specifies the solver choices for the initial solution stage and the subsequent equilibrium iterations for each time/step marching. The available solver types are:
SOLVER_DIRECT: indicates sparse direct solver
SOLVER_SPARSE_ITERATIVE: indicates sparse iterative solver
SOLVER_FRONTAL: indicates frontal solver
SOLVER_ITERATIVE: indicates element-by-element iterative solver

The iterative_solver_method& could be one of the followings: **PCG, MLBICGSTAB, BICGSTAB, TFQMR and BGMRES.** Details of the difference of these methods can be found in the AMPS on-line reference manual.

The iter_solver_error_norm specifies the iterative solver's desired error_norm. It defaults to be 1.0e-7. Finally, drop_tol is a setting for the incomplete LU factorization by-pass value. If the value is zero, it is automatically determined based on the ratio of the minimum to the maximum diagonal of the equations.

SYS_TIMESTEP_CONTROLS (MIXED) error_norm max_iteration_limit
The variable error_norm is the desired error norm in each time/step solution. The variable max_iteration_limit is the allowed iteration counts during the equilibrium iterations in each time/step solution.

SYS_TIMESTEP_AUTOMATIC (DBL) minimum_time_step maximum_time_step
When the maximum iteration limit is reached, if this data number exists, the processor will automatically decrease the time step size and restarts the time/step solution from the position it failed. The variables minimum_time_step and maximum_time_step specify the minimum and the maximum step sizes this automatic stepping control can use. The automatic stepping will abort if the minimum_time_step size is reached, and will not take any larger time/step increment than the specified size.

SYS_ANALYSIS_EIGEN (INT) n_modes
This data numbers specifies the desired eigen modes in the eigen analysis.

OPT_AXISYMMETRIC (INT)

This data number is an INDEX_SWITCH type entry that specifies whether the analysis is an axisymmetric pseudo-three-dimensional analysis or not. The application program normally will use the function mdb_index_active(OPT_AXISYMMETRIC,0) to check its existence**.**

OPT_PLANESTRESS (INT)
This data number is an INDEX_SWITCH type entry that specifies whether the analysis is a plane stress analysis. If this switch does not exist for a two-dimensional analysis problem, it is assumed to be in the plane strain mode. The application program normally will use the function mdb_index_active(OPT_PLANESTRESS,0) to check its existence**.**

OPT_GEOMETRY_FORMULATION (INT&) formulation&
This data number is an INDEX_DATA type entry that specifies the nonlinear geometry formulations desired. The formulation could one of the followings:
LINEAR : this is the default small deformation analysis, if there is no specification.
TOTAL_LAGRANGIAN: large deformation with the total Lagrangian reference formulation.
UPDATED_LAGRANGIAN: large deformation with the updated Lagrangian reference formulation.

OPT_MULTITHREAD (INT) max_thread
This data number is an INDEX_DATA type entry that specifies the maximum parallel threads should be used instead of the default system configuration. When there are multiple CPU's, AMPS will automatically kick start parallel processing whenever possible (such as in equation solving, element processing, post-processing, etc). It is useful on machines that have more than one processor and the user wants to reserve some resource for other system processing.

OPT_RESTART (INT) restart_flag
This data number specifies whether to write an analysis restart file in case the user desires to continue the analysis later using the restart option. The default is to generate the restart file if the data number does not exist, or if the restart_flag is non-zero.

OPT_ADAPTIVE (MIXED&) control1& option1& value1 control2& option2& value2…
This data number specifies the adaptive refinement options. The 'control' variable could possibly be H_ADAPTIVE, P_ADAPTIVE, OPTIMIZE, or TARGET. If it is the word TARGET, it will be followed by the desired error_norm value for refinement. Otherwise it will be followed by the 'option&' variable of either AUTOMATIC, or UNIFORM. If it is the word UNIFORM, then it will be followed by the third integer to specify the n_division for the h_adpative method or the n_order for the p_adaptive choice. If the key word is OPTIMZE, then it means to renumber the refined mesh to optimize for solution speed, and there will be no more additional variable after it. For example, the control could possibly take one of the following forms:

H_ADAPTIVE AUTOMATIC P_ADAPTIVE AUTOMATIC TARGET 0.05 OPTIMIZE
This example specifies both h and p automatic refinements to achieve the error_norm to 0.05, and renumber the mesh after each refinement.

P_ADAPTIVE UNIFORM 2 TARGET 0.05
This example specifies uniform p refinement by converting all elements into the quadratic 2^{nd} order elements, and the target mesh refinement size is to minimize error_norm to 0.05.

H_ADAPTIVE UNIFORM 2 P_ADAPTIVE AUTOMATIC TARGET 0.05
This example indicates uniform h refinement by dividing all elements by a factor of 2 in each direction (essentially splitting) and uses automatic p_order calculation to achieve the target error_norm of 0.05.

OPT_ADAPTIVE_TIMESTEP (INT&[]) control1& value1 control2& value2…
This data number specifies the time/step when the user defined adaptive control should be executed. The possible control variables are:

MAXIMUM: This specifies the maximum allowable adaptive refinement. The maximum value desired should be specified in the integer value immediately next to the control word.

STEP: This specifies the step increment when the adaptive refinement should take place. The step increment value is the next integer value immediately next to this key word.
RESTART: This indicates to restart the analysis after the specified mesh refinement takes place, and repeat till the desired error norm is achieved. If this key word does not exist, the analysis should continue in time/step analysis and follow the solution control command. There is no additional value necessary after this key word.

OPT_ACCELERATION (DBL) accel-x accel-y accel-z
This data number specifies the global acceleration applied to the model in the Cartesian coordinate system. The data could possibly be shorter, depending on the setting of field_dimension value (declared in mdb.h) .

OPT_ACCELERATION_TIMETABLE (DBL) time1 value1 time2 value2…
This data number specifies how the global acceleration field is applied to the model. It uses a piece-wise linear interpolation table for the scaling of the time/step acceleration components. The linear interpolation is calculated from the pairs of the specified data.

OPT_ROTATINGFRAME[1.76] (MIXED) center-x, center-y center-z axis-x axis-y axis-z rotation_speed. opt_centrifugal opt_coriolis
This data number indicate the model should be formulated in a rotating frame attached to the origin at position (center-x, center-y, center-z), with a rotating axis along the vector (axis-x, axis-y, axis-z), and with a rotating speed of rotation_speed. The last two integer data indicate whether the centrifugal and the Coriolis acceleration effects should be included in the analysis or not (a zero term indicated the effect should be neglected).

OPT_ROTATINGFRAME_TIMETABLE[1.76] (DBL) time1 value1 time2 value2…
This data number specifies the time/step scaling factor for the rotation speed of rotating frame. It uses a piece-wise linear interpolation table for the scaling of the rotation speed. The linear interpolation is calculated from the pairs of the specified data.

UNIT_LENGTH_SCALING[2.5]**, UNIT_MASS_SCALING**[2.5]**, ,UNIT_TEMP_SCALING**[2.5]**, UNIT_TEMP_SCALING**[2.5] (DBL) scale offset
These are mainly for system unit conversions such as converting input nodal coordinate from centi-meter into nano-menter, etc. Each data item will contain two double values for scaling and offsetting. The unit data are converted using the following formula:
NewUnit = (*OldUnit**scale) + offset
For all data access, this unit conversion should be followed as the user's specified data conversion. Currently, in MDB version 2.5 and above, only **UNIT_LENGTH_SCALING** is active, and is mainly for the nodal coordinate transformation.

4.2.2 Elements, Nodes, and Geometry Information

ROD, BEAM, BEAM3, BEAM4, QUAD4, QUAD9, QUAD16, TRI3, TRI6, TRI10, TETRA4, TETRA10, TETRA20, HEXA8, HEXA27, HEXA64, SHELL3, SHELL6, SHELL10, SHELL4, SHELL9, SHELL16

ELEMENT (INT[]) type node1 node2 …
These are all the elements that AMPS currently supports. The element information is generated from the FE file. The MDB stores all element types in a single data number ELEMENT, and then stores the element type at the first entry, followed by the associated nodes of the element. The application program can use the mdb_index_size() function to identify the ELEMENT' length at any index. The MDB does not store individual element such as ROD, BEAM, TRI6, etc. Instead, they are all stored in a general ELEMNT entry for easy access.

Example:

```
long int el_name,*elnode, nodes, elem, size;
…
elnodes = mdb_integer(ELEMENT,elem,size);
el_name = elnodes[0];
nodes = elnode+1; //elnodes[1 to size-1] are the element nodes
if (el_name == QUAD4) //process QUAD4
else if (el_name == SHELL3) //process SHELL3
```

...

In the Appendix, we have listed the element node numbering scheme and the side and face numbering convention. The orientation of the material property and the beam/rod major axis orientation are specified in the material definition sections.

BODYELEMENT (INT[]) elem1 elem2 …
FACEELEMENT (INT[]) elem1 elem2 …
LINEELEMENT (INT[]) elem1 elem2 …
FACEELFACE (INT[]) elem1 side1 elem2 side2 …
EDGEELSIDE (INT[]) elem1 side_node_start1 side_node_end1 side1 elem2 side_node_start2 side_node_end2,side2…
FACENODE (INT[]) node1 node2 …
EDGENODE (INT[]) node1 node2 …
VERTEXNODE (INT[]) node
The above entries associate the finite element model to the solid model geometry. The AMPS AMPSolid solid modeling/meshing program generates this information and they are initially stored in the GEO file. These can be treated as a geometry entity grouping, where the grouping is described using the discrete finite element node, element, and side information.

BODYELEMENT tells how many solid elements in a BODYELEMENT indexed storage. FACEELEMENT is generally used in 2D or shell models, and it describes the amount of 2D or shell elements forming a geometry face. LINELEMENT, in a similar fashion, tells how many ROD, BEAM or one-dimensional elements forms a geometric edge/line.

FACEELFACE describes how a solid face is constructed from the solid element's face. It has pairs of entry in each data index, and they are element number and the element's side number forming the surface. For details of the solid elements' side numbering scheme, please refer to the description in the Appendix section.

EDGEELSIDE is used to describe how the geometry edge is formed from the finite element side. This could be the two-dimensional or three-dimensional elements' side, and they are described in the format of 4 numbers. The first number is the element number, the 2nd and the 3rd numbers are the two end nodes (in the global numbering sense) forming the element side, and the last number is the element's side number. If the side number is -1, that means the side sequence is unknown, but it should not happen if the file is generated by the AMPS system. For details of the 2D or 3D elements' side numbering, please refer to the description in the Appendix section. Note that for higher-order elements, there could be additional nodes between these two end nodes, and it is up to the application program to decide what to do with them. For example, if the user specifies a prescribed DISPX on the EDGEELSIDE, it generally means that all nodes along that side should be fixed.

BODYNODE (INT[]) node1 node2 …
This entry does not exist when the MDB is created (as of version 1.74). It is used internally for AMPS operation. The application program can construct this entry by grouping all nodes associated with all elements in a BODYELEMENT index. This is only needed if the application program desires to give the user the convenience of using the BODYELEMENT selection to imply all nodes in a body.

GROUPELEMENT (INT[]) elem1 elem2 …
GROUPNODE (INT[]) node1 node2 …
These two entries are for AMPView's element and node grouping display purpose only. The application program can use this for any desired operation. These two data entries are created using AMPView's element/node grouping feature.

GEOMETRY_POINT (DBL[]) pt_x pt_y pt_z pt_size
This data number is not used.

GEOMETRY_LINE (DBL[]) pt_x pt_y pt_z dir_x dir_y dir_z length side_flag

This data number specifies an analytical geometry line for the analysis. The geometry line is defined by using a point (pt_x, pt_y, pt_z) and a line direction vector (dir_x, dir_y, dir_z). The line has a length limit descriptor. If the geometry line is used in a 2D contact control or has time movement scaling, the side_flag could be: 0 specifies the left side and 1 for the right side if traversing from the starting point.

GEOMETRY_PLANE (DBL[]) pt_x pt_y pt_z normal_x normal_y normal_z side_flag
This data number specifies an analytical geometry plane for the analysis. The geometry plane is defined by a point (pt_x, pt_y, pt_z) and a normal direction vector (normal_x, normal_y, normal_z). If the geometry plane is used in a 3D contact control or has time movement scaling, the side_flag could be: 0 specifies the normal side and 1 for the reverse side.

4.2.3 Nodal Local Coordinate System

Currently, there are total of four different types of user defined nodal coordinate system options in the AMPS system. They are defined using the NODE_LOCAL_COORD and NODE_LOCAL_COORD_DATA MDB entries.
.

NODE_LOCAL_COORD(INT[]) geom_type1, id1 …, geom_type2, id2 …
This data number indicates to what part of the model the specified nodal local coordinate system should be created. The data entry format refers to the geometry type and index as discussed in the BC_GEOMETRY entry. For more information, please refer to BC_GEOMETRY entry description in the Boundary Conditions section.

NODE_LOCAL_COORD_DATA[MIXED[]) Type LHS_trans_option vec1-x vec1-y vec1-z vec2-x vec2-y vec2-z
This entry specifies the nodal local coordinate data. The integer data 'Type' indicates the type of the local coordinate system. If the local coordinate system refers to the local a, b and c orthogonal axes, then

Type=1: User specifies vector 1 as the a-axis and vector 2 as the b-axis. The application program should re-compute the b direction vector by orthogonalizing it with vector a, then compute the third local axis from the cross product of axes a and b,

Type=2: The application program should compute the specified geometry's surface normal (3-d) or side normal (2-d) as the local a-axis. Vector 1 is orthogonalized to the normal as the b-axis direction. Vector 2 is used if vector 1 is in the normal direction.

Type=3: Cylindrical coordinate system. Defined from the origin as specified by (vec1-x, vec1-y, vec1-z) and the cylinder axis direction as defined by (vec2-x, vec2-y, vec2-z). In such system the local a-axis is along the radial direction, and the b-axis is tangent to the radial in the rotational theta direction. The local c-axis is always along the specified z-axis direction.

Type=4: Spherical coordinate system. Defined from the origin as specified by (vec1-x, vec1-y, vec1-z) and the North Pole axis direction as defined by (vec2-x, vec2-y, vec2-z). The local a-axis is along the radial direction, and the b-axis is tangent to the radial in the rotational theta direction. The third tangential direction is the projection of the z-axis vector tangent to the local a-axis (radial) direction.

The LHS_trans_option indicates whether the transformation should be applied to the LHS equation system, or it is just used for the forcing terms RHS usage (when this option is of value zero).

4.2.4 Material Property Definitions

This section describes how the material properties associated with different physics are defined. In AMPS, most material properties are designed to be state dependent, so each definition could contain a prop_multiplier_data_index value. The prop_multiplier_data_index, if non-zero, indicates that the data value should be scaled using the property dependency multiplier specified using PROP_MULTIPLIER and PROP_MULTIPLIER_TABLE to be described later.

BODYMATERIAL (INT[]) material_id
FACEMATERIAL (INT[]) material_id
LINEMATERIAL (INT[]) material_id
ELEMENTMATERIAL (INT[]) material_id

These four data number specify the material_id to all elements in BODYELEMENT, FACELEMENT, LINELEMENT, and ELEMENT with the same index. The specified material type should then be described in the data number MATERIAL_TYPE and the associated property data entries.

ELEMENT_MATERIALTYPE (INT[]) element_material_type
When MDB is first created, each finite element has a default material index as specified in the FE file, and it is stored in this data entry. Usually, it is material index zero. However, during the pre-processing in AMPView, the user may specify different material indexes to different part of the model, and they are stored in the aforementioned BODYMATERIAL, FACEMATERIAL,LINEMATERIAL or even a direct ELEMENTMATERIAL assignment. For convenience, in the MDB utility routine, we have provided a routine mdb_materialdata_expand() that will take all these material index specifications and then update the ELEMENT_MATERIALTYPE to reflect the up to date material property assignment. If this mdb_materialdata_expand() is not called, the application program must carry the responsibility to check the additional BODYMATERIAL, FACEMATERIAL,LINEMATERIAL and ELEMENTMATERIAL assignment for each element. It is suggested that the user processing this mdb_materialdata_expand() once so the subsequence material index inquiry for each element can be directly retrieved from this data number.

MATERIAL_TYPE (INT&[]) physics1& physics2&…
This data number specifies the physics associated with a material type. It determines the different types of the physics in a material index specification. Currently, the available physics field types are:

CONTINUUM_FIELD, THERMAL_FIELD, FLOW_FIELD, ELECTRIC_FIELD, MAGNETIC_FIELD, USER_FIELD

If the material type specifies more than one field, the finite element differential equations should have the associated DOF's associated with each field, and the proper material properties should be specified using the same index**.** For example, to specify the material_type index 1 with Young's modules and thermal conductivity with nonlinear property, the DAT file will have the following:

```
material_type 1 continuum_field thermal field
prop_elastic_young 1 3.0e7 0
prop_thermal_conductivity 1 1.0e5 1
```

PROP_MULTIPLIER_DATA[2.6] (INT&[]) multiplier_index1, multiplier_index2…
This data number specifies the material property multiplier dependency index or indexes. Each index refers to the PROP_MULTIPLIER and the corresponding PROP_MULTIPLIER_TABLE for the scaling of the property. If the material data number using this PROP_MULTIPLIER_DATA has material data that are longer than the length of the multiplier index length, the last one is repeatedly used for as the multiplier index. Beginning with version 2.6, this PROP_MULTIPLIER_DATA is used for all material data that may have nonlinear dependency.

PROP_MULTIPLIER (INT&[]) variable_type& table_id
This data number specifies the material property multiplier dependency on the runtime variable variable_type&. The dependent variable could be any solution variable as in the BC_TYPE variables, but with additional TIME and SIGEQIV variables for time and equivalent stress dependency specification. The 2nd value table_id specifies the index of PROP_MULTIPLIER_TABLE as to how the actual multiplier should be computed from this runtime variable variable_type&.

PROP_MULTIPLIER_TABLE (DBL[]) n_reserved tablet_ype n_values value1 value2…
This is the multiplier data type and values specification. The first value n_reserved is not used for now. The 2nd value table_type (after converted into an integer) could be one of the following:
0: Piecewise linear interpolation. In such case, the next value n_values (after converted into an integer) specifies the amount of the rest of the data, and it should be of even number. The rest of the data come in pairs of (value1, data1), (value2, data2), etc for linear interpolation.

1: **Polynomial coefficients for polynomial order of zero to (n_values-1).** That is, the multiplier is computed from the polynomial expansion: multiplier=value1+value2*VAR+value3*VAR^2…, where VAR is the variable value as specified in the PROP_MULTIPLIER runtime variable.
2: **Exponential multiplier**. In this case, n_values=2, and there should be only two additional values stored afterward. The multiplier is computed as: multiplier=value1*(VAR**value2).

PROP_DAMPING (DBL[]) value prop_multiplier_data_index
This data number specifies the hysteric viscous mass damping coefficient. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_DENSITY (DBL[]) value prop_multiplier_data_index
This data number specifies the mass density of the material. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ELASTIC_POISSON (DBL[]) value prop_multiplier_data_index
This data number specifies the elastic Poisson's ratio of the material. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ELASTIC_YOUNG (DBL[]) value prop_multiplier_data_index
This data number specifies the elastic Young's modulus value of the material. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ELASTIC_ANISOTROPIC_YOUNG (DBL[]) young1 young2 young3 prop_multiplier_data_index
This data number specifies the elastic anisotropicYoung's modulus values in the specified material axis directions as specified in the PROP_ANISOTROPIC_AXES. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ELASTIC_ANISOTROPIC_POISSON (DBL[]) poisson12 poisson,13 poisson23 prop_multiplier_data_index
This data number specifies the elastic anisotropic Poisson's values in the specified material axis directions as specified in the PROP_ANISOTROPIC_AXES. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ELASTIC_ANISOTROPIC_SHEAR(DBL[]) g12 g23 g13 prop_multiplier_data_index
This data number specifies the elastic anisotropic shear modulus values in the specified material axis directions as specified in the PROP_ANISOTROPIC_AXES. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ELASTIC_ANISOTROPIC_EXPANSION (DBL[]) beta1 beta2 beta3 prop_multiplier_data_index
This data number specifies the elastic anisotropic thermal expansion coefficient values in the specified material axis directions as specified in the PROP_ANISOTROPIC_AXES. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_ANISOTROPIC_AXES (DBL[]) a-x a-y a-z b-x b-y b-z
This data number specifies the local material axis **a** and **b** direction. The 3rd material axis is computed by taking the cross product of vectors **a** and **b**. The application program can reliably assume that the direction vectors are unit vectors, and they are orthogonal to each other if the data is generated by AMPS.

For shell material in the 3-dimensional space, the vector **a** direction as entered will be projected to the shell surface to indicate the shell orthotropic major axis direction. If the **a** axis projection is zero (e.g., happen to be in the shell's normal direction), the 2nd axis **b** is then used to project to the shell and to indicate the 2nd orthotropic direction. In such case, the major axis direction can then be determined from the shell's normal vector and the 2nd axis.

PROP_EXPANSION_LINEAR (DBL[]) value prop_multiplier_data_index
This data number specifies the elastic thermal expansion coefficient. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_EXPANSION_REFERENCE (DBL[]) value prop_multiplier_data_index
This data number specifies the elastic thermal expansion reference temperature. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_HYPER_MOONEY_RIVLIN (DBL[]) c1 c2 prop_multiplier_data_index
This data number specifies the hyperelastic Mooney strain energy function c1 and c2 coefficients controlling the first and the second strain invariant. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_PLANE_STRESS (DBL[]) thickness
This data number only applies to either the plane stress or the shell material model. It specifies the thickness of the material type of the indicated index.

PROP_PLASTIC_VONMISES (DBL[]) plastic_yield_stress prop_multiplier_data_index
This data number specifies the initial yield stress for the von Mises plastic material model. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_PLASTIC_HARDENINGMODULUS (DBL[]) h-module prop_multiplier_data_index
This data number specifies the isotropic hardening modulus. This modulus can be computed from the uni-axial test result E and Et. The tangent modulus Et is the apparent slope of the stress-strain curve after the initial yield, and E is the initial Young's modulus. The hardening modulus H is defined as

$H = Et/(1-Et/E)$

PROP_PLASTIC_HEAT_CONVERSION (DBL[]) strain-to-thermal prop_multiplier_data_index
This data number specifies the strain energy to thermal energy conversion constant for thermal elastic plastic energy transform calculations. Multiplier scaling should be used if prop_multiplier_data_index is nonzero

PROP_PLASTIC_KINEMATIC_HARDENING (DBL[]) b prop_multiplier_data_index
This data number specifies the isotropic kinematic Bauschinger hardening effect in the cyclic loading behavior. The effective yield stress is computed from $Sy = Sy0 + (1-b) H \varepsilon p$, where Sy0 is the initial yield stress, εp is the accumulated effective plastic strain computed, and H is the isotropic hardening modulus. When b=0, the kinematic hardening effect is excluded, and when b=1, the full kinematic effect is applied. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_PLASTIC_MOHRCOULOMB (DBL[]) cohesion angle
This data specifies the Mohr-Coulomb cohesive material with cohesion and the friction angle. The friction angle should be between 0 and 90 degrees, and should be in radian unit.

PROP_THERMAL_CAPACITY (DBL[]) Cp prop_multiplier_data_index
This data number specifies the thermal heat capacity of the material index. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_THERMAL_CONDUCTIVITY (DBL[]) conductivity prop_multiplier_data_index
This data number specifies the thermal conductivity of the material index. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_THERMAL_DENSITY (DBL[]) rho prop_multiplier_data_index
This data number specifies the thermal density of the material index. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_VISCOSITY (DBL[]) viscosity prop_multiplier_data_index
This data number specifies the viscosity of the material index, and is usually applied to the flow field. Multiplier scaling should be used if prop_multiplier_data_index is nonzero.

PROP_SECTION_DIR (DBL[]) b-x b-y b-z b1-x b1-y b1-z

This data number specifies the beam/rod local b-axis direction with a 2nd backup b1 direction vector. The beam/rod uses local axes a, b and c system to identify the orientation of the section. The a-axis direction is along the beam/road axial direction. The user enters the local b-axis direction through the local b-vector component. An additional b1-axis vector specification is used is the b-axis vector specified happens to be coincident with the beam/rod member axis.

PROP_SECTION (MIXED[]) section_type value1 value2 …
This data number specifies four different section types of beam/rod cross sections. The section_type should be:
1: User defined sectional properties: the values define the pre-computed user data of the cross section area, the torsional rigidity, and the moment of inertia about the local **a** and **b** axes. If the last value is nonzero, it is the shape factor of the section for shear stress correction.
2: Rectangular section: the values define the width and the height of the cross section following by the shape factor value. If the last value is nonzero, it is the shape factor of the section for shear stress correction.
3: Circular section: the values define the diameter of the cross section and the shear shape factor if shear stress correction for deep beam is desired.
4: Tubular section: the values define outer and the inner diameters of the cross section and the shear shape factor if shear stress correction for deep beam is desired

For rectangular cross-section, the term 'width' refers to the dimension in the local c-axis direction while 'height' refers to the dimension in the local b-axis direction.

PROP_SECTION_LOADS (MIXED[]) load_type value1 value2 …bc_time_index
Other than the regular boundary condition controls, the beam/rod element has the additional member loading ability by using this data entry. The load_type variable should be:
0: Initial Tension and/or Tension/Compression Only: this setting is only applicable to rod element. After the load_type integer variable, the 2nd value is the "rope-mode" integer indicator. A non-zero rope-mode value specifies that the rod element can only take tension only, as a rope. The 3rd value is the initial tension or compression stress value, the 4th value is the initial slack that is required for the tension/compression action to engage. This is usually used in an open gap contact situation. The last value in the integer index to the BC_TIMETABLE index is used to indicate the load multiplier in time/step.
1: Uniformly Distributed Local Axis Loads: the 2nd, 3rd and the 4th value of the data number specify a uniformly distributed load along the local a, b and c direction. The total force should be calculated from the integration of these uniformly distributed loads after axes transformation. The last value in the integer index to the BC_TIMETABLE index is used to indicate the load multiplier in time/step.

PROP_INTEGRATION_TYPE (INT&[]) integration_type&
This data number specifies the element formulation integration type used in the corresponding MATERIAL_TYPE index. The following element integration types are possible:
INTGAUSS: Use the Gaussian quadrature rule
INTREDUCED: Use the reduced Gaussian quadrature rule
INTLOBATTO: Use the Lobatto quadrature rule
DEFAULT: Use the element default quadrature rule

PROP_USER_PLASTICITY (DBL[]) usr_data1, usr_data2 …
When a material index is associated with the PROP_USER_PLASTICITY, it indicates that user is using their own plasticity material model. The data associated with the user's material model are stored in each index, and will be pass into the user's USRAPP.DLL function usrprop_plasticity(). For details of this, please refer to the "AMPS User Application Interface Reference Manual".

PROP_USER_INITDATA (DBL[]) usr_data1, usr_data2 …
When a material index is associated with the PROP_USER_INITDATA, it indicates that user is using their own plasticity material model (see PROP_USER_PLASTICITY), and the initial values of the user plasticity history data can be initiated in this MDB control. At the beginning of the analysis for each element integration point, the user routine usrprop_init() will be call so they can decide what type of special data initiations are needed.

4.2.5 Boundary Conditions

AMPS system uses a "when, where, why, how" approach to specify the boundary conditions, and tries to minimize the repeated definitions by grouping the BC into different predefined sets. To understand the boundary condition setting, it is necessary to describe how AMPS system uses a BC_COMBINATION_SET to define the BC.

BC_COMBINED_SET (INT[]) geometry_id type_id type_variable_id type_values_id time_table_id
The BC is defined from a combination of predefined data index. The first entry in a BC_COMBINED_SET specifies the geometry_set_id (where the BC is applied). The 2nd number refers to the BC_TYPE id (what type of BC). The 3rd and the 4th numbers specify what variables (e.g. DISPX, VELX, etc), and the specific values applied. The last number refers to the time/step multiplier index.

BC_GEOMETRY(INT[]) geom_type1, id1 …, geom_type2, id2 …
This data entry contains one or more geometry description sets. Each geometry description set starts with a geometry type ID, then followed by the necessary number to describe the geometry entity. Depending on the geometry type, the number following the type ID could be different length. This geometry description then repeats till all desired geometry items are finished. Some of the ID's are the index to the corresponding geometry data number (such as BODYELEMENT, FACEELFACE, etc.). Currently, Other than **Face** and **Side** geometry types, all other geometry types contain only one index.

The possible geometry types are described in the header file geomtype.h, and they are actually encapsulated as an enumerator variable 'ShellSelectType'. Their corresponding geometry specification descriptors are:

Geometry Type	**Descriptor(s)**
Cell,	elem_id
Face,	elem_id face_side_id (solid element only)
Side,	elem_id side_node_start side_node_end side_no
Node,	node_id
BodyElement,	bodyelement_index
FaceElement,	faceelement_index
FaceNode,	faceode_index
FaceElFace,	faceelface index (solid element only)
EdgeNode,	edgenode_index
EdgeElSide,	edgeleside index
VertexNode,	vertexnode_index
LineElement,	lineelement_index
LineNode,	(reserved)
SideEl,	(reserved)
FaceEl,	(reserved)
Any,	(reserved)
FEAny16,	(reserved)
FEAny17,	(reserved)
FEAny18,	(reserved)
FEAny19,	(reserved)
FEBoundary,	(reserved)
GeomPoint,	geompoint_index
GeomLine,	geomline_index
GeomCircle,	(reserved)
GeomPlane,	geomplane_index
GeomSphere,	(reserved)
GeomCylinder,	(reserved)

Example:

The following example shows how to apply nodal BC from the specified geometry set.

```
#include "geomtype.h"

ShellSelectType geomtype;
long in, index=0, length, geomval = mdb_integer(BC_GEOMETRY,index,length);

for (n=0;n<length;) {
     geomtype = (ShellSelectType) geomval[n];
     switch (geomtype) {
          case Node: //process single node
                    inod = geomval[n+1];
                    …
                    n+=2;
                    break;
          case Cell: //process element
                    elem = geomval[n+1];
                    …
                    n+=2;
                    break;
          case BodyElement: //process bodyelement nodes
                    in = geomval[n+1];
                    …
                    n+=2;
                    break;
          case Face: //process 3D element face nodes
                    in = geomval[n+1];
                    …
                    n+=3;
                    break;
          case Side: //process element side nodes
                    for (in=2;in<4;in++) {
                         inod = geomval[n+in];
                         …
                    }
                    n+=5;
                    break;
          case FaceElement: //process 2D or shell nodes in an FaceElement elements
          case FaceElFace:  //process 3D solid element face nodes
          case FaceNode: //process 3D nodes on a single element
                    in = geomval[n+1];
                    … //process all nodes on a face- could be from facenode or faceElement or FaceElface
                    n+=2;
                    break;
          case EdgeElSide: //process nodes on EdgeElSide
          case EdgeNode: //process single element side nodes
                    in = geomval[n+1];
                    … //let's treat it if the corresponding EDGENODE is there
                    n+=2;
                    break;
          case VertexNode: //process single geometry vertex node
                    in = geomval[n+1];
                    …
                    n+=2;
                    break;
          default:
                    n++;
                    prints("Warning - skipping unknown nodal BC geometry item!\n");
                    break;
     }
}
```

BC_TYPE (INT&[]) type& local_coord_option&[1.76]
This data number specifies the type of boundary conditions. Note that the variable is encoded, so the application program must retrieve the actual bc_type variable as follows:

```
long int bc_type,local=0, index=0, length, *bc_types = mdb_integer(BC_TYPE,index,length);
bc_type = mdb_data_number(bc_types[0]);
```

```
if (bc_type ==BC_FORCE) …//process BC_FORCE
else if (bc_type == BC_PRESSURE) …//process BC_PRESSURE
…
if (length>1) {
  local = mdb_data_number(bc_type[1]);
     if (local == LOCAL) {
          //process local coordinate system direction
          …
     }
}
…
```

The optional variable local_coord_option, if exist, and the value is LOCAL, then the boundary condition should refer to the nodal local coordinate direction (if applicable) as defined by the entries NODE_LOCAL_COORD and NODE_LOCAL_COORD_DATA. For example, if BC_FORCE is the "type" value, and "local_coord_option" is LOCAL, then the boundary condition indicates that the force as specified in the BC_VALUE entries refers to the local coordinate system direction for nodes that have the local coordinate system defined. For more details information about the nodal local coordinate system, please refer to the section "Nodal Local Coordinate System."

The available types are list below:

BC_FORCE, BC_PRESSURE, BC_FLUX, BC_CONVECTION, BC_RADIATION, BC_NODEAPPLIED, BC_NODEFIXED, BC_NODESTIFFNESS, BC_NODEMASS, BC_NODEDAMPING, BC_CONTACT, BC_TIE, BC_CONSTRAINT

These are the available boundary condition types currently available in AMPS. They are nouns used in BC_TYPE for boundary condition identification. The difference between BC_FORCE and BC_PRESSURE is that BC_FORCE is an applied pressure with global vector components (and the component is specified in BC_VARIABLES data number) while BC_PRESSURE always implies normal pressure (positive as away from the surface). BC_FLUX is similar to the BC_PRESSURE, except it is for heat/electric flux. BC_CONVECTION and BC_RADIATION are purely used in thermal natural boundary condition. BC_NODEAPPLIED refer to either nodal force/heatlfow/electric current RHS item. BC_NODEFIXED is used to specify the LHS unknown, and could be depend on the BC_VARIABLE type the BC is referring to. BC_CONTACT/BC_TIE specify the contact and tie/glue boundary condition. BC_CONSTRAINT is the constraint equation that specifies a constraint equation of several nodes and DOF's.

BC_TYPE_VARIABLES (INT&[]) var1& var2&…
This data number specifies the unknown variables specified by the user to be used in the BC. These variables could be one of the MDB nouns (but not limited to them as the AMPS features expand) below:

DISPX, DISPY, DISPZ, ROTX, ROTY, ROTZ, VELX, VELY, VELZ, PRESSURE
VORT1, VORT2, VORT3, TEMP, TFLUXX, TFLUXY, TFLUXZ, VOLTAGE, DENSITY, ALL,

The application program should determine the meaning of these variables and apply the corresponding value in the BC_TYPE_VALUES to the DOF associated with the BC. For example, a prescribed deformation in DISPX, DISPY, or DISPZ directions, the boundary condition type will be BC_FIXED, and the application program will do the following checks:

```
long int bc_var, index=0, length, *bc_vars = mdb_integer(BC_TYPE_VARIABLES,index,length);
for (int n=0;n<length;n++) {
     bc_var = mdb_data_number(bc_vars[n]);
     if (bc_var ==DISPX) …//process prescribed deformation in global X direction
     else if (bc_var ==DISPY) …//process prescribed deformation in global Y direction
     else if (bc_var ==DISPZ) …//process prescribed deformation in global Z direction
     else …
}
```

Note that TEMP refers to temperature. VORT1, VORT2 and VORT3 are only used in AMPS fluid LSFEA formulation that requires vorticity variables. TFLEXX, TFLUXY and TLFUXZ are thermal flux variables. The last

variable ALL refers to some BC_TYPE's (such as BC_TIE) that use this noun to indicate all active runtime variables.

BC_TYPE_VALUES (DBL[]) value1 value2 …
This data number stores the user-specified values associated with the variables in the BC_TYPE_VARIABLES of the same index. For example, if the BC_TYPE is BC_NODEAPPLIED and BC_TYPE_VARIABLES contains two data DISPX and DISPZ, then the BC_TYPE_VALUES should have two values corresponding to the applied nodal values corresponding to the X and the Z directions.

Sometimes, the BC_TYPE setting will govern the BC_TYPE_VALUES, and the BC_TYPE_VARIABLES has no specific meaning. For example, when BC_TYPE is BC_PRESSURE, then only the first BC_TYPE_VALUES is meaningful since pressure is a scalar variable with the direction already specified in the surface normal direction.

Several BC_TYPE conditions have values that are not associated with the BC_TYPE_VARIABLES, but have special meaning. The exceptions are:

BC_RADIATION, BC_CONVECTION
When the BC_TYPE is either one of the above, the corresponding BC_TYPE_VALUES index will contains up to 6 values. The first two values are the convection coefficient and the ambient temperature, and the last four values are the Stefan_boltzman constant, the surface emissivity coefficient, the radiation ambient temperature and the absolute reference temperature respectively. The application should only use the radiation or the convection part of the data, pending on the BC_TYPE setting.

BC_TIE
For the tie/glue boundary condition, the BC_TYPE_VALUES will contain three values:
Value_1: bit 1 of this value indicates that this geometry tie constraint is time dependent. When bit 2 is active, it specifies that the tie factor specified in the later part of the BC_TYPE_VALUES should be used, instead of the automatic geometric tie factor.
Value_2: the master geometry index for the TIE condition. The default value is zero. When TIE condition is specified, the master geometry contains the independent variable, and the rest of the geometry items as specified in BC_GEOMETRY are constrained to this master geometry. The integer value Value_2 specifies which of the BC_GEOMETRY is the master geometry.
Value_3: this is the user specified tie factor. The value is used only when value_1 bit 2 is active. Normally, when a tie BC is specified, the application will have to determine the geometry relationship and apply proper tie factor to the corresponding equations.

BC_CONTACT
For surface/side contact boundary condition, the BC_TYPE_VALUES will contain possibly up to six values:
Value_1: When bit 3 of this value is active, it indicates the contact BC applies to the continuum_field or fluid_field. When bit 4 is active, it specifies the contact BC applies to the thermal_field. When the contact geometry is an analytical type, such as line, plane, etc., bit 5 specifies the contact geometry has a time dependent movement as indicated in the Value_6 position BC_TIME index. Bits 1 and 2 are reserved for internal used.
Value_2: the penalty scaling factor for the continuum/fluid contact. The default value is zero (i.e. no scaling). This scaling is applied to the internally computed automatic penalty weighting value. A scaling factor less than one will improve the dynamic equilibrium iteration speed, but at the sacrifice of the slight contact penetration, while a larger scaling factor will tighten up the contact penetration, but could lead to more iterations in solution and possibly divergence behavior.
Value_3: the kinematic Coulomb friction coefficient for the continuum contact.
Value_4: the penalty value for the thermal contact. The default value is zero.
Value_5: the frictional energy to thermal energy conversion constant (in heat transfer with stress contact analysis)
Value_6: the BC_TIME index to use if the contact geometry is analytical (plane, line, etc), and the Value_1 bit 5 is active.

BC_CONSTRAINT[1.76]
When the BC_TYPE is BC_CONSTRAINT, the BC_TYPE_VARIABLES and BC_TYPE_VALUES entries will have the data format as below

BC_TYPE_VARIABLES GeomINDX1 var1& GeomINDX2 var2&…
BC_TYPE_VALUES use_time_control dependent_geom value_1 value_2…const_h

The constraint refers to the constraint equation of Sum(NiDOFi*Ci)+h=0.0, where Ni is the node number as indicated by the GeomINDXi (refer to the sequence count in BC_GEOMETRY). DOFi refers to the variable of the node, and the constant term h is the non-homogeneous term of the constraint. Ci is the constraint equation coefficient of each DOF, and is specified in the BC_TYPE_VAUES value_i position. The constant term h is specified as the last entry of the BC_TYPE_VALUE. If use_time_control is a non-zero number, then it indicates that the constraint equation is subjected to the time_index scaling of the BC_COMBINED_SET set. When the time_index sclaling is less or equal to zero, the constraint equation should not be applied. The constraint equation is constructed by assuming the first DOF can be expressed as the dependent variable and can be expressed in terms of the rest of the equation variables. By default, the first DOF is the default dependent variable. The dependent_geom index refers to this dependent variable expressed in terms of the sequence count of the selected geometry nodes.

BC_TIMETABLE (INT[]) time1 multiplier1 time2 multiplier2 …
This is the piece-wise linear time scaling lookup table to scale up the initial values stored in the BC_TYPE_VALUES. If the data is generated by AMPS, the time values should be increasing. If time1=time2=0.0, then it specifies an initial boundary condition for the BC_COMBINED_SET, and should be neglected in the subsequent time step analysis.

BC_ARCLEN (MIXED[]) Arc_length_method Load_increment_max Load_decrease_min Desired_defrom_increment Max_deform_increment Min_deform_increment
This is the nonlinear softening/post-buckling arc_length control. If BC_ARCLEN data index exit for a specific BC_TYPE index, then it means the arc_length load stepping control is desired for this BC set. The application program can use mdb_index_active(BC_ARCLEN,index) to check the existence of this setting. Usually, arc_length control applies only to RHS quantities (pressure, flux, force, nodal loads, etc). The controls are as follows:

Arc_length_method: 0 Riks' method 1 Crisfield's method
Load_increment_max: maximum load increment after a successful arc_length stepping (default = 3.0)
Load_decrease_min: minimum load decrease after a unsuccessful arc_length stepping (default = 0.001)
Desired_deform_increment: desired deformation increment (0 for computed from initial loading)
Max_deform_increment: maximum deformation increment size (0 for 5 times the current step size)
Min_deform_increment: minimum deformation increment size (0 for 0.001 times the current step size)

BC_NODE_APPLIED (DBL[]) value1 value2 …
BC_NODE_FIXED (DBL[]) value1 value2 …
NODE_DAMPING (DBL[]) value1 value2 …
NODE_STIFFNESS (DBL[]) value1 value2 …
NODE_MASS (DBL[]) value1 value2 …
These five data are no longer used in AMPView/AMPSol. They were used to support direct nodal input specifications.

Finally, for general reference, followings are the possible solution variables used in AMPS analysis:

DISPX, ISPY, DISPZ, ROTX, ROTY, ROTZ, VELX, VELY, VELZ, PRESSURE
VORT1, VORT2, VORT3, TEMP, TFLUXX, TFLUXY, TFLUXZ, DENSITY, SIGXX, SIGXY, SIGXZ, SIGYY, SIGYZ, SIGZZ, SIGEQUIV, YIELDF, EPSPLASTIC, TOTEPSXX, TOTEPSXY, TOTEPSXZ, TOTEPSYY, TOTEPSYZ, TOTEPSZZ, VOLTAGE, ECHARGE, EINTENSITYX, EINTENSITYY,EINTENSITYZ, ECURRX, ECURRY, ECURRZ, DAMAGE, VIODRATIO, USERVARIABLES

Chapter 5: Creating Result File for AMPS Post-Processing

When an application has finished processing the model, it may generate a result file for AMPS AMPView to read in for post-processing. The analysis program should write the results in a file with the file name extension ".out" for a standard analysis, or in a file with an extension ".eig" for an eigen value analysis such as buckling, modal analysis. For models with shell, beam and/or rod elements, AMPS supports specialized element result display such as moment diagrams etc., and the output file can be written into a file with extension ".eou". The file name should be, in general, the same as the model file name since AMPS will try to detect these result files (*.out, *.eig and *.eou files) and will turn on/off certain post-processing result display options in AMPView.

AMPS is designed to support general linear/nonlinear FE applications. It reads the output results in steps and tries to display them whenever possible. It is very common for the user to read the partially finished analysis results while the analysis program is still running. The analysis program should avoid locking the output file for exclusive access, and should try to flush the result file buffer to reflect the most recent update.

5.1 Creating Nodal-Based Results

The application program should always try to generate the results based on the node since this is the most useful information and there are more post-processing tools for this type of nodal based information. For finite element analysis, usually there are already interpolation functions available to project the results to the nodal positions using local extrapolation/smoothing function, or processing global interpolation/smoothing to obtain the nodal-based results.

The nodal output file format should be as follows:

data_version version_number
data_title output_result_title
node_result label1 label2 …
time/step timestep_or_eigen_value_0
node1, value1 value2…
node2, value1 value2…
…
time/step timestep_or_eigen_value_1
node1, value1 value2…
node2, value1 value2…
…
Several special output variable labels are worth mentioning. If the output variable labels contain the words "dispx", "dispy", or "dispz", they are automatically picked up by the AMPS AMPView as deformation. In such a case, menu items specific to deformation animation, deformation size, etc. will be enabled. If the output variable labels contain the word "velx", "vely", or "velz", then the vector display menu will be activated to display velocity vector as an arrow. Also, for the current AMPView design, it is required to use the label "time/step" as the step results separator.

There is no limit of items for result processing, as long as each active node in the model has the corresponding output results, and the nodal results match the output variable labels. For each result set, the time/step value (or the eigen value for the eigen result) will be used to identify the result set in AMPView display. If there are more than one result set, AMPView will enable the multiple time/step animation ability and the corresponding menu (such as history plot).

The following example demonstrates how to write the output files for both standard results and eigen results:

```
#include <stdio.h>
#include <string.h>
#include <fstream.h>
```

```
#include "mdbverb.h"
#include "mdb.h"

extern double time_step, *dispx *dispy, dispz, *temp_result, *eig; //results for disp, temp, and eigen values
long int disp[] = {DISPX,DISPY,DISPZ};
char outfilename[MAX_CHAR],eig_outfilename[MAX_CHAR];

void output_result(void)  // output standard result file
{
     long int node, i, maxnode;

     SetModelFileName(outfilename, ".out");
     ofstream out;
     out.open(outfilename);

     out << "data_version " << data_version << "\n";
     out << "data_title " << data_title << "\n";
     out << "node_result";
     if (field_displacement) {
          for (i=0;i<field_dimension;i++) out << " " << mdb_data_name(disp[i]);
     }
     if (field_temperature) {
          out << " " << mdb_data_name(TEMP);
     }
     out << "\n";
     out << "time/step " << time_step << "\n";

     maxnode=mdb_index_max(NODE);
     for (node=0;node<=maxnode;node++) { //output the actual result
          if (mdb_index_active(NODE,node)) {
               out << node;
               if (has_displacement) {
                         out << " " << dispx[node];
                         if (field_dimension>1) out << " " << dispy[node];
                         if (field_dimension>2) out << " " << dispz[node];
               }
               if (has_temperature) {
                         out << " " << temp_result[node];
               }
               out << "\n";
          }
            }
     out.close();
}

void output_eig_result(void) // output standard result file
{
     long int node, i, modes, maxnode;

     SetModelFileName(eig_outfilename, ".eig");
     ofstream out;
     out.open(eig_outfilename);

     out << "data_version " << data_version << "\n";
     out << "data_title " << data_title << "\n";
     out << "node_result";
     if (field_displacement) {
          for (i=0;i<field_dimension;i++) out << " " << mdb_data_name(disp[i]);
     }
     if (field_temperature) {
                    out << " " << mdb_data_name(TEMP);
               }
     out << "\n";

     maxnode = mdb_index_max(NODE);
     modes = *mdb_integer(SYS_ANALYSIS_EIGEN,0,i);
     for (mode=0;mode<=modes;mode++) {
          out << "time/step " << eig[mode] << "\n";
          for (offset=0,node=0;node<=maxnode;node++,offset+=maxnode+1) {
               if (mdb_index_active(NODE,node)) {
```

```
                    out << node;
                    if (has_displacement) {
                        out << " " << dispx[node+offset];
                        if (field_dimension>1) out << " " << dispy[node+offset];
                        if (field_dimension>2) out << " " << dispz[node+offset];
                    }
                    if (has_temperature) {
                        out << " " << temp_result[node+offset];
                    }
                    out << "\n";
                }
            }
        }
        out.close();
}
```

5.2 Creating Element-Based Results

If a model contains any rod, beam or shell elements, element-based special results such as moment or shear force display are possible. The feature is designed due to the need to display results that are discontinuous across the elements. The file name for the element-based result file should always has the extension ".eou".

The element-based result output file format should be as follows:

data_version version_number
data_title eleement_output_result_title
element_result_rod label1 label2 … n_rod_labels
element_result_beam label1 label2 … n_beam_labels
element_result_shell label1 label2 … n_shell_labels
time/step timestep_0
elem1 ngeom nlayer n_results
value1 value2 … n_value
… (output n_result values for ngeom*nlayer times)
elem2 ngeom nlayer n_results
value1, value2…
…
time/step timestep_1
elem1 ngeom nlayer n_results
value1, value2…n_value
…

The label **element_result_rod**, **element_result_beam** or **element_result_shell** should only be here if there is such corresponding elements. Also, the data_version must be at least 1.6 or above since the earlier MDB versions use a little different format. We suggest the application program use the MDB variable "data_version" when writing this version number.

For each element output, there are three additional variables needed to be included: ngeom, nlayer, and n_results. For rod elements, the 'ngeom' variable must be one, and it must be two for beam elements. For shell elements, the 'ngeom' variable should be corresponding to the amount of corner nodes. For example, for SHELL4 and SHELL9 elements, it must be 4, and should be 3 for SHELL3 or SHELL6 elements. The 'ngeom' variable refers to the number of geometric corner location in an element. The rod is treated as one-geometry element since it has only one element axial direction result. The beam element has two end nodes, and the shell elements have 'ngeom' corner positions. Note that when outputting the results for shell element, it must follows a counterclockwise sequence for the corner nodes. For example, for SHELL4 element, the sequence should be 1, 2, 4 and 3. It's the application program's responsibility to produce meaningful result corresponding to these 'ngeom" locations.

The variable 'nlayer' tells AMPS how many layers in an element that can be displayed. Currently, for both rod and beam element, it must be one. For shell element, it must be two to indicate the back and the front faces of the shell

element. It is also the application program's responsibility to make sure that, for each element, the variable n_result matches the corresponding amount of variable labels for the rod, beam and shell element label header information.

For each element, immediately after writing out these three control information (ngeom, nlayer and n-results), the application program should write out 'ngeom' of results, each with 'n_results' values corresponding to the header variables as declared. Then, if nlayer is greater than one, repeats the output for the next layer. Essentially, total of 'ngeom*nlayer' results, each with n_results values, should be written to the file.

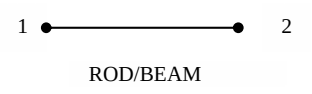
The following example demonstrate how to create the element-based result file:

```
#include <stdio.h>
#include <string.h>
#include <fstream.h>
#include "mdbverb.h"
#include "mdb.h"

extern double time_step, **results; //results for disp, temp, and eigen values
extern bool hasRod, hasBeam, hasShell;
char elem_outfilename[MAX_CHAR];

void elem_output_result()
{
    long int elem, i, maxelem, *elnodes, point, npoint, len, offset, ngeom, eltype, layer, nlayer, element_nresult[3];
    double *elem_results;

    ofstream out;
    SetModelFileName(elem_outfilename, ".eou");
    out.open(elem_outfilename);
    out << "data_version " << data_version << "\n";
    out << "data_title " << data_title << "\n";

    if (hasRod) {
        out << "element_result_rod";
        out << " Axial-dir1";
        out << "\n";
    }
    if (hasBeam) {
        out << "element_result_beam";
        out << " Axial-dir1";
        out << " Shear-dir2";
        out << " Shear-dir3";
        out << " Torsion-dir1";
        out << " Moment-dir2";
        out << " Moment-dir3";
        out << "\n";
    }
    if (hasShell) {
        out << "element_result_shell";
        out << " Force-X";
        out << " Force-Y";
        out << " Force-Z";
        out << " Moment-X";
        out << " Moment-Y";
        out << " Moment-Z";
        out << mdb_data_name(SIGXX);
        out << mdb_data_name(SIGXY);
        out << mdb_data_neme(SIGXZ);
        out << mdb_data_name(SIGYY);
        out << mdb_data_name(SIGYZ);
        out << mdb_data_name(SIGZZ);
    }

    out << "\n";
    out << "\time/step " << time_step << "\n";

    //output these results for demonstration purpose
    element_nresult[0] = 1; // axial stress
```

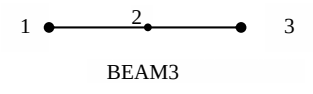
```
    element_nresult[1] = 6; // beam end forces
    element_nresult[2] = 12; // 6 stress and 6 end forces

    maxelem = mdb_index_max(ELEMENT);
    for (elem=0;elem<=maxelem;elem++) {
        *elnodes=mdb_integer(ELEMENT,elem,len);
        if (elnodes) {
            eltype = elnodes[0];
            elem_results = results[elem];
            if ( eltype == ROD) {
                ngeom = 1;
                nlayer = 1;
                out << elem << " " << ngeom << " " << nlayer << " " << element_nresult[0] << "\n";
                out << " " << elem_results[0];
                out << "\n";
            } else if ( eltype == BEAM) {
                ngeom = 2;
                nlayer = 1;
                out << elem << " " << ngeom << " " << nlayer << " " << element_nresult[1] << "\n";
                for (point=0 ; point < ngeom; point++) {
                    offset = point*6;
                    for (i=0;i<6;i++) out << " " << elem_results[offset+i];
                    out << "\n";
                }
            } else if ( eltype >= SHELL3 && eltype <= SHELL16 ) {  //process shell element output
                if (eltype == SHELL3 || eltype == SHELL6)
                    ngeom = 3;
                else
                    ngeom = 4;
                nlayer = 2;
                out << elem << " " << ngeom << " " << nlayer << " " << element_nresult[2] << "\n";
                for (layer=0; layer<nlayer; layer++) { //process 2 layers, bot and top
                    for (point=0 ; point < ngeom; point++) {
                    offset = (layer*ngeom+point)*12;
                    for (i=0;i<6;i++) out << " " << elem_results[offset+i]; //output end forces
                    for (i=0;i<6;i++) out << " " << elem_results[offset+6+i]; //output stresses
                    out << "\n";
                    }
                }
            }
        }
    }
    out.close();
}
```

Appendix: Naming and Numbering Scheme in AMPS system

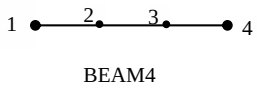
The following pictures describe the finite element Node, Side and Face numbering schemes used in AMPS system. The number with the prefix letter "S" refers to the element side/edge number, and the number with an 'F' prefix refers to the face number. Most AMPS elements are generated systematically from a Lagrangian family polnomial, so their node numbering usually are ordered from the low to the high dimension space, in sequence. Note that when retriving these nodes/faces/edges from MDB, the index should start from 0.



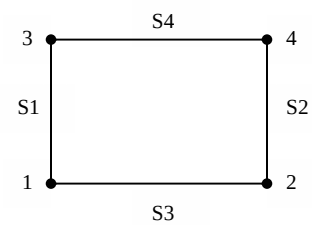
ROD/BEAM



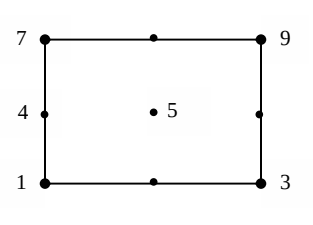
BEAM3



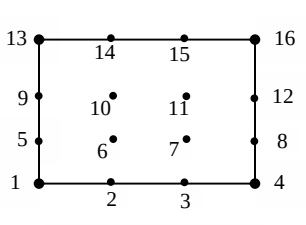
BEAM4



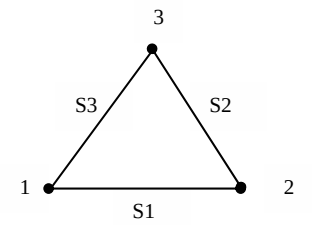
QUAD4/SHELL4



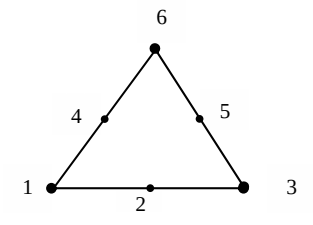
QUAD9/SHELL9



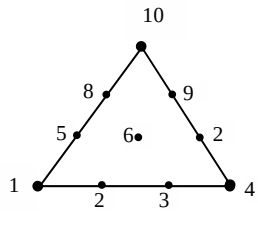
QUAD16/SHELL16



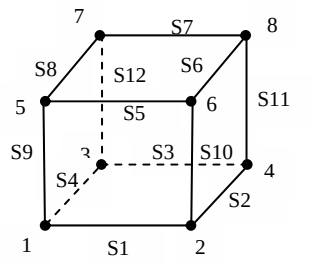
TRI/SHELL3



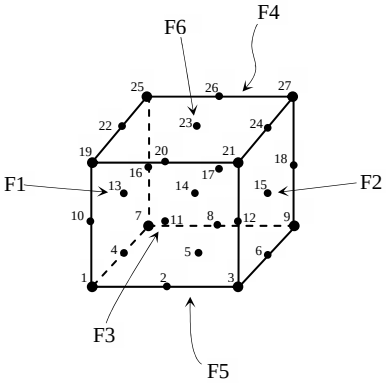
TRI6/SHELL6



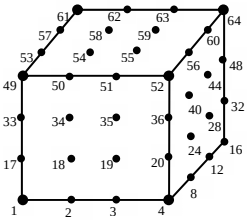
TRI10/SHELL10



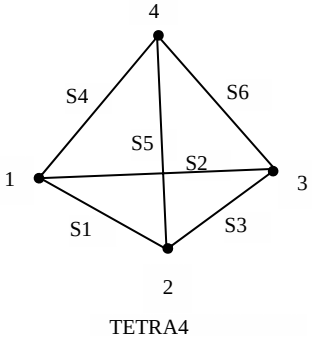
HEXA8



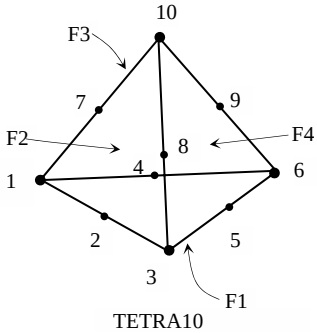
HEXA27



HEXA64



TETRA4



TETRA10