AMPS User Application Interface Reference Manual

Version 1.3, June, 2005

Table of Contents

Revision History:

Revision 0.8: October 22, 2003. Initial draft.
Revision 0.9: January 20, 2004. Revised interface functions usrload_nodal(), usrload_distributed(),usrmat_continuum. A newly created version naming control to distinguish the new DLL version.
Revision 1.0: February 2004. Official release.
Revision 1.1: February 2004. Added user constitutive material model interface, user nodal and distributed load functions
Revision 1.2: January 2005. Added initial material model properties initiation and user defined plasticity functions.
Revision 1.3: June 2005. Added user defined plasticity and viscoelastplasticity law.

Chapter 1: Introduction

Since the introduction of AMPS (Advanced Multi-Physics Simulation) , there are many various requests to open the AMPS system to the user supplied constitute material models and special user function that are not in the general AMPS design. Some users have requested to have special hydro loading, and some expressed desire to have special material models. It is for these reasons that we start to offer the user supplied interface module for wider AMPS applications.

To allow the general user functions to be used in AMPS analysis, we have carefully reviewed the existing methods and technologies, and have designed the interface based on the user's convenience, and with the best software engineering design. We have chosen the C++ language, specifically, Microsoft Visual Studio C++, as the basic platform of the user module development, but that can be easily expanded into any other language through the interface standard. The effort is part of the OpenAMPS development, and we will continue to expand the open system design.

This manual is structured in the following way after this introductory chapter: Chapter Two documents the concept and details of the user interface routines. Chapter Three presents a completed sample user interface routines. Appendix A gives some programming information and cautions about the multithread coding as used in AMPS system.

Chapter 2: Creating AMPS User Application Interface Routines

The AMPS user interface routines are designed as an implicit runtime dynamic-link-library (DLL). In such a way, these user routine specifications, as declared in the header file, must be carefully followed by the both the AMPS code and the user application routines to avoid dynamic runtime library linking errors. Although there are other methods available, using this method has demonstrated to be the most efficient method as in reflected in lots of DLL interfaces used in scientific software.

The user/developer should provide the DLL file to the AMPS system so it can be initialized during runtime. Usually, this means replacing the file usrapp.dll in the AMPS installation directory, or in any directory reachable during AMPS execution. Currently, all available interface routines are defined in the header file usrapp.h.

The application developers should install the supplied the sample code and the utility routines in their preferred directory on the development system. Currently, the sample usrapp.dll is compiled using MSVC 6.0 SP6, but other binary compatible compilers can be used to create the DLL library. All user routines should be contained in this usrapp.dll according to the interface declaration as described in this chapter.

It is necessary to set the compiler to use multithread option with a pre-processor setting of USRAPP_DLL to configure the proper macro for the DLL routine compilation. Declaring USRAPP_DLL compiler pre-processor definition will configure the macro USRAPP_API to the appropriate definition for the DLL code creation.

Since AMPS will be calling these user routines in a multithread fashion, there are some software design cautions about multithread coding should be considered. Please refer to Appendix A for some of these discussions.

2.1 User Application Interface Design

Following are the list of the current AMPS application interface routines (please look into the usrapp.h file for a complete declaration). We will refer to them in general term as "usrapp". They can be classified into the following categories:

General version control and identification routines:

```
void usrapp_name(char *app_name, const long int &len);
void usrapp_initdata(const double &usrapp_api_version, const long int &problem_dimension,
        const long int &plane_mode, const long int &n_dof_var, char **dof_name);
```

User constutive model interface routines:

```
void usrmat_history_size(long int &nstatev);
void usrmat_history_varname(char **history_varname, const long int &name_size);
void usrmat_continuum(const long int &prop_index, const long int &elementid, const long int &npt,
        const long int &kstep, const long int &iteration, const double &time, const double &dtime,
        double *stress, double *ddsdde, double &heatgen_density,
        double *strain, double *dstrain, double *dtstrain, double *dof_new, double *dof_old,
        double *new_deften, double *old_deften, double *statev, double *props, const long int &nprops,
        double *coords, double *drot, double *paxes, const long int &shellelement, double *shell_normal);
```

User defined load/flux/lhs/rhs interface routines:

```
void usrload_distributed(const long int &element, const long int &shellelement, const long int &prop_index,
    const long int &iteration, const long int &lenvalues, double *values, double *coord, const double &time, const double
    &dtime, double *dof_new, double *normal, double *usr_rhs, double *usr_lhs);

void usrload_nodal(const long int &node, const long int &prop_index, const long int &iteration, const long int &lenvalues,
    double *values, double *coord, const double &time, const double &dtime, double *dof_new, double *usr_rhs, double
    *usr_lhs);
```

When any of the user interface function is used in the model (such as a user material model or loading), the AMPS processor will automatic initiate the DLL by calling routine usrapp_name[] first to get the user specified announcement and the identifying

character string. The solver then echoes such identification string to the user during run time to inform the user what type of usrapp.dll is active. After that, it will pass the necessary model information to inform the usrapp. These could be information such as the active DOF, the problem dimension, etc. It is up to the user routine to store this information for their subsequent computation use, if necessary.

After these two initial function calls, the AMPS processor starts to process the model simulation calculations till it needs an user specified functions such as user material model or loading information.

As the usrapp interface demand grows, we will unavoidable add more routines or modify the existing interface to satisfy different application interfaces. The version control variable USRAPP_API_VER as declared in the usrapp.h is used to control this. Since this usrapp.dll is designed with an implicit DLL linking for error checking and best user flexibility, this interface header file "usrapp.h" should not be changed. Instead, if there is a need to change it or to add more functions, please contact AMPS support group for new functionality.

2.2 Details of User Interface Routines

Details of each interface routines are explained in this section. If these interfaces change, the version control variable USRAPP_API_VER is used to identify the interface version format. Currently, the following documentation are based on version 1.0.

void usrapp_name(char *app_name, const long int &len)

This is the first interface routine that will be called. The user is responsible to pass back to the AMPS system a string that is less than than **len** characters long, and it will be echoed back during runtime in the AMPS processor display window. Usually, this string is used to identify the purpose of usrapp, usage information, author/version information, etc. For example, the sample implementation coding could be:

```
char title1[]="My sample isotropic material constitutive law - ver 1.0\n";
strncpy(app_name, title1, len);
```

void usrapp_initdata(const double &usrapp_api_version, const long int &problem_dimension,
const long int &plane_mode, const long int &n_dof_var, char **dof_name)

This is the second interface routine that will be called. In this routine, the AMPS system informs the user couple key model information, and the user should store these data in their local storage, if they need to use them later.

usrapp_api_version
The AMPS system informs the usr the USRAPP_API_VER format version through **usrapp_api_version** variable. It is the user's responsible to stop the AMPS execution if this version number is newer than the version in the usrapp.h header file.
problem_dimension
This variable is the dimension of the problem. It is in the range from 1 to 3.
plane_mode
For two-dimensional problem, the variable **plane_mode** will be 0: plane-strain problem, 1: plane-stress problem, or 2:axisymmetric problem.
n_dof_var
This variable is the number of the solution variables that will be passed into user functions, and contains all active DOF's in the model.
dof_name
This variable is the name of the runtime variables. Pending on the analysis types included, they could be the following: "dispx","dispy", "dispz","rotx", "roty", "rotz", "velx", "vely", "velz", "rotvelx", "rotvely", "rotvelz", "pressure", "vort1", "vort2", "vort3", "temp", "voltage", "mpotential". If the user is interested in any of these runtime variables for dependency calculations, it is necessary to find out the position of the variable in the sequence of **n_dof_var** and store them in the local usrapp code. The user needs to do so since later in the calculations, all runtime variables data will be passed to the user function in these orders. Another important fact is that the DOF sequence is not fixed from model to model since they will depend on the order that the user defines the physical behavior.

```
//example code to find out the temperature DOF position and to check whether it is active
int temp_dof = -1;   //no temperature DOF in the model
for (int i=0;i<n_dof_var;i++) {
    if (!strcmp(dof_name[i],"temp")) {
        temp_dof=i;  //yes, heat transfer temperature is active
        break;
    }
}
```

void usrmat_history_size(long int &nstatev)

If the AMPS system detects that the model is requesting a user supplied constitutive model, it will first ask the user routine to specify the amount of history/path-dependent variables that the user routine desires. It will set aside such variable space for each element integration point and at each node to store such information.

void usrmat_history_varname(char **history_varname, const long int &name_size)

If there are history variables requested, the user is asked to supply the naming desired for these variables. These are needed since for each result output stage, these user variable values will be printed in the output file, and will need a name for AMPView post-processing view. The variable **name_size** is the maximum character length allowed for the naming. For example, the sample implementation could be:

```
char var_name[12];
for (int i=0;i<nStateVariable;i++) {// nStateVariable is the local copy of n_dof_var
    sprintf(var_name,"my_var%d",i);
    strncpy(history_varname[0],var_name,name_size);
}
```

void usrmat_continuum(const long int &prop_index, const long int &elementid, const long int &npt,
const long int &kstep, const long int &iteration, const double &time, const double &dtime,
double *stress, double *ddsdde, double &heatgen_density,
double *strain, double *dstrain, double *dtstrain, double *dof_new, double *dof_old,
double *new_deften, double *old_deften, double *statev, double *props, const long int &nprops,
double *coords, double *drot, double *paxes, const long int &shellelement, double *shell_normal)

This is the user constitutive material construction routine. This routine is called at every integration point of the element with the user-defined material property. The AMPS program will pass the previously stored information, and the user's responsibility is to provide the updated corresponding data, specifically, **stress, ddsdde**, **dtstrain** and **statev,** as detailed below.

prop_index

Since the user defined material model can be used in different material sets, this number is the index of the material property set. It is useful if the user decides to treat different property sets with different calculations.

elementid,

This is the element number of the integration.

npt

This is the integration point count, and the counting is zero based, i.e. the first point is zero.

kstep

This is the index of the analysis type being performed. In most of cases, the analysis usually contains only one analysis type. When the user is asking for an eigen analysis or an in-situ modal/eigen analysis, there will be more than one analysis types. The first one is usually a static/dynamic analysis, and the second analysis is an eigen analysis. The counting is zero based.

iteration

This is the analysis iteration count, and it will start with index one. For each new time/load step, AMPS always starts the iteration counting from index one.

time, dtime

Variable **time** is the time/step at the beginning of the load increment, and **dtime** is the time/load step increment.

Stress

This is the Cauchy stress tensor at the integration point at the beginning of the step loading. Its components are σxx, σyy, σzz, τxy, τxz, τyz. It is the user's responsibility to update this stress tensor based on the given previous stress and the strain data, or from any associated history state variables that the user specified. In a small deformation analysis or when the nonlinear geometry formulation is a Total Lagrangian formulation, this stress state reference frame is the original un-deformed state, and for the Updated Lagrangian formulation, it refers to the previous reference configuration.

ddsdde

This is a 6x6 matrix relating the strain increment to the stress increment at the integration point that the user should supply. The strain DOF's are εxx, εyy, εzz, γxy, γxz, γyz, and the stress components are σxx, σyy, σzz, τxy, τxz, τyz. It is the user's responsibility to properly fill the 6x6 constitutive matrix relating the incremental strain to the incremental stress.

heatgen_density

If the model contains the heat transfer region, the user can supply a heat generation density at the integration point, and the processor will compute the equivalent nodal heat flow from this heat generation density using volume integration.

strain, dstrain, dtstrain

Array **strain** is the strain tensor at the integration point at the beginning of the time step, and similarly, array **dstrain** is the current strain increment corresponding to the **dtime** time increment. The six strain components are εxx, εyy, εzz, γxy, γxz, γyz, and they are the mechanical strain as stored without the thermal strain. The temperature strain is given to the user in the **dtstrain** array computed by the processor based on the temperature state and the thermal expansion coefficient, and can be modified by the user if the supplied function desires to compute the thermal strain based on the user's specific method.

dof_new, dof_old

These are the run time active DOF variables in the order as described in the routine **usrapp_initdata.** The size of the array is **n_dof_var** with each DOF name as described in **dof_name** array. Since the user routine may need the solution variable of the current state and the previous state, both data sets are supplied.

new_deften, old_deften

These are the deformation tensor that is generally needed if the user desire to usr other stress quantity such as Piola_kirchoff stress or Green-Lagrangian strain other than the stanard Cauchy stress and strain.

statev

This is the user specified history state variable as requested by the user in function **usrmat_history_varname** at the beginning of the execution. The state variables at the beginning of the step loading iteration are passed to the user, and the user is responsible to update these variables to reflect the new user state information, if any. The AMPS processor will maintain the old and the new history variable automatically for the user, and it is the user's responsibility to use it in a consistent manner.

props, nprops

The user-specified material property array **props**, if any, are passed into this routine. It is up to the user to decide the usage of them. For example, they could be the constants for the material property calculations, or any desired values that the routine wants to get from the user. The integer variable **nprops** is the length of the **props**[] array. Note that these data are enetered in AMPView, and then stored in the data file. The first entry is reserved for version control, and the user data are stored in index 1 to n, i.e., **nprop** should be n+1 if there are n user data entered in AMPView.

coords

This is the coordinate of the integration point position in (X,Y,Z) coordinate format. If either UL or TL large deformation option is used, it refers to the deformed position.

drot

This is the incremental rotational matrix representing the rigid body rotation from the previous state to the current state. It is provided to the user in case in the large deformation analysis (either the Total Lagrangian or the Updated Lagrangian method) that the user function may need such incremental rotation transformation. For a small deformation analysis, this is an identity matrix.

paxes

If this array is not a NULL array, then it is the indication that the user has specified anisotropic axes through the AMPS AMPView controls. It will be an array of (x1,y1,z1, x2,y2,z2). The unit vector (x1,y1,z1) will point to the first local axis, and (x2,y2,z2) for the second axes. If the element is a shell/membrane element, then these two vectors should be projected onto the shell/membrane surface using the specified **shell_normal** array. For more details, please refer to the AMPS/AMPView on-line reference manual, specifically in the anisotropic material property section and in the shell material property controls.

shellelement

If this is a non-zero number, it indicates that the element is a membrane/shell element. The shell normal direction will be given in the shell_**normal**[] array, and the other local axis directions can be found from the **paxes**[] array. For more details, please refer to the AMPS/AMPView on-line reference help.

shell_normal

If the element being considered is an shell/membrane element, this will contain the shell element normal direction vector (x,y,z).

void usrload_distributed(const long int &element, const long int &shellelement, const long int &prop_index, const long int &iteration, const long int &lenvalues, double *values, double *coord, const double &time, const double dtime, double *dof_new, double *normal, double *usr_rhs, double *usr_lhs)

This is the user distributed load calculation routine. This routine is called at every integration point of the surface/side load integration process. The user is responsible to compute the surface/side integration point value of the distributed pressure/flux/rhs intensity and store them in the usr_rhs array, and AMPS will use them to compute the total surface/side integral by multiplying the returned usr_lhs and usr_rhs data with the proper integration weighting and the associate geometric integration weight. If the distributed pressure/flux requires a LHS entry (e.g. in a convective heat transfer boundary condition), the user should also calculate the LHS value and store them in the usr_lhs array.

element

This is the element number of the integration point.

shellelement

If this number is non-zero, it indicates this is a membrane/shell element. In such case, the shell normal direction will be given in the **normal**[] array, and the other local axis directions can be found from the **paxes**[] array. For more details, please refer to the AMPS/AMPView on-line reference help.

prop_index

Since the user-defined distributed load can be used in different boundary condition sets, this number is the index of the BC_TYPE. It is useful if the user decides to treat different BC_TYPE sets with different calculations (e.g. BC_TYPE=1 is a hydrostatic, and BC_TYPE=2 for user surface convective heat transfer).

iteration

This is the analysis iteration count, and it will start with index one. For each new time/load step, AMPS always starts the iteration counting from index one**.**

lenvalues

When the user-defined distributed load is used, AMPView can store a **values**[] array so the data can be used for load/flux calculations. The variable l**envalues** is the length of the **values**[] array size.

values

This user-specified array data are passed into this routine. It is up to the user to decide the usage of them. Example usage are pressure value, heat transfer coefficient, or any desired values that the routine wants to get from the user.

coord

This is the coordinate of the integration point position in (X,Y,Z) coordinate format. If UL or TL large deformation options is used, it is at the deformed position.

time, dtime

Variable **time** is the time/step at the beginning of the load increment, and **dtime** is the time/load step increment.

dof_new

This is the run time active DOF variables in the order as described in the routine **usrapp_initdata.** The size of the array is **n_dof_var** with each DOF name as described in **dof_name** array.

normal

If the element associated with the surface integration is an shell/membrane element, this will contain the shell element normal direciton (x,y,z) normal vector.

usr_rhs, usr_lhs

These are the values that the user will compute and return them back to AMPS for integration calculations. The sequence of these data must be associated in the DOF order as described in the **usrapp_initdata.** The size of the array provided from AMPS is **n_dof_var,** and the user must put appropriate values in the corresponding DOF as described in **dof_name** array.

void usrload_nodal(const long int &node, const long int &prop_index, const long int &iteration, const long int &lenvalues, double *values, double *coord, const double &time, const double &dtime, double *dof_new, double *usr_rhs, double *usr_lhs, long *node_dof)

This is the user nodal load calculation routine. This routine is called at every node that the user has specified a user-defined nodal load type.

node
This is the node number of the node.
prop_index
Since the user-defined distributed can be used in different boundary condition sets, this number is the index of the BC_TYPE. It is useful if the user decides to treat different BC_TYPE set with different calculations.
iteration
This is the analysis iteration count, and it will start with index one. For each new time/load step, AMPS always starts the iteration counting from index one**.**
lenvalues
When the user-defined distributed load is used, AMPView can store the **values** array so the data can be used for load/flux calculations. The variable **lenvalues** is the length of the **values** array size.
values
This user-specified array data are passed into this routine. It is up to the user to decide the usage of them. Example usage are load value, heat transfer coefficient, or any desired values that the routine wants to get from the user.
coord
This is the coordinate of the integration point position in (X,Y,Z) coordinate format. If UL or TL large deformation options is used, it is at the deformed position.
time, dtime
Variable **time** is the time/step at the beginning of the load increment, and **dtime** is the time/load step increment.
dof_new
This is the run-time active DOF variable in the order as described in the routine **usrapp_initdata.** The size of the array is **n_dof_var** with each DOF name as described in **dof_name** array.
usr_rhs, usr_lhs
These are the values that the user will compute and return them back to AMPS for nodal load calculation. The sequence of these data must be associated in the order as described in the **usrapp_initdata.** The size of the array provided from AMPS is **n_dof_var,** and the user must put appropriate values in the corresponding DOF as described in **dof_name** array.
node_dof
If the user decides to set the nodal solution to a desired value(s), this **node_dof** array of the associated DOF should be set to an integer value 1 to indicate that the **usr_lhs** array contains desired fixed value. Please note that once the **node_dof** DOF is set, the **usr_rhs** data will not affect the result at all since the corresponding DOF is now a prescribed value, rather than governed by the **usr_rhs** load.

void usrprop_init(const long int &element, const long int &int_pt, const long int &prop_index, double *coord_int, double *dof_initial, double *stress_initial, double *strain_initial)

This is the user defined material properties initiation function. The commonly used stress and strain data can be initialized quickly at each element integration point, and if there are specifically DOF requiring initial values, the user can also acess that through a more detailed **dof_initial** array.
This function will be called from the AMPSol processor if the user specify in AMPVIew that the material property **prop_index** need special initialization.
element
This is the element number of the model.
int_pt
It refers to the integration point number inside the element.
prop_index
This is the material property index as assigned in the AMPView. It is passed to the user just for information, and can be used by the user if a specific initialization process is necessary.
dof_initial
This is an array of the run time variable at this integration in the order as described in the routine **usrapp_initdata.** The user can set each data to any value corresponding to the desired initial state.
stress_initial, strain_initial
This is part of the dof_inistal array, except it has been specifically initialized in the AMPView material initial value assignment.

void usrprop_plasticity(const long &element, const long &int_pt, const long &update_statev, const double &dtime,
const long int &nprops, double *props, double *dof_old, double *dof_new,
double *old_strain, double *old_plastic_strain, double *dstrain, double *plastic_dstrain,
double *old_statev, double *new_statev, double *stress, double &yield_f, double &flow_f)

This is the user defined plasticity function. It is designed for special plastic yield function or non-associated flow rule that the standard plastic material model could not provide. The function is activated in AMPView when the user selects the "user-defined plasticity" check box in the material property sheet.

element

This is the element number of the model.

int_pt

It refers to the integration point number inside the element.

update_statev

This is an interger value passed from the AMPSol processor. When it is a non-zero value, the user is required to calculate and update the history data array **new_statev** based on the current state of stress and strain passed into this fucntion.

dtime

dtime is the current time/load step increment..

nprops, props

These two are the user supplied property data as entered in the AMPView processor. **nprops** is the size of the **props** array. Note that props[0] is reserved for version control, and the user data entered in AMPView is stored from props[1] to props[n].

dof_old, dof_new

These two arrays are the data of the previous and the current run time variables at this integration in the order as described in the routine **usrapp_initdata**. The commonly used strain and stress data are provided at the separate array data, and if the user need more information at this integration point other than those provided, these two arrays will always contain all runtime variables describing the state of the analysis**.**

old_strain, old_plastic_strain, dstrain, plastic_dstrain

These are the total strain and the incremental strain values of the previous and the current time step. Note that they are in the order of ε_{xx}, ε_{xy}, ε_{xz}, ε_{yx}, ε_{yy}, ε_{yz}, ε_{zx}, ε_{zy}, ε_{zz}.

old_statev, new_statev

These are the user specified history state variable as requested by the user in function **usrmat_history_varname** at the beginning of the execution. They corresponding to the previous and the current time step history values. If the integer data **update_statev** is non-zero, it is require to update the **new_statev** array data to reflect the current new variable state.

stress

This is the stress tensor corresponding to the current state. Note that it is in the order of are σ_{xx}, τ_{xy}, τ_{xz}, τ_{yx}, σ_{yy}, τ_{yz}, τ_{zx}, τ_{zy}, σ_{zz}.

yield_f, flow_f

These two are the plastic yield function and the flow rule function used for the plastic normality flow strain calculations. Normally these two are the same for a plastic model following the associated flow rule, and they are different if non-associated plasticity models.

void usrprop_viscoplastic (const long &element, const long &int_pt, const long int &nuser_data, const double *user_data, const double &time, const double &dtime, const double &temp, const double &stress_equ, const double &e_creep_strain, const double *dof_old, const double *dof_new, double &creep_strain, double &creep_strain_rate)

This is the user defined creep function for thermoelastoplasticity calculation. It is designed for special creep function to be used in the AMPS viscoelastoplasticity calculation. The function is activated in AMPView material property sheet when the user selects the "with creep effect (thermo/elasto/plastic)" check box "User defined creep function" in the creep property sheet. Note that since AMPS uses alpha-method for time step integral, this function is called multiple times per integration point

element

This is the element number of the model.

int_pt

It refers to the integration point number inside the element.

time, dtime

time is the current calculation time, and **dtime** is the current time/load step increment..

nuser_data, user_data

These two are the user supplied property data as entered in the AMPView processor. **Nuser_data** is the size of the **user_data** array. Note that user_data[0] is reserved for version control, and the user data entered in AMPView is stored from user_data[1] to user_data[n].

temp

If heat transfer analysis is activated, this is the current temperature at the integration point.

stress_equ

This is the current estimated equivalent uniaxial stress that can be used for creep function calculations.

e_creep_strain

This is the adjusted equivalent creep strain based on the ORNL ornl-tm-3602 creep strain calculation guidelines.

dof_old, dof_new

These two arrays are the data of the previous and the current run time variables at this integration in the order as described in the routine **usrapp_initdata**. This is only necessary if the user need more information at this integration point other than those provided. These two arrays will always contain all runtime variables describing the state of the analysis**.**

creep_strain

This is the estimated creep strain that the user should compute and provide based on the current state (stress, temperature, time, etc.). This data is needed for the creep strain hardening calculations.

creep_strain_rate

This is the estimated creep strain rate that the user should compute and provide based on the current state. The incremental creep strain is computed based on this creep strain rate.

Chapter 3: A Sample usrapp.dll Implementation

The following is a completed usrapp.dll sample reference implementation. The complete listing is available in the usrapp installation directory.

The example material model is a simple isotropic material model with the user material property supplied from the AMPS processor. The example distributed pressure/flux example is a general hydrostatic distributed loading and the user nodal load routine implement a simple time dependent nodal load.

```
// usrapp.cpp : Defines the entry point for the DLL application.

#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include "usrapp.h"
#include "utils.h"

long int nStateVariable=1; //nStateVariable is the material model state variable length (material model need history data storage)
long int ProblemDimension=3;
long int PlaneMode=0;
long int nDofVar=0;
long int temp_dof=-1, disp_dof=-1, force_disp_dof[3]={-1,-1,-1},eps_plastic_dof=-1, history_dof=-1;;

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved)
{
#ifdef _DEBUG
  switch (ul_reason_for_call)
  {
    case DLL_PROCESS_ATTACH: // 1
    case DLL_THREAD_ATTACH:  // 2
    case DLL_THREAD_DETACH:  // 3
    case DLL_PROCESS_DETACH: // 0
      break;
  }
#endif
  return TRUE;
}

USRAPP_API void usrapp_name(char *app_name, const long int &len)
{
  char title1[]="Sample isotropic material constitutive law - ver 1.0\n";
  char title2[]="Sample hydrostatic loading and convective boundary conditiond - ver 1.0";
  strncpy(app_name,title1, len);
  strncat(app_name,title2, len);
}

USRAPP_API void usrapp_initdata(const double &usrapp_api_version, const long int &problem_dimension,
              const long int &plane_mode, const long int &n_dof_var, char **dof_name)
{
  /*
  usrapp_api_version: reserved for api format control.
  problem_dimension: the dimension of the problem being solved
  plane_mode: for two-dimensional problem, 0:plane strain, 1:plane stress, 2:axisymmetric
```

```
 n_dof_var: the number solution variables that will be passed to user functions
 dof_name: the name of the variable. pending on the analysis types included, they could be the following:
  "dispx","dispy", "dispz","rotx", "roty", "rotz", "velx", "vely", "velz", "rotvelx", "rotvely", "rotvelz",
  "pressure", "vort1", "vort2", "vort3", "temp",
  "voltage", "mpotential"
 */
 if (usrapp_api_version > USRAPP_API_VER) {
  printf("Error! usrapp version format error!");
  throw "Error in usrdll";
 }

 ProblemDimension=problem_dimension;
 PlaneMode=plane_mode;
 nDofVar=n_dof_var;

 //example code to find out the temperature DOF position and to check whether it is active
 for (int i=0;i<nDofVar;i++) {
  if (!strcmp(dof_name[i],"temp")) {
   temp_dof=i;
  } else if (!strcmp(dof_name[i],"dispx")) {
   force_disp_dof[0]=i;
   disp_dof = i; //AMPS always stores dispx/y/z in sequence
  } else if (!strcmp(dof_name[i],"dispy")) {
   force_disp_dof[1]=i;
  } else if (!strcmp(dof_name[i],"dispz")) {
   force_disp_dof[2]=i;
  } else if (!strcmp(dof_name[i],"voltage")) {
   voltage_dof=i;
  } else if (!strcmp(dof_name[i],"eps_plastic")) {
   eps_plastic_dof=i;
  } else if (!strcmp(dof_name[i],"my_var0")) {
   history_dof=i;
  }
}

USRAPP_API void usrmat_history_size(long int &nstatev)
{
 //inform AMPS that we need this amount of history state variables in our material model
 nstatev = nStateVariable;
}

USRAPP_API void usrmat_history_varname(char **history_varname, const long int &name_size)
{ //sample to set the variable name
 char var_name[12];
 for (int i=0;i<nStateVariable;i++) {
  sprintf(var_name,"my_var%d",i);
  strncpy(history_varname[0],var_name,name_size);
 }
}

USRAPP_API void usrmat_continuum(const long int &prop_index, const long int &elementid, const long int &npt,
            const long int &kstep, const long int &iteration, const double &time, const double &dtime,
            double *stress, double *ddsdde, double &heatgen_density,
            double *strain, double *dstrain, double *dtstrain, double *dof_new, double *dof_old,
           double *new_deften, double *old_deften,double *statev, double *props, const long int &nprops, double *coords,
           double *drot, double *paxes, const long int &shellelement, double *shell_normal)
```

```
{
 //Cauchy stress:Sxx,Syy,Szz,Sxy,Sxz,Syz, Lagrangian Strain: Exx,Eyy,Ezz,Gxy,Gxz,Gyz
 double young=props[1], poisson=props[2], fac, tmp[6];
 //a simple linear elastic thermal stress material model, no history variable needed

 //ddsdde[6x6] has been initialized to zero in AMPS
    if (ProblemDimension==2 && PlaneMode==1) {//plane stress mode
        fac = young/(1.-poisson*poisson);
        ddsdde[0] = ddsdde[7] = fac;
        ddsdde[1] = ddsdde[6] = fac*poisson;
        ddsdde[21] = fac*(1-poisson)*0.5;
    } else {
        fac = young*(1.-poisson)/((1.+poisson)*(1.-2*poisson));
        ddsdde[0] = ddsdde[7] = ddsdde[14] = fac; //lamda+2pnu
        ddsdde[1] = ddsdde[2] = ddsdde[8] = ddsdde[6] = ddsdde[12] = ddsdde[13] = (poisson/(1.-poisson))*fac; //lamda
        ddsdde[21] = ddsdde[28] = ddsdde[35] = ((1.-2.*poisson)/(1.-poisson))*fac * 0.5;//pnu
    }

 mat_multiply(ddsdde,dstrain,tmp,6,6,1);
 vector_add(stress,tmp,stress,6);
 if (temp_dof>=0) {
  mat_multiply(ddsdde,dtstrain,tmp,6,6,1);
  vector_add(stress,tmp,stress,6);
  heatgen_density = props[3];
 }
 return;
}

USRAPP_API void usrload_distributed(const long int &element, const long int &shellelement, const long int &prop_index,
     const long int &iteration, const long int &lenvalues, double *values, double *coord, const double &time, const double
    &dtime, double *dof_new, double *normal, double *usr_rhs, double *usr_lhs)
{
 long int idof;
 double hydro_pressure=0, surf_level[3]={0.0,0.0,0.0}, dir[3]={0.0,-1.0,0.0}, h0, h1;
 if (lenvalues>1 && disp_dof>=0) { //if there is user data, and the continuum field is active
  //sample hydrostatic pressure with the user input data as:
  //value[0] is the weight density
  //value[1-3] is a point in the fluid surface, default is (0,0,0)
  //value[4-6] is the direction of the gravity, default is (0,-1,0)
  if (lenvalues>=3)
   vector_copy(values+1,surf_level,ProblemDimension);
  if (lenvalues>=6) {
   vector_copy(values+4,dir,ProblemDimension);
   vector_normalize(dir,ProblemDimension);
  }
  h0=vector_inner_product(surf_level,dir,ProblemDimension);
  h1=vector_inner_product(coord,dir,ProblemDimension);
  if (h1>h0) {
   //compute the hydrostatic pressure in gloabl components, normal[] is away from surface
   hydro_pressure = values[0]*(h1-h0);
   for (idof=0;idof<ProblemDimension;idof++)
    usr_rhs[force_disp_dof[idof]] = -hydro_pressure*normal[idof];
}
 }
 if (temp_dof>=0) {
  //sample heat transfer convection - compute a simple convective heat transfer with default h=0.1, ambient temperature = 100;
```

```
   //value[7-8] is the optional surface convective coefficient and the ambient temperature
   double h_conv=0.1, temp_ambient=100.0;//sample default setting
   if (lenvalues>=7) h_conv = values[7];
   if (lenvalues>=7) temp_ambient = values[8];
   usr_rhs[temp_dof] = h_conv*(temp_ambient-dof_new[temp_dof]);
   usr_lhs[temp_dof] = h_conv;
  }
}

USRAPP_API void usrload_nodal(const long int &node, const long int &prop_index, const long int &iteration, const long int
&lenvalues, double *values,double *coord, const double &time, const double &dtime, double *dof_new, double *usr_rhs,
double *usr_lhs, long *node_dof)
{
  int idof;
  if (lenvalues>0 && disp_dof>=0) { //if there are user data and if the continuum field is active
    //sample time dependent nodal loads
    //value[0-2] is the load magnitude in 3 direction, varying with time
      int ndof = (lenvalues > ProblemDimension) ? ProblemDimension : lenvalues;
      double scale = time+dtime;
      for (idof=0;idof<ndof;idof++)
           usr_rhs[force_disp_dof[idof]] = values[idof]*scale; //compute the nodal force scaled by time
  }
}

USRAPP_API void usrprop_plasticity(const long &element, const long &int_pt, const long &update_statev, const double
&dtime,
              const long int &nprops, double *props, double *dof_old, double *dof_new,
              double *old_strain, double *old_plastic_strain, double *dstrain, double *plastic_dstrain,
              double *old_statev, double *new_statev, double *stress, double &yield_f, double &flow_f)

// plasticity yield function and flow rule function.
// element: element number
// int_pt: element integration point number
// props[]: Input. User supplied data values.
// dof_new: Input. It contains the primary dof values at this integration point at the new time point t+dt.
// old_statev: Input. It contains the old estimates for the materi_history_variables (if initialized in the data file).
// new_statev: Output. On output it should contain the new estimates for the materi history variables (if initialized in the data
file).
// stress:   stress.
// yield_f:  Yield function
// flow_f    plastic flow rule function (for associated flow rule, flow_f = yield_f)
{
  double sigY0, Hp;
  //Example standard von Mises yield function
  //if sigY0 is the initial stress and Hp is the hardening parameter, and they are stored in props[]
  double s11,s22,s33,s12,s13,s23,sig0,s1,s2,s3, sig_equ, eps_plastic, temp = 20.0, coef_sigk= 768.6, coef_k=12.77,
sig_yield=377.0, y_strain=0.002;
  double strain[9], eps_total, tmp;
  s11 = stress[0];
  s22 = stress[4];
  s33 = stress[8];
  s12 = stress[1];
  s13 = stress[2];
  s23 = stress[3];
  sig0 = (s11+s22+s33)/3.0;
  s1 = s11-sig0;
```

```
  s2 = s22-sig0;
  s3 = s33-sig0;
  //equivalent stress is sqrt(3/2 sum(DevitoricStressIJ*DevitoricStressIJ))
  sig_equ = sqrt(1.5*(s1*s1+s2*s2+s3*s3 + 2.0*(s12*s12+s13*s13+s23*s23)));


  sigY0 = props[1];
  Hp = props[2];
  eps_plastic = dof_new[eps_plastic_dof];//eps_plastic_dof has been stored in usrapp_initdata
  sig_yield = sigY0+Hp*eps_plastic;

  //define the yield function and the flow rule (use associated flow rule,i.e. same as yield function)
  yield_f = sig_equ - sig_yield;
  flow_f = yield_f;
}

USRAPP_API void   usrprop_viscoplastic (const long &element, const long &int_pt, const long int &nuser_data, const double
     *user_data, const double &time, const double &dtime,  const double &temp, const double &stress_equ, const double
     &e_creep_strain, const double *dof_old, const double *dof_new, double &creep_strain, double &creep_strain_rate)
// user creep law
// element: element number
// int_pt: element integration point number
// crpcon[]: Input. User supplied creep law constants.
// time: Input. Current analysis time
// dtime: Input. Current analysis time increment
// temp: Input. Current temperature.
// e_creep_strain: Input. Effective creep strain based on ORNL creep strain origin modification.
// stress_equ: Input. Current equivalent uniaxial stress
// creep_strain:  Output. estimated creep strain
// creep_strain_rate: Output.  estimated creep strain rate
{
      //sample power creep law - note that power() is the special user defined power function since the standard pow() function
      //can not evaluate the result when the based is zero.
     double tmp = user_data[1] * power(stress_equ, user_data[2]);
     creep_strain = tmp * power(time, user_data[3]);
     creep_strain_rate = tmp * user_data[3] * power(time, user_data[3]-1.0);
}
```

Appendix A: Multithread Software Design Consideration

Since the AMPS system is developed with multithread operations in mind in order to achieve faster and more responsive calculations, it is important to compile all codes with the multithread option. It is also very important to understand the multithread nature of potential simultaneous execution of the code in single/multiple CPU's environment. It is required to have thread-safe coding in mind.

A typical mistake is to assume the shared heap memory data remain the same in the multithread environment. For example,

```
double stress[6]; //allocated in heap on shared memory

void compute_stress(double *stress)
      {
        //compute stress[] from material data, e.g. Sij = Cijkl*Ekl;
      }

      double compute_I1(void)
      {
        compute_stress(stress);
        return (stress[0]+stress[1]+stress[2]);
      }
```

Since it's possible the code will be simultaneously executed from different threads, there are possibilities that the stress[] variable will be changed by other threads, and the compute_stress[] part will be wrong. Instead, the thread's stack storage must be used, so each thread will have its own separate stress[] values.

```
double compute_I1(void)
{   double stress[6]; //allocated in thread's stack
    compute_stress(stress);
    return (stress[0]+stress[1]+stress[2]);
}
```

This is especially true if you are using the element/node/bc user-defined coding due to the potential of parallel execution of the multithread code.

Further detailed multithread code development can be found from most operation system and compiler manuals.