

4 WRITING NEW CONSTITUTIVE MODELS

4.1 Introduction

The methodology is similar to that for developing a user-defined model with *FISH* (Section 2.8 in the *FISH* volume). A model written in C++ is compiled as a DLL file (dynamic link library) that can be loaded whenever it is needed (see Section 4.9). The main function of the model is to return new stresses, given strain increments. However, the model must also provide other information, such as names, and perform operations such as writing and reading save files.

In the C++ language, the emphasis is on an *object-oriented* approach to program structure, using classes to represent objects. The data associated with an object are encapsulated by the object and are invisible outside the object. Communication with the object is by member functions that operate on the encapsulated data. In addition, there is strong support for a hierarchy of objects — new object types may be derived from a base object and the base-object's member functions may be superseded by similar functions provided by the derived objects. This arrangement confers a distinct benefit in terms of program modularity. For example, the main program may need access to many different varieties of derived objects in many different parts of the code, but it is only necessary to make reference to base-objects, not to the derived objects. The runtime system automatically calls the member functions of the appropriate derived objects. A good introduction to programming in C++ is provided by Stevens (1994); it is assumed that the reader has a working knowledge of the language.

The methodology of writing a constitutive model is described in Sections 4.2 through 4.5. Support functions and examples are described in Sections 4.6 through 4.8, and the mechanism for loading or creating a DLL is described in Section 4.9. Note that the DLL must be compiled using Microsoft Visual C++ (VC++) for operation in *FLAC*. The C++ source files mentioned here are compiled using VC++ Version 6.0. At present, the pre-release Version 4.1 of *FLAC* is necessary for using DLLs compiled in VC++; it is not as extensively tested as the release version, 4.0, and results should be treated with caution.

4.2 Base Class for Constitutive Models

The methodology described above is exploited in *FLAC*'s support for user-written constitutive models. A base class provides a framework for actual constitutive models, which are classes derived from the base class. The base class, called **ConstitutiveModel**, is termed an “abstract” class because it declares a number of “pure virtual” member functions (which are signified by the **=0** syntax appended to the function prototypes). This means that no object of this base class can be created and that any derived-class object *must* supply real member functions to replace each one of the pure virtual functions of **ConstitutiveModel**. [Example 4.1](#) provides a partial listing of **ConstitutiveModel** (which is contained in file “CONMODEL.H”). Some members of **ConstitutiveModel**, such as utility functions, are omitted from the listing in [Example 4.1](#); the use of utility functions (such as **YoungPoissonFromBulkShear**) is self-evident, and examples of their use can be found in the supplied model source files. Other functions are used by *FLAC* to manipulate and interrogate constitutive models; there is no reason for a user-written model to use or redefine these.

Example 4.1 Partial class definition for base class, **ConstitutiveModel**

```
class ConstitutiveModel {
public:
    EXPORT ConstitutiveModel(unsigned uTypeIn,bool bRegister=false);
    EXPORT virtual ~ConstitutiveModel(void);
    // ROUTINES THAT MUST BE SPECIFIED BY THE DERIVED TYPE
    virtual const char *Keyword(void) const=0;
    virtual const char *Name(void) const=0;
    virtual const char **Properties(void) const=0;
    virtual const char **States(void) const=0;
    virtual double GetProperty(unsigned ul) const=0;
    virtual ConstitutiveModel *Clone(void) const=0;
    virtual double ConfinedModulus(void) const=0;
    virtual double ShearModulus(void) const=0;
    virtual double BulkModulus(void) const=0;
    virtual double SafetyFactor(void) const=0;
    virtual unsigned Version(void) const=0;
    virtual void SetProperty(unsigned ul,const double &d)=0;
    EXPORT virtual const char *Copy(const ConstitutiveModel *cm)=0;
    virtual const char *Initialize(unsigned uDim,State *pst)=0;
    virtual const char *Run(unsigned uDim,State *pst)=0;
    EXPORT virtual const char *SaveRestore(ModelSaveObject *mso)=0;
};
```

4.3 Member Functions

Any derived constitutive-model class must provide actual functions to replace the virtual member-functions in **ConstitutiveModel**. These functions perform the operations described below.

const char *Keyword() A pointer is returned to a character array containing the name of the constitutive model, as the user will refer to it with the **MODEL** command. For example, **"elastic"** would be a valid string in C++.

const char *Name() A pointer is returned to a character array containing the name of the constitutive model that is to be used on printout (e.g., resulting from the **PRINT zone** command). The name may or may not be the same as that given by the **Keyword** member function, but note that *FLAC* may truncate long strings on printout. An example of a valid string is: **"Linear/elastic"**.

const char **Properties() A pointer is returned to an array of strings (character arrays) containing the names of model properties, with a null pointer to denote the end of the array of strings. The following array of strings is a valid example: **{"shear", "bulk", 0}**. The given names will be those recognized by the **PROPERTY** command. Note: The array of strings should be terminated by 0, as shown.

const char *States() A pointer is returned to an array of strings containing state names, with a null pointer to denote the end of the array. The names are used on printout and in plotting to identify user-defined internal states of the model (e.g., plastic flow). The following array of strings is a valid example: **{"yielding", "tension", 0}**. See the variable **mState** in [Section 4.5](#). Note: The array of strings should be terminated by 0, as shown.

SetProperty(unsigned n, const double &dVal) The value of **dVal** supplied by the call comes from a *FLAC* command of the form **PROP name=dVal**; the supplied value of **n** is the sequence number (starting with 1) of the property name previously specified by means of a **Properties()** call. The model object is required to store the supplied value in its appropriate private memory location.

double GetProperty(unsigned n) A value should be returned for the model property of sequence number **n** (previously defined by a **Properties()** call, with **n = 1** denoting the first property).

const char *Copy(const ConstitutiveModel *cm) This member function should first call the base class **Copy** function, and then copy all essential data from the model object pointed to by **cm** (assumed to be of the same derived class as the current model). In the event of an error, a string describing the error is returned; otherwise 0 should be returned. It is not necessary to copy data members that are recomputed when the **Initialize()** function is called.

const char *Initialize(unsigned uDim, State *ps) This function is called once for each model object (i.e., for each full zone) when the *FLAC* **CYCLE** command is given and when the large-strain update is performed. The model object may

perform initialization of its property or state variables, or it may do nothing. The dimensionality (e.g., this is 2 for *FLAC*) is given as **uDim**, and structure **ps** (see [Section 4.5](#)) contains current information for the zone containing the model object. A pointer to a character string should be returned if an error is detected; otherwise 0 is returned. Note that strains are undefined when **Initialize** is called. The *average* stress components for the full zone are available in the state structure; they cannot be changed by the **Initialize()** member function.

const char *Run(unsigned uDim, State *ps) This function is called for each sub-zone (up to ten per zone) at each cycle from within *FLAC*'s zone scan. The model must update the stress tensor from strain increments. The structure **ps** (see [Section 4.5](#)) contains the current stress components and the computed strain increment components for the sub-zone being processed. The stress components already contain the rotation-correction terms when **Run()** is called. A pointer to a character string should be returned if an error is detected; otherwise 0 is returned.

double ConfinedModulus(void) The model object must return a value for its best estimate of the maximum confined modulus. This is used by *FLAC* to compute the stable timestep. For a linear, elastic model, the confined modulus is $K + 4G/3$.

double ShearModulus(void) The model object must return a value for its best estimate of the current tangent shear modulus. This is used by *FLAC* to determine coefficients for quiet boundaries in dynamic mode.

double BulkModulus(void) This is not used by *FLAC* at present, but the model object should return its best estimate of the current tangent bulk modulus.

double SafetyFactor(void) This function is not used at present. It should return some value, such as 10.0.

unsigned Version(void) The version number of the constitutive model should be returned. This may be used to deal with the case of restoring files containing objects of earlier versions of the model, which perhaps omit certain variables.

ConstitutiveModel *Clone(void) A new object must be created, of the same class as the current object, and a pointer to it of type **ConstitutiveModel** returned. This function is called whenever *FLAC* installs the model in a zone.

const char *SaveRestore(ModelSaveObject *mso) This function is called when either the **SAVE** or **RESTORE** command is given. The model object should first call the **SaveRestore()** function of the base class. **SaveRestore** allows the model to save and restore data members of each object. Only integers and real variables are accepted, so other data types must be converted to those. The derived-class function must call **mso->Initialize(nd,ni)**, where **nd** is the number of **doubles** and **ni** is the number of **ints** to be saved/restored. The variables are then identified by calling **mso->Save(ns,var)**, where **ns** is the sequence number of the variable (from 0 to **nd-1** or 0 to **ni-1**, depending on whether reals or integers are being saved/restored),

and **var** is the variable to be saved. There are separate functions **Save()** for processing **double** or **int** variables. The structure of the **ModelSaveObject** class is irrelevant, except for the use of the member functions mentioned. It is defined in “CONMODEL.H.”

The model class definition should also contain a constructor that must invoke the base constructor. If the base constructor is called with **bRegister**, set **true**, then the derived model is *registered* with *FLAC* (see [Section 4.4](#)). A type number (**uTypeIn**) unique to the model must also be passed; this enables the correct model to be reinstalled in each zone when a problem is restored from a save file. It is recommended that a high value of type number be chosen (e.g., 100 or higher), to avoid conflicts with the built-in models, which start from type 1. In all other cases, the derived-class constructor should be called with no parameters, as in the **Clone** member function. Initialization of data members may be performed by the constructor, as illustrated in [Example 4.2](#). In this example, the model’s unique type number is the integer **mnMohrModel** (see [Example 4.4](#) for its definition), and the symbols **dBulk**, **dShear** etc., are the data members for the derived model.

Example 4.2 Typical model constructor

```
MohrModel::MohrModel(bool bRegister)
    :ConstitutiveModel(mnMohrModel,bRegister), dBulk(0.0),
      dShear(0.0), dCohesion(0.0), dFriction(0.0), dDilation(0.0),
      dTension(0.0), dYoung(0.0), dPoisson(0.0), dE1(0.0), dE2(0.0),
      dG2(0.0), dNPH(0.0), dCSN(0.0), dSC1(0.0), dSC3(0.0),
      dBISC(0.0), dDTEN(0.0) { }
```

4.4 Registration of Models

Each user-written constitutive model contains its own name and the name of its properties and state indicators. *FLAC* can determine this information by calling the appropriate member function, as described in [Section 4.3](#). *FLAC* is made aware of a user-written constitutive model by a constructor call invoked by a static global instance of a model object — see [Example 4.3](#). The object is constructed either when *FLAC* is loaded (for the “built-in” models such as Finn), or when a DLL is loaded (for external models). The **true** value of the argument causes the base constructor to “register” the new model, and add it to the list of models. Only one static registration of a particular model should be made; it is convenient to place it in the C++ source file of the model, so that the model is registered when its corresponding DLL file is loaded. The static instance of the model is consulted whenever *FLAC* needs any information about the model or needs to instantiate a copy of the model (using the **Clone** function).

Example 4.3 Global instantiation of a model object

```
static ElasticModel modelInstance(true);  
// ... forces a constructor call to the model registry
```

4.5 Information Passed between Model and *FLAC* during Cycling

The most important link between *FLAC* and a user-written model is the member-function **Run(unsigned nDim, State *ps)**, which computes the mechanical response of the model during cycling. A structure, **State** (defined in “CONMODEL.H”), is used to transfer information to and from the model. The members of **State** (all public) are as follows. Not all the information may be used by a particular code; the structure is intended to serve all Itasca codes.

unsigned char bySubZone Sequence number of the sub-zone currently being processed, starting at 0. This information may be used to scale accumulated sub-zone data correctly. For example, if four sub-zones are present (see **byTotSubZones**), accumulated values will need to be divided by four, in order to obtain the average for the whole zone.

unsigned char byTotSubZones Total number of sub-zones in the zone currently being processed, including those from all overlays, if present.

unsigned char byOverlay Number of times the volume of the current zone is represented (e.g., 2 represents two overlays).

unsigned long mState Model state indicator flag (or bitmap). Specific bits in this flag correspond to names in the **States()** member function. For example, a flag value of 1 (bit 0) represents the first state, 2 (bit 1) the second, 4 (bit 2) the third, 8 (bit 3) the fourth, etc. Any number of bits may be selected simultaneously (for example, both shear and tensile yield may occur together).

double dSubZoneVolume Volume of the current sub-zone.

double dZoneVolume Volume of the current full zone.

STensor stnE Strain increment tensor, input to the constitutive model. See [Section 4.6](#) for the names of components.

STensor stnS Stress tensor. The current tensor is input to the constitutive model, and the model must return the updated tensor. See [Section 4.6](#) for the names of components. Note that the components are effective stresses: the zone pore pressure is added to the direct total stresses (negative compression) before calling the **Run()** function, and subtracted on exit.

STensor stnI Stress increment tensor — not used by *FLAC*.

double dvRotX Increment of sub-zone rotation (spin velocity multiplied by timestep), input to the model in large-strain mode. This information may be used by models that have directional properties that must be updated in large-strain mode.

double dvRotY Not used by *FLAC*.

double dvRotZ Not used by *FLAC*.

double bDensity Density of full zone. Not used by *FLAC* at present.

double dTemp Temperature of the complete zone (input to model only).

double dTimeStep Timestep (input only).

double dppInc Optional increment of pore pressure produced by the model.

double dPorosity Current porosity of zone (input only).

double dTMUtility Unused at present.

bool bLarge True if the model is currently running in large-strain mode.

bool bTherm True if the thermal calculation mode is active.

bool bCreep True if the creep calculation mode is active.

bool bFluid True if the fluid (groundwater) calculation mode is active.

bool bViscous True if viscous strains are to be computed for the stiffness-proportional component of Rayleigh damping. This flag defaults to **false** if not set. Typically, it should be set **true** for an elastic increment, and **false** for an increment in which yielding occurs.

The main task of member-function **Run()** is to compute new stresses from strain increments. In a nonlinear model, it is also useful to communicate the internal state of the model, so that the state may be plotted and printed. For example, the supplied models indicate if they are currently yielding or have yielded in the past. Each sub-zone may set the variable **mState**, that records the state of a model as a series of bits that can be on or off (1 or 0). Each bit can be associated with a message that is displayed on the screen. The string returned by member function **States** contains sub-strings corresponding to bit positions that the model may set in **mState**. The first sub-string refers to bit 0, the second to bit 1, and so on. Several bits may be set simultaneously. For example, both shear and tensile yield may occur together. The operation of the state logic may be appreciated by consulting any of the nonlinear model files — e.g., “CMMOHR.CPP.”

4.5.1 Effect of Various Symmetry Modes

The existence and significance of various stress and strain components depends of the calculation mode selected in a *FLAC* simulation. These are examined below (denoting the strain increment components by ϵ_{ij} , and the effective stress components by σ_{ii}).

4.5.1.1 Plane Strain

Input: ϵ_{xx} , ϵ_{yy} , ϵ_{xy} are nonzero; ϵ_{zz} , ϵ_{yz} , ϵ_{xz} are zero.

The **Run()** function must return σ_{xx} , σ_{yy} , σ_{xy} , σ_{zz} .

4.5.1.2 Plane Stress

Input: ϵ_{xx} , ϵ_{yy} , ϵ_{xy} are nonzero; ϵ_{zz} , ϵ_{yz} , ϵ_{xz} are zero.

The **Run()** function must return σ_{xx} , σ_{yy} , σ_{xy} , and $\sigma_{zz} = 0$. Although ϵ_{zz} is not computed (or used) by *FLAC*, it may be computed internally in the constitutive function. There is no difference in *FLAC*'s calculation (externally to the constitutive model) for plane stress, compared to plane strain. The condition of plane stress is purely determined by the constitutive model; for example, by using different coefficients in the elasticity matrix (see below for a listing of the built-in Fortran elastic model). No equations of motion are performed for the out-of-plane (*z*) dimension. A user-defined model may compute an "internal" ϵ_{zz} , as mentioned above, and accumulate the increments, if the current out-of-plane strain is needed.

4.5.1.3 Axisymmetry

Input: ϵ_{xx} , ϵ_{yy} , ϵ_{xy} , ϵ_{zz} are nonzero; ϵ_{yz} , ϵ_{xz} are zero.

The **Run()** function must return σ_{xx} , σ_{yy} , σ_{xy} , σ_{zz} . ϵ_{zz} is computed by *FLAC* before calling the model's **Run()** function, based on motion in the *xy*-plane. The value of σ_{zz} returned by the model will influence the motion in the *xy*-plane computed by *FLAC*.

4.5.1.4 Summary

The user-written function should be identical for use in both plane strain and axisymmetry. *FLAC* handles (externally to the model) the generation of the four strain components and the correct use of the returned stress components. Thus, one model will run under plane strain or axisymmetry, without change or switches necessary.

If a user-defined model is to reproduce plane stress conditions, it should be written accordingly; the action of *FLAC* (external to the model) is identical for plane strain or plane stress. The user may designate a material "property" to act as a switch to invoke plane strain or plane stress. This is entirely a matter for the model.

Stress-rotation correction terms are applied by *FLAC* in large-strain mode before the model function is called. The user-defined function should just operate as if it is in small-strain mode.

Thus, there should be no need for a user-defined model function to know whether a *FLAC* simulation is in plane strain or axisymmetry. For plane stress, the model can be notified by a user-defined property switch.

4.5.1.5 *Illustration of the Difference between Plane Strain and Plane Stress*

The following extract is from *FLAC*'s built-in Fortran coding for the elastic model. Note that it is the model that determines whether the solution is plane strain or plane stress.

```

      .
      .
c -----
c Initialisation section
c -----
100 e1 = zx(kk) + c4d3 * zx(kg)
    e2 = zx(kk) - c2d3 * zx(kg)
    if ( (e1.eq.dpc0) .and. (e2.eq.dpc0) .and. mecflg) then
        nerr = 208
        error = .true.
        return
    endif
    if (pstrss) then
        div1 = e2 * e2 / e1
        zx(ke1) = e1 - div1
        zx(ke2) = e2 - div1
    else
        zx(ke1) = e1
        zx(ke2) = e2
    endif
    zx(kg2) = dpc2 * zx(kg)
    return
c -----
c Running section
c -----
200 continue
    s11 = s11 + de11 * zx(ke1) + (de22+de33) * zx(ke2)
    s22 = s22 + (de11+de33) * zx(ke2) + de22 * zx(ke1)
    if(.not.pstrss) s33 = s33 + (de11+de22) * zx(ke2) + de33 * zx(ke1)
    s12 = s12 + de12*zx(kg2)
c
    Rvisc = dpc1
    return
      .
      .

```

4.6 Utility Structures

Three structures/classes are provided to assist in writing and communicating with constitutive models: **STensor**, **Axes** and **ConTableList**, contained in files “STENSOR.*,” “AXES.*” and “CONTABLE.*,” respectively. The usage of the structures may be determined from the annotations in the header files and from examples in the constitutive model source files provided. The only essential components are the following data members of **STensor**: **d11**, **d22**, **d33**, **d12**, **d13** and **d23**. These correspond to the three direct stress or strain components, and the three shear-stress or -strain components, respectively. (Note that components **d13** and **d23** are not used by *FLAC*.) Tensors are assumed to be symmetric. The structure **STensor** also contains member functions that compute principal stresses and resolve stresses from global to principal axes, and vice versa. The full source code of these, and other, utility functions is available to the user for inspection.

4.7 Example Constitutive Model

The source code of all constitutive models used by *FLAC* are provided for the user to inspect or adapt. Here we extract, for illustration, parts of the Mohr-Coulomb elastic/plastic model contained in files “CMMOHR.*.” [Example 4.4](#) provides the class specification for the model, which also includes the definition of the model’s unique type number. Note that there are more private variables than property names (see the **Properties()** member function). In this model, some of the variables are for internal use only: they occupy memory in each zone, but they are not available for the user of *FLAC* to change or print out.

Example 4.4 Class specification for the Mohr-Coulomb model: file CMMOHR.H

```

class MohrModel : public ConstitutiveModel {
public:
    // static
    enum ModelNum { mnMohrModel=3 };
    // Creators
    EXPORT MohrModel(bool bRegister=false);
    // Accessors
    virtual const char *Keyword(void) const { return("mohr"); }
    virtual const char *Name(void) const { return("Mohr-Coulomb"); }
    virtual const char **Properties(void) const;
    virtual const char **States(void) const;
    virtual double GetProperty(unsigned ul) const;
    virtual ConstitutiveModel *Clone(void) const { return(new MohrModel()); }
}

virtual double ConfinedModulus(void)
    const { return(dBulk + d4d3*dShear); }
virtual double ShearModulus(void) const { return(dShear); }
virtual double BulkModulus(void) const { return(dBulk); }
virtual double SafetyFactor(void) const { return(10.0); }
virtual unsigned Version(void) const { return(2); }
// Manipulators
virtual void SetProperty(unsigned ul,const double &dVal);
virtual const char *Copy(const ConstitutiveModel *m);
virtual const char *Initialize(unsigned uDim,State *ps);
virtual const char *Run(unsigned uDim,State *ps);
virtual const char *SaveRestore(ModelSaveObject *mso);
private:
    double dBulk,dShear,dCohesion,dFriction,dDilation,
        dTension,dYoung,dPoisson;
    double dE1,dE2,dG2,dNPH,dCSN,dSC1,dSC2,dSC3,dBISC,dDTEN;
};

```

[Example 4.5](#) provides the constant definitions used by the model as well as the global instantiation of the model, as discussed in [Section 4.4](#).

Example 4.5 *Constant definition for Mohr-Coulomb model, and instantiation*

```
static const double d2d3 = 2.0 / 3.0;
static const double dPi = 3.14159265358979;
static const double dDegRad = dPi / 180.0;
static MohrModel mohrmodel(true);
```

The constructor for this model was listed in [Example 4.2](#). [Example 4.6](#) provides listings of the member functions for initialization and execution (“running”). Note that private model variables **dE1**, **dE2**, **dG2**, etc., are not computed at each cycle, to save time. Note also the use of the **State** structure in providing strain increments and stresses. In general, separate sections should be provided in every model for execution in two and three dimensions, to allow the same models to be used efficiently in *FLAC*. In this example, the 2D section is identical to the 3D section. Please refer to the file “CMMOHR.CPP” for listings of member functions: **Properties**, **States**, **GetProperties**, **SetProperties**, **Copy** and **SaveRestore**.

Example 4.6 *Initialization and execution sections of the Mohr-Coulomb model*

```
/****** INITIALIZE *****/
const char *MohrModel::Initialize(unsigned uDim,State *) {
    if ((uDim!=2)&&(uDim!=3)) return("Illegal dimension in Mohr constitutive
model");
    dE1 = dBulk + d4d3 * dShear;
    dE2 = dBulk - d2d3 * dShear;
    dG2 = 2.0 * dShear;
    double dRsin = sin(dFriction * dDegRad);
    dNPH = (1.0 + dRsin) / (1.0 - dRsin);
    dCSN = 2.0 * dCohesion * sqrt(dNPH);
    if (dFriction) {
        double dApex = dCohesion * cos(dFriction * dDegRad) / dRsin;
        dTension = dTension < dApex ? dTension : dApex;
    }
    dRsin = sin(dDilation * dDegRad);
    double dRnps = (1.0 + dRsin) / (1.0 - dRsin);
    double dRa = dE1 - dRnps * dE2;
    double dRb = dE2 - dRnps * dE1;
    double dRd = dRa - dRb * dNPH;
    dSC1 = dRa / dRd;
    dSC3 = dRb / dRd;
    dSC2 = dE2 * (1.0 - dRnps) / dRd;
    dBISC = sqrt(1.0 + dNPH * dNPH) - dNPH;
    dDTEN = (dBulk * dShear * 3.0)/((3.0 * dBulk + dShear)*dE1);
```

```

    return(0);
}
/***** RUN *****/
const char *MohrModel::Run(unsigned uDim,State *ps) {
    if ((uDim!=3)&&(uDim!=2)) return("Illegal dimension in Mohr constitutive
model");

    /*      store 'now' info. as 'past' and turn 'now' info off ---*/
    if (ps->mState & mShearNow) ps->mState = (unsigned long)(ps->mState  mS-
hearPast);
    ps->mState = (unsigned long)(ps->mState & ~mShearNow);
    if (ps->mState & mTensionNow) ps->mState = (unsigned long)(ps->mState
mTensionPast);
    ps->mState = (unsigned long)(ps->mState & ~mTensionNow);

    int iPlas = 0;
    double dTeTens = dTension;
    /* --- trial elastic stresses --- */
    double dE11 = ps->stnE.d11;
    double dE22 = ps->stnE.d22;
    double dE33 = ps->stnE.d33;

    ps->stnS.d11 += dE11 * dE1 + (dE22 + dE33) * dE2;
    ps->stnS.d22 += (dE11 + dE33) * dE2 + dE22 * dE1;
    ps->stnS.d33 += (dE11 + dE22) * dE2 + dE33 * dE1;
    ps->stnS.d12 += ps->stnE.d12 * dG2;
    ps->stnS.d13 += ps->stnE.d13 * dG2;
    ps->stnS.d23 += ps->stnE.d23 * dG2;
    /* --- calculate and sort ps->incips->l stresses and ps->incips->l direc-
tions --- */
    Axes aDir;
    double dPrinMin,dPrinMid,dPrinMax,sdif=0.0,psdif=0.0;
    int icase=0;
    bool bFast=ps->stnS.Resoltopris(&dPrinMin,&dPrinMid,&dPrinMax,&aDir,uDim,
&icase, &sdif, &psdif);

    double dPrinMinCopy = dPrinMin;
    double dPrinMidCopy = dPrinMid;
    double dPrinMaxCopy = dPrinMax;
    /* --- Mohr-Coulomb failure criterion --- */
    double dFsurf = dPrinMin - dNPH * dPrinMax + dCSN;
    /* --- Tensile failure criteria --- */
    double dTsurf1 = dTension - dPrinMin;
    double dTsurf3 = dTension - dPrinMax;
    /* --- tests for failure */
    if (dTsurf3 >= 0.) {

```

```

if (dFsurf < 0.) {
    iPlas = 1;
    /* --- shear failure: correction to ps->incips->l stresses ---*/
    ps->mState = (unsigned long)(ps->mState 0x01);
    dPrinMin -= dFsurf * dSC1;
    dPrinMid -= dFsurf * dSC2;
    dPrinMax -= dFsurf * dSC3;
}
} else if (dTsurf1 >= 0.) {
double dPdiv = -dTsurf3 + (dPrinMin - dNPH * dTension + dCSN) / dBISC;
if (dFsurf < 0.0 && dPdiv < 0.0) {
    iPlas = 1;
    /* --- shear failure: correction to ps->incips->l stresses ---*/
    ps->mState = (unsigned long)(ps->mState 0x01);
    dPrinMin -= dFsurf * dSC1;
    dPrinMid -= dFsurf * dSC2;
    dPrinMax -= dFsurf * dSC3;
} else {
    iPlas = 2;
    /* --- tension failure: correction to ps->incips->l stresses ---*/
    ps->mState = (unsigned long)(ps->mState 0x02);
    double dTco = dE2 * dTsurf3 / dE1;
    dPrinMin += dTco;
    dPrinMid += dTco;
    dPrinMax = dTension;
    double dTemp = dTeTens + dDTEN * dTsurf3;
    dTeTens = dTemp > 0.0 ? dTemp : 0.0;
}
} else {
    iPlas = 2;
    /* --- tension failure: correction to ps->incips->l stresses ---*/
    ps->mState = (unsigned long)(ps->mState 0x02);
    double dTco = dE2 * dTsurf3 / dE1;
    dPrinMin += dTco;
    dPrinMid += dTco;
    dPrinMax = dTension;
    double dTemp = dTeTens + dDTEN * dTsurf3;
    dTeTens = dTemp > 0.0 ? dTemp : 0.0;
}
}
if (iPlas) {
    ps->stnS.Resoltoglob(dPrinMin,dPrinMid, dPrinMax, aDir,
    dPrinMinCopy,dPrinMidCopy,dPrinMaxCopy, uDim, icense, sdif, psdif,
    bFast);
    ps->bViscous = false; // Inhibit stiffness-damping terms
} else {
    ps->bViscous = true; // Allow stiffness-damping terms
}

```

```
    }  
    return(0);  
}
```

4.8 *FISH* Support for Constitutive Models

The following *FISH* intrinsics are available in *FLAC* to interface with C++ models.

z_prop(*zp*,*p_name*)

This can be used on the left- or right-hand side of an expression.

Thus,

```
val = z_prop(zp,p_name)
```

stores in *val* the floating-point value of property named *p_name* in the zone of address *zp*. *p_name* may be a string containing the property name, or a *FISH* variable that evaluates to such a string. For example, **z_prop**(*zp*,*'bulk'*) would refer to the bulk modulus. If there is no constitutive model in *zp*, or the model does not possess the named property, then 0.0 is returned. Similarly,

```
z_prop(zp,p_name) = val
```

stores *val* in the property named *p_name* in zone *zp*. Nothing is stored if there is no constitutive model in *zp* or if the model does not possess the named property or *val* is not an integer or floating-point number. In both uses, *zp* must be a zone pointer and *p_name* must either be a string or a *FISH* variable containing a string.

z_model(*zp*)

The intrinsic is used in the following way:

```
var = z_model(zp)
```

causes the returned variable *var* to contain a string of the name of the model installed in zone *zp*. Otherwise, an integer variable zero will be returned, denoting that no constitutive model is present (the user's *FISH* function must check for the variable type before doing a comparison).

Note that model objects can only be installed in zones with the **MODEL** command (see [Section 4.9](#)). In the case of a conflict between the name of a built-in model and a loaded DLL model, the built-in model will be used. However, the use of the DLL model can be forced, by preceding the model name with the keyword **dll**. For example,

```
model dll elastic i = 1, j = 1
```

installs the DLL version of a model named "elastic" in zone (1,1).

4.9 Using and Creating User-written Model DLLs

Model DLL files may be loaded into *FLAC* while it is running by giving the command **MODEL load** <filename>, with the filename of the DLL. Thereafter, the new model name and property names will be recognized by *FLAC* and *FISH* functions that refer to the model and its properties. If the **MODEL load** command is given for a model that is already loaded, nothing will be done, but an informative message will be displayed.

In order to create a DLL in VC++, it is first necessary to create a workspace. The workspace will contain projects that are essentially a collection of C++ source and header files and their dependencies.

A workspace has already been created for the user — “UDM.DSW.” It contains a project called “UDM.DSP” that contains example source and header files called “USERSSOFT.CPP” and “USERSSOFT.H.” To create a DLL, the user may modify these files, or delete these files and include new ones. The user should refer to Microsoft VC++ documentation for further information on how to add and delete files from the project, change the settings for the project, etc. For more information on particular functions, refer to the additional comments found in the header and source files.

The user-defined models depend on the following files.

1. axes.h — specifies a particular axes system
2. conmodel.h — utility structure used to communicate with constitutive model
3. contable.h — defines the **TABLE** interface for general constitutive models
4. stensor.h — symmetric tensor storage

These header files should also be included in the project. These files get all unresolved definitions from “VCMODELS.DLL,” to which they are linked through the import library “VCMODELS.LIB.”

The resulting DLL will be placed in the “.\Release” directory. The default name of the DLL is “USERSSOFT.DLL.” To change this,

1. Set the particular project as active by clicking Project→Set Active Project.
2. Next, click on Project→Settings and click on the Link tab. Under category “General,” and following it, there is an edit box to input the output file name and the directory.

An example DLL implementation, which creates the file “USERSSOFT.DLL,” is included. This model is identical to the built-in strain-softening model. The data file “SS.DAT” provides a simple test that compares **MODEL userssoft** to **MODEL ss**.

4.10 References

Stevens, A. *Teach yourself C++*, 4th Ed. New York: MIS Press, 1994.