4 WRITING NEW CONSTITUTIVE MODELS

4.1 Zones

Users may create their own constitutive model for use in *UDEC*. This is an optional feature in *UDEC*. The model must be written in C++ and compiled as a DLL (dynamic link library) file. It can be loaded whenever it is needed. The main function of the model is to return new stresses, given strain increments. However, the model must also provide other information (such as name of the model and material property names) and describe certain details about how the model interacts with the code.

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 of 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hierarchy of objects: new object types may be derived from a base object, and the base-object's member functions may be superseded by similar functions provided by the derived objects.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. It is assumed that the reader has a working knowledge of the C++ programming language. (A good introduction is provided by Stevens 1994.)

The methodology of writing a constitutive model in C++ for operation in *UDEC* is described in Section 4.1.1. This includes descriptions of the base class, member functions, registration of models, information passed between the model and *UDEC*, and the model state indicators. The implementation of a DLL model is described and illustrated in Section 4.1.2. This includes descriptions of the support functions used by the model, the source code for an example model, *FISH* support for user-written models, and the mechanism for creating and loading a DLL. All of the files referenced in this section are contained in the "\UDEC700\pluginfiles\cmodels" folder.

Note that a DLL ***must*** be compiled using Microsoft Visual Studio 2017 for operation in *UDEC*.

To get started quickly and provide a project for examination, take the following steps.

1. Install *UDEC* and run it. The first time you run it, application data will be copied to Documents\Itasca\udec700 (by default).
2. Go to this folder and navigate to pluginfiles\cmodels\example.
3. Copy the examples folder and rename it.
4. Rename Example2017.sln, Example2017.vxproj, modelexample.h and modelexample.cpp to include the name of your new model in place of example2017 and example.
5. Open the solution in Visual Studio 2017.

6. Visual Studio will tell you that it cannot load the project. Click OK and then right click on the Example project in the Solution Explorer and choose Remove.
7. Right click on the solution and select Add -> Existing Project. Choose your renamed project from step 4.
8. In the new project, remove Example.h and Example.cpp.
9. Add the renamed .h and .cpp files from step 4.
10. At the top of the .cpp file, change the name of the .h file to be included.
11. In the .h and .cpp file, do a search and replace to rename the class from ModelExample to your model class.
12. Change the Solution Configuration to Release and the Solution Platform to x64
13. In the .cpp file, change the return values from extern "C" EXPORT_TAG const char *getName() to the name of your model. Note that the name has to be of the form modelxxxx where xxxx is the name of the constitutive model.
14. Change the return value in the getName() function to be just the name of the model.
15. Change the return value for the getFullName function.
16. Go to ProjectâŁ“>Properties. Click on Configuration PropertiesâŁ“>General. Ensure that the Configuration is set to Release and the platform is set to x64. Change the target name to modelxxxx006_64 where xxxx is the name of your model.
17. Build the project. The dll should appear in the x64\release folder.
18. Make modifications to the model and rebuild.
19. Put the dll in the folder C:i\Program Files\Itasca\udec700\exe64\plugins\cmodel where *UDEC* is installed.
20. Start *UDEC* and use the command zone model xxxx to change the zone constitutive model.

4.1.1 Methodology

4.1.1.1 Base Class for Constitutive Models

The methodology described above is exploited in *UDEC*'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 (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**.

4.1.1.2 Member Functions

Any derived constitutive-model class must provide actual functions to replace the virtual member-functions in **ConstitutiveModel**.

The model class definition should also contain a constructor that must invoke the base constructor. In all 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 4.1. In this example, the symbols **bulk_**, **shear_**, etc. are the data members for the derived model.

Example 4.1 Typical model constructor

```
ModelExample::ModelExample()
  bulk_(0.0), shear_(0.0), cohesion_(0.0),
  friction_(0.0), dilation_(0.0), tension_(0.0),e1_(0.0),
  e2_(0.0), g2_(0.0), nph_(0.0), csn_(0.0), sc1_(0.0),
  sc2_(0.0), sc3_(0.0), bisc_(0.0), e21_(0.0), rnps_(0.0)
```

4.1.1.3 Registration of Models

Each user-written constitutive model is compiled into a DLL that must be instantiated in the *UDEC* process. By convention, there are four exported functions in a DLL used as a plug-in to *UDEC*: **getName()**, **getMajorVersion()**, **getMinorVersion()** and **createInstance()**. You must also provide a stub function called **DllMain()**, which is called when the library is loaded and unloaded from the system. For example, here is how these functions appear for the Hoek-Brown model:

```
int _stdcall DllMain(void *,unsigned, void *)
{
  return 1;
}

extern "C" EXPORT_TAG const char *getName()
```

```
{
  return "modelhoekbrown";
}

extern "C" EXPORT_TAG unsigned getMajorVersion()
{
  return MAJOR_VERSION;;
}

extern "C" EXPORT_TAG unsigned getMinorVersion()
{
  return MINOR_VERSION;
}

extern "C" EXPORT_TAG void *createInstance()
{
  models::ModelHoek *m = new models::ModelHoek();
  return (void *)m;
}
```

The **EXPORT_TAG** macro indicates that these functions should be exported from the DLL.

The **DllMain()** function is always the same.

The exported DLL function **getName()** should always return a string that begins with the word "model." This indicates that the DLL is a constitutive model plug-in. In the above example, the string "modelhoekbrown" is returned by the exported DLL function **getName()**. The **getName()** method of the **ConstitutiveModel** class returns the string "hoekbrown" (without "model" prefixed to the name as in the **getName()** function that is exported. This convention of prefixing the exported name with "model" should be followed. The **ConstitutiveModel**'s **getName()** function must return a unique string (this is the method used to distinguish constitutive models from each other).

The **getMajorVersion()** function should not be altered. The major version is determined by the base constitutive model DLL, and indicates binary compatibility. By convention, this number will also be indicated in the file name of the DLL you produce.

The **getMinorVersion()** function indicates the minor version update of your constitutive model. For example, if new properties are added to a constitutive model, it might not be save file-compatible with an older version. In this case, the minor version number (defined in file "version.txt") would be incremented by 1.

The **createInstance()** function actually creates and returns an instance of your class. This is stored in a registry and used (via the **clone()** function) to create all other instances.

4.1.1.4 Information Passed between Model and UDEC during Cycling

The most important link between *UDEC* 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 "state.h"), is used to transfer information to and from the model.

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 whether they are currently yielding or have yielded in the past. Each zone may set the variable **`state_`**, which 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 **`state_`**. 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 bit assignment is described in Section 4.1.1.5. The operation of the state logic may be appreciated by consulting any of the nonlinear model files (e.g., "modelexample.cpp").

4.1.1.5 State Indicators of Zones

Each zone has a member variable that maintains its current state indicator. The member variable has 32 bits that can be used to represent a maximum of 16 distinct states. The state indicator bits are used by built-in constitutive models to denote plastic failure of triangles in a zone. See Table 4.1 for bit assignment and the corresponding failure state for built-in constitutive models.

Table 4.1 Failure states and bit assignments

		Hex	Decimal	Binary
mShearNow	=	0x0001	1	0000 0000 0000 0001
mTensionNow	=	0x0002	2	0000 0000 0000 0010
mShearPast	=	0x0004	4	0000 0000 0000 0100
mTensionPast	=	0x0008	8	0000 0000 0000 1000
mJointShearNow	=	0x0010	16	0000 0000 0001 0000
mJointTensionNow	=	0x0020	32	0000 0000 0010 0000
mJointShearPast	=	0x0040	64	0000 0000 0100 0000
mJointTensionPast	=	0x0080	128	0000 0000 1000 0000
mVolumeNow	=	0x0100	256	0000 0001 0000 0000
mVolumePast	=	0x0200	512	0000 0010 0000 0000
unused	=	0x0400	1024	0000 0100 0000 0000
unused	=	0x0800	2048	0000 1000 0000 0000
unused	=	0x1000	4096	0001 0000 0000 0000
unused	=	0x2000	8192	0010 0000 0000 0000
unused	=	0x4000	16384	0100 0000 0000 0000
unused	=	0x8000	32768	1000 0000 0000 0000

For user-defined constitutive models, the user can create a named state and assign any particular bit for that state, and subsequently update the triangular state indicator variable. The named states in Table 4.1 are used by built-in constitutive models to update the failure states of triangular zones, and show one particular use of the state indicator variable. If a user uses the state indicator variable to indicate failure states in their own model, they should make sure that there is no conflict with failure state constants of built-in models if they plan to use both of them in an analysis.

The named states in Table 4.1 are used by built-in models to update the triangular zone state indicator:

1. Drucker-Prager
2. Mohr-Coulomb
3. Strain-Hardening/Softening
4. Ubiquitous-Joint
5. Softening-Ubiquitous-Joint
6. Double-Yield
7. Modified-Cam-Clay

8. Cap-yield
9. Cap-yield-simplified
10. Hoek-Brown
11. Hoek-Brown-PAC
12. WIPP-drucker
13. Power-Mohr
14. Burger-Mohr

UDEC calls the constitutive model function **`run()`** for each zone, to update its stress values. Typically, the state indicator is also updated in this process by the constitutive model. For built-in models, the state indicator denotes the failure state of the triangle. This is updated by the constitutive model using the logical "or" (|) operation with the zone state indicator variable and the current failure state calculated by the constitutive model. The user should be certain to appropriately set or un-set all previous states updated prior to the current state calculated by the constitutive model. The state of a triangle can then be checked using the logical "and" (&) operator with the state variable and desired user-defined state.

Suppose a triangle is undergoing failure in tension and shear. This failure state is stored in the state indicator of the triangle. The built-in constitutive model updates the state variable:

(a.) Initially, check the current state and set/un-set state indicator appropriately. For example,

```
if (Tet_State & mShearNow) {
   Tet_State & = ~mShearNow;   /* unset previous state */
   Tet_State | = mShearPast;   /* set previous state to new state */
}
if (Tet_State & mTensionNow) {
   Tet_State & = ~mTensionNow; /* unset previous state */
   Tet_State | = mTensionPast; /* set previous state to new state */
}
```

(b.) Calculate the current state of the triangle.

(c.) Update the state if necessary. For example,

```
   Tet_State |= mShearNow;
   Tet_State |= mTensionNow;
```

The result of the preceding operations is that the first and second bits are set for that particular zone. Additionally, during initialization (step a), the third and the fourth bits are also set. The *FISH* function **z_state(zi)** (see Section 2.5.3 in the ***FISH* volume**) returns a value of 15, the decimal equivalent of the first four bits being set.

Users can use the *FISH* logical **and** to find out the failure state at any point in the analysis. Users can also split the output into additive powers of two, and find out all distinct failure states. For

example, if **z_state** returns 15, then 15 can be written as $15 = 1 + 2 + 4 + 8$ (additive powers of two), and from Table 4.1, the zone is in shear and tensile failure, and had shear and tensile failure in the past.

4.1.2 Implementation

4.1.2.1 Utility Structures

A few structures/classes are provided to help in writing and communicating with constitutive models. Some are provided by "base005.dll", which defines the base level of functionality common to all plug-in interfaces. Others are provided by "conmodel005.dll", which defines the specific interface used by the constitutive model system. In all cases, full documentation of the class definition is available in the base interface module of the programmer's interface documentation (in the on-screen HELP menu).

"base006.dll" provides:

Int, UInt, Byte, UByte, Double, Float, etc. – defined in base/src/basedef.h.
These types are substitutions for the standard C++ types **int**, **unsigned**, **char**, **double**, etc. These types are used instead to define consistent size definitions. For instance, an **Int** is guaranteed to be a signed 32-bit quantity regardless of the platform.

String – defined in base/src/string.h.
This class is derived from **std::basic_string<wchar.t>**, or the ANSI C++ standard string class for unicode. The **String** class adds a few handy utility functions, such as string, to numeric conversion.

DVect3 – defined in base/src/vect.h.
This class is actually the **double** instance of the template class **Vector3<T>**. Similar predefined types are **IVect (Vector3<Int>)** and **UVect (Vector3<UInt>)**. This class allows one to treat a three-dimensional vector as a primitive type, with full operator overloading for convenient syntax.

Variant – defined in base/src/variant.h.
This defines a type that can represent many different primitive types (e.g., **String**, **Double**, **Int**, etc.). This class is used to pass properties to and from the constitutive model.

Axes – defined in base/src/axes.h.
This class allows for the definition of an orthonormal basis. This basis can be used to convert coordinates to and from a "global" basis represented by the traditional Cartesian axes.

SymTensor – defined in base/src/symtensor.h.
This class defines a 3×3 symmetric tensor, typically used to represent stresses and strains. Member access is available through the **s11()**, **s12()**, **s13()**, **s22()**, etc. functions. Member modification is available through the **rs11()**, **rs12()**, etc. functions. In addition, eigenvector information (or principal directions and values) can be obtained through the **getEigenInfo()**

function. The helper class **SymTensorInfo** is used to allow the user to modify principal values while maintaining principal directions, and build up a new **SymTensor** with the result.

Orientation3 - defined in base/src/orientation.h.
This class provides storage and manipulation of an *orientation*, or a direction in space. This orientation can be specified either by a normal vector, or by a dip and dip direction.

In addition to the **ConstitutiveModel** and **State** interfaces, "conmodel007.dll" provides the following two utility functions in "convert.h": **getYPFromBS()** and **getBSfromYP()**. These functions can be used to convert Young's modulus and Poisson's ratio values to bulk and shear modulus values, and vice versa.

4.1.2.2 Example Constitutive Model

The source codes of all constitutive models used by *UDEC* 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 "Modelexample.*". Example 4.2 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 **getProperties()** 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 *UDEC* to change or print out. Also note that the **getProperty()/setProperty** interface is used for Save/Restore.

Example 4.2 Class specification for the model: file "modelexample.h"

```
#pragma once

#include "../src/conmodel.h"

namespace models
{
  class ModelExample : public ConstitutiveModel {
  public:
    ModelExample();

    virtual String  getName() const;
    virtual String  getFullName() const;
    virtual UInt    getMinorVersion() const;
    virtual String  getProperties() const;
    virtual String  getStates() const;
    virtual Variant getProperty(UInt index) const;
    virtual void    setProperty(UInt index,const Variant &p,
                                uint restoreVersion=0);
    virtual ModelExample *clone()
                    const {return new ModelExample(); }
    virtual Double  getConfinedModulus() const
                      { return bulk_+shear_*4.0/3.0; }
```

```
    virtual Double  getShearModulus() const { return shear_; }
    virtual Double  getBulkModulus() const { return bulk_; }
    virtual bool    supportsHystereticDamping() const {return true; }
    virtual void    copy(const ConstitutiveModel *mod);
    virtual void    run(UByte dim,State *s);
    virtual void    initialize(UByte dim,State *s);
    // Optional
    virtual Double  getStressStrengthRatio(const SymTensor &st) const;
    virtual void    scaleProperties(const Double &scale,
                                    const std::vector<UInt> &props);
    virtual bool    supportsStressStrengthRatio() const {return true;}
    virtual bool    supportsPropertyScaling() const { return true; }
  private:
    Double bulk_,shear_,cohesion_,friction_,dilation_,tension_;
    Double e1_,e2_,g2_,nph_,csn_,sc1_,sc2_,sc3_,bisc_,e21_,rnps_;
  };
} // namespace models

// EOF
```

Example 4.3 provides the constant definitions used by the model.

Example 4.3 Constant definition for the example

```
static const double d2d3 = 2.0 / 3.0;
static const double dPi=3.14159265358979323846264338327950288419716 9;
static const double dDegRad = dPi / 180.0;
static UserMohrModel usermohrmodel(true);
```

The constructor for this model was listed in Example 4.1. Example 4.4 provides listings of the member functions for initialization and execution ("running"). Note that, to save time, private model variables **e1_**, **e2_**, **g2_**, etc. are not computed at each cycle. Also note 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* or *UDEC*. In this example, the 2D section is identical to the 3D section. Please refer to the file "modelexample.cpp" for listings of member functions **getProperties**, **getStates**, **getProperty**, **setProperty** and **copy**.

Example 4.4 Initialization and execution sections of the example model

```
/*************************** INITIALIZE ************************/
void ModelExample::initialize(UByte dim,State *s)  {
     ConstitutiveModel::initialize(dim,s);
     e1_ = bulk_ + shear_*d4d3;
```

```
    e2_ = bulk_ - shear_*d2d3;
    g2_ = shear_*2.0;
    Double rsin = std::sin(friction_ * degrad);
    nph_  = (1.0 + rsin) / (1.0 - rsin);
    csn_  = 2.0 * cohesion_ * sqrt(nph_);
    if (friction_) {
        Double apex = cohesion_ / std::tan(friction_ * degrad);
        tension_ = std::min(tension_,apex);
    }
    rsin = std::sin(dilation_ * degrad);
    rnps_ = (1.0 + rsin) / (1.0 - rsin);
    Double ra = e1_ - rnps_ * e2_;
    Double rb = e2_ - rnps_ * e1_;
    Double rd = ra - rb * nph_;
    sc1_  = ra / rd;
    sc3_  = rb / rd;
    sc2_  = e2_ * (1.0 - rnps_) / rd;
    bisc_ = std::sqrt(1.0 + nph_*nph_) + nph_;
    e21_  = e2_ / e1_;
}

/*************************** RUN ************************/
void ModelExample::run(UByte dim,State *s) {
    ConstitutiveModel::run(dim,s);
    if (s->modulus_reduction_factor_>0.0) {
        Double shear_new=shear_ * s->modulus_reduction_factor_;
        e1_ = bulk_ + shear_new * d4d3;
        e2_ = bulk_ - shear_new * d2d3;
        g2_ = 2.0 * shear_new;
        Double ra = e1_ - rnps_ * e2_;
        Double rb = e2_ - rnps_ * e1_;
        Double rd = ra - rb * nph_;
        sc1_  = ra / rd;
        sc3_  = rb / rd;
        sc2_  = e2_ * (1.0 - rnps_) / rd;
        e21_  = e2_ / e1_;
    }
    // plasticity indicator:
    // store 'now' info. as 'past' and turn 'now' info off
    if (s->state_ & shear_now) s->state_ |= shear_past;
    s->state_ &= ~shear_now;
    if (s->state_ & tension_now) s->state_ |= tension_past;
    s->state_ &= ~tension_now;
    UInt plas = 0;

    /* --- trial elastic stresses --- */
```

```
Double e11 = s->stnE_.s11();//strain tensor normal components
Double e22 = s->stnE_.s22();
Double e33 = s->stnE_.s33();
s->stnS_.rs11() += e11 * e1_ + (e22 + e33) * e2_;
s->stnS_.rs22() += (e11 + e33) * e2_ + e22 * e1_;
s->stnS_.rs33() += (e11 + e22) * e2_ + e33 * e1_;
s->stnS_.rs12() += s->stnE_.s12() * g2_;
s->stnS_.rs13() += s->stnE_.s13() * g2_;
s->stnS_.rs23() += s->stnE_.s23() * g2_;

// default settings, altered below if found to be failing
s->viscous_ = true;  // Allow stiffness-damping terms

if (canFail()) {
    // Calculate principal stresses
    SymTensorInfo info;
    DVect3 prin = s->stnS_.getEigenInfo(&info);

    /* --- Mohr-Coulomb failure criterion --- */
    Double fsurf = prin.x() - nph_ * prin.z() + csn_;//
    /* --- Tensile failure criteria --- */
    Double tsurf = tension_ - prin.z();//
    Double pdiv = -tsurf +
                 (prin.x() - nph_ * tension_ + csn_)
                  * bisc_;

    /* --- tests for failure */
    if (fsurf < 0.0 && pdiv < 0.0) {
        plas = 1;
    /* shear failure: correction to principal stresses */
        s->state_ ^= shear_now;
        prin.rx() -= fsurf * sc1_;
        prin.ry() -= fsurf * sc2_;
        prin.rz() -= fsurf * sc3_;
    } else if (tsurf < 0.0 && pdiv > 0.0) {
        plas = 2;
    /* tension failure: correction to principal stresses */
        s->state_ ^= tension_now;
        Double tco = e21_ * tsurf;
        prin.rx() += tco;
        prin.ry() += tco;
        prin.rz()  = tension_;
    }

    if (plas) {
    /* transform back to refrence frame */
```

```
            s->stnS_ = info.resolve(prin);
        /* Inhibit stiffness-damping terms */
            s->viscous_ = false;
        }
    }
}
} // namespace models
```

4.1.2.3 FISH Support for Constitutive Models

The following *FISH* intrinsic is available in *UDEC*:

block.zone.prop(*zp,p_name*)

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

Thus,

```
val = block.zone.prop(zp,p_name)
```

stores in ***val*** the floating-point value of the property named ***p_name*** in the zone with index ***zp***. ***p_name*** may be a string containing the property name, or a *FISH* variable that evaluates to such a string. For example, **block.zone.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,

```
block.zone.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 index and ***p_name*** must either be a string or a *FISH* variable containing a string.

4.1.2.4 Loading and Running User-Written Model DLLs

Model DLL files may be loaded into *UDEC* while it is running by giving the **program load cmodel** filename command, with the file name of the DLL. DLL files will be automatically loaded if they are placed in the "exe64\plugins\cmodel" folder. Thereafter, the new model name and property names will be recognized by *UDEC* and *FISH* functions that refer to the model and its properties. If the **program load cmodel** command is given for a model that is already loaded, nothing will be done, but an informative message will be displayed.

Before constitutive model plug-ins can be assigned to zones, the model must be configured for their use by giving the **block config cppudm** command. Once so configured, the model will not cycle unless your *UDEC* license includes the C++ plug-in option.

4.1.2.5 Notes on Converting from the Previous Constitutive Model Interface

When converting a model written using the old constitutive model interface, the easiest thing to do is to follow the steps for creating a new constitutive model project described in the previous section. From that point, copy and edit methods from the old class to the new class. There are several noteworthy specific changes:

- Model number and registration Booleans are no longer needed in constructors.
- The **`getName()`** method is equivalent to the old **`Keyword()`** method.
- The **`getFullName()`** method is equivalent to the old **`Name()`** method.
- The **`getProperties()`** method now returns a single string delimited by commas (,) rather than an array of string pointers.
- The **`getStates()`** method now returns a single string delimited by commas (,) rather than an array of string pointers.
- The **`getProperty()`** and **`setProperty()`** functions now get/set a **`Variant`** rather than a **`double`**. On return, this conversion will happen automatically; on set, you will have to convert the **`Variant`** to a **`double`** using the **`toDouble()`** method.
- The **`Version()`** method is now called **`getMinorVersion()`**.
- The **`SafetyFactor()`** method is now called **`getStressStrengthRatio()`**.
- Note that **`initialize()`** is now called automatically by **`run()`** if **`isValid()`** returns **`false`**. **`isValid()`** is automatically set to **`false`** when a property changes.
- The **`SaveRestore()`** function is no longer used; all serialization is done using the property interface (**`getProperty()/setProperty()`**). If you have state variables that need to be saved, they need to have property names.
- The **`HDampInit()`** function is no longer used. Instead, return **`true`** from **`supportsHystereticDamping()`**, and use the **`hysteretic_damping_`** member of the **`State`** class.
- The **`State`** class now supplies much of its information via virtual functions. This allows little-used data to be calculated on demand, increasing overall efficiency.
- The **`STensor`** class is now called **`SymTensor`**. The six components of stress are no longer public members, and must be accessed through member functions.
- Getting principal stress components and directions has been simplified through the **`getEigenInfo()`** method and the **`SymTensorInfo`** utility class.

4.2 Joints

This optional feature allows the use of joint constitutive models that are developed and compiled outside of the executable code. The models exist as runtime dynamic link library (DLL) files. The section contains instructions and examples that assist the user in developing their own specialized joint constitutive models.

Use of the DLL models requires the use of the **block config cppudm** command. Any DLL model files placed in the "UDEC700\exe64\plugins\jmodel" folder will be loaded automatically at start-up. Otherwise, the models must be loaded via the **program load jmodel** <**filename**> command prior to their use. The loaded models become part of the runtime code. The models themselves are not saved or restored as part of the save files, so it is necessary to load any models that are not in the "plugins" folder prior to restarting a save file that uses them.

4.2.1 Mohr Coulomb-Slip Joint Model

This basic joint constitutive model ("jmodelexample.cpp") is the generalization of the Coulomb friction law. This law works in a similar fashion both for contacts between rigid blocks and contacts between deformable blocks. Both shear and tensile failure are considered, and joint dilation is included.

In the elastic range, the behavior is governed by the joint normal and shear stiffnesses, K_n and K_s.

The contact displacement increments are used to calculate the elastic force increments. The normal force increment, taking compressive force as positive, is

$$\Delta F^n = -K_n \ \Delta U^n \ A_c \tag{4.1}$$

and the shear force vector increment is

$$\Delta F_i^s = -K_s \ \Delta U_i^s \ A_c \tag{4.2}$$

where A_c = area of the contact.

The total normal force and shear force vectors are updated for the contact as

$$F^n = F^n + \Delta F^n \tag{4.3}$$

and

$$F_i^s = F_i^s + \Delta F_i^s \tag{4.4}$$

This instantaneous loss of strength approximates the "displacement-weakening" behavior of a joint. The new contact forces are corrected in the following manner (note that normal compressive force is positive).

for tensile failure:

$$\text{If } F^n < T_{\max}, \text{ then } F^n = T_{\text{residual}} \tag{4.5}$$

for shear failure:

$$\text{If } F^s > F^s_{\max}, \text{ then } F^s_i = F^s_i \frac{F^s_{\max}}{F^s} \tag{4.6}$$

where the shear force magnitude, F_s, is given by

$$F^s = (F^s_i \; F^s_i)^{1/2} \tag{4.7}$$

Dilation takes place only when the joint is at slip. The shear increment magnitude, ΔU^s, is given by

$$\Delta U^s = (\Delta U^s_i \; \Delta U^s_i)^{1/2} \tag{4.8}$$

This displacement leads to a dilation of

$$\Delta U^n(dil) = \Delta U^s \; tan\psi \tag{4.9}$$

where ψ is the dilation angle.

The normal force must be corrected to account for the effect of dilation – i.e.,

$$F^n = F^n + K_n \; A_C \; \Delta U^s \; tan\psi \tag{4.10}$$

Dilation is a function of the direction of shearing. Dilation increases if the shear displacement increment is in the same direction as the total shear displacement, and decreases if the shear increment is in the opposite direction.

This joint model is illustrated in Figure 4.1 for the case that joint cohesion is initially zero.

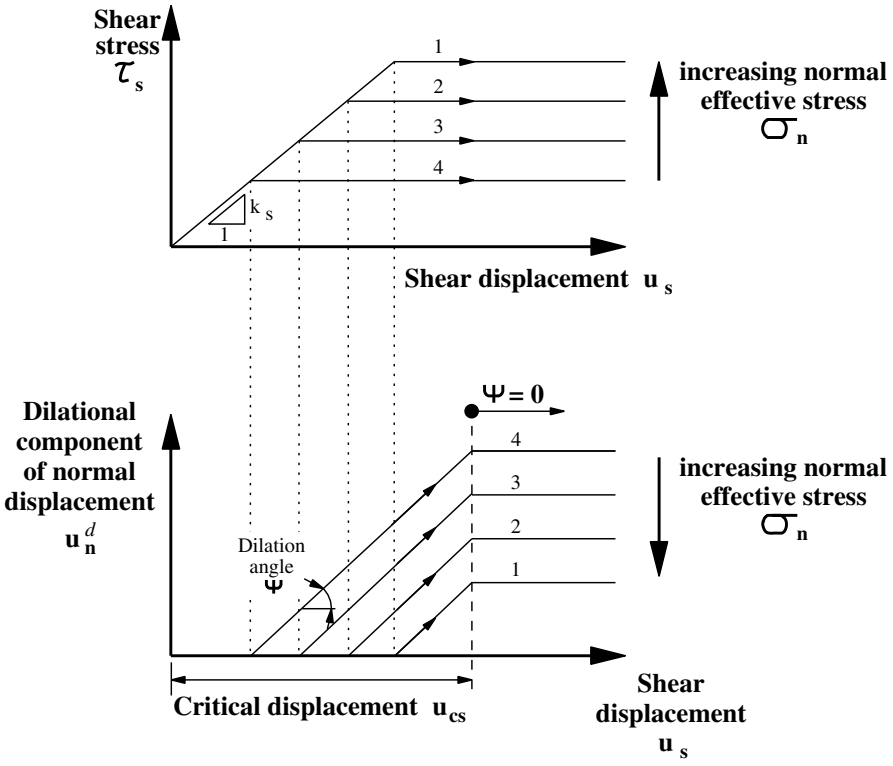

Figure 4.1 Mohr-Coulomb slip model (for zero joint cohesion)

For an intact joint (i.e., without previous slip or separation), the tensile normal force is limited to

$$T_{\max} = -T\ A_c \tag{4.11}$$

where T is the joint tensile strength.

The maximum shear force allowed is given by

$$F^s_{\max} = c\ A_c + F^n\ tan\phi \tag{4.12}$$

where c and ϕ are the joint cohesion [stress] and friction angle.

Once the onset of failure is identified at the contact (in either tension or shear), and residual values are specified, the tensile strength and cohesion are set to the residual values

$$T_{\max} = T_{\text{residual}} \tag{4.13}$$

$$F^s_{\max} = C\text{residual}A_c + F^n\ tan\phi \tag{4.14}$$

4.2.2 User-Defined Joint Models

4.2.2.1 Introduction

There is no *FISH* framework for adding joint constitutive models: the model must be written in C++, and compiled as a DLL (dynamic link library) file that can be loaded whenever it is needed. The main function of the model is to return new forces, given displacement 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 joint constitutive model in C++ is described in Section 4.2.2.2. This includes descriptions of the base class, member functions, registration of models, information passed between the model and the code and the model state indicators. The implementation of a DLL model is described and illustrated in Section 4.2.2.3. This includes descriptions of the support functions used by the model, the source code for an example model, *FISH* support for user-written models, and the mechanism for creating and loading a DLL. All of the files referenced in this section are contained in the "\ITASCA\UDEC700\pluginfiles\Jmodel" folder.

Note that a DLL ***must*** be compiled using Microsoft Visual Studio 2017 for operation in *UDEC*.

To get started quickly and provide a project for examination, take the following steps.

1. Install *UDEC* and run it. The first time you run it, application data will be copied to Documents\Itasca\udec700 (by default).
2. Go to this folder and navigate to pluginfiles\jmodels\example.
3. Copy the examples folder and rename it.
4. Rename Example2017.sln, Example2017.vxproj, jmodelexample.h and jmodelexample.cpp to include the name of your new model in place of example2017 and example.
5. Open the solution in Visual Studio 2017.
6. Visual Studio will tell you that it cannot load the project. Click OK and then right click on the Example project in the Solution Explorer and choose Remove.

7. Right click on the solution and select Add -> Existing Project. Choose your renamed project from step 4.
8. In the new project, remove jmodelexample.h and jmodelexample.cpp.
9. Add the renamed .h and .cpp files from step 4.
10. At the top of the .cpp file, change the name of the .h file to be included.
11. In the .h and .cpp file, do a search and replace to rename the class from jModelExample to your model class.
12. Change the Solution Configuration to Release and the Solution Platform to x64
13. In the .cpp file, change the return values from extern "C" EXPORT_TAG const char *getName() to the name of your model. Note that the name has to be of the form jmodelxxxx where xxxx is the name of the constitutive model.
14. Change the return value in the getName() function to be just the name of the model.
15. Change the return value for the getFullName function.
16. Go to ProjectâŁ“>Properties. Click on Configuration PropertiesâŁ“>General. Ensure that the Configuration is set to Release and the platform is set to x64. Change the target name to modelxxxx006_64 where xxxx is the name of your model.
17. Build the project. The dll should appear in the x64\release folder.
18. Make modifications to the model and rebuild.
19. Put the dll in the exe64\plugins\jmodel folder where *UDEC* is installed (e.g., C:\Program Files\Itasca\udec700\exe64\plugins\jmodel).
20. Start *UDEC* and use the command zone model xxxx to change the zone constitutive model.

4.2.2.2 Methodology

Base Class for Constitutive Models

The methodology described above is exploited to provide 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 **`JointModel`**, is termed an “abstract” class because it declares a number of “pure virtual” member functions (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 **`JointModel`**. Example 4.5 provides a partial listing of **`JointModel`** (contained in file “jconmodel.h”). Other functions are provided to manipulate and execute constitutive models; there is no reason for a user-written model to use or redefine these.

***Example 4.5 Partial class definition for base class,* `ConstitutiveModel`**

```
#pragma once

#include "jmodelbase.h"
#include <vector>
#include <iostream>

// NOTE:
// The file name of the DLL produced must be jmodel<name><###>.dll
//   or jmodel<name><###>_debug.dll (in debug).
// Where <name> is the string returned by getName(),
//   and <###> is the Major version number (3 digits).
// For instance jmodelexample006.dll or jmodelexample006_debug.dll

namespace jmodels
{
  struct State;

  class JMODEL_EXPORT JointModel
  {
  public:
    virtual String  getName() const=0; // Must be unique, used
      to identify model in save/restore, on command line, filename.
    virtual String  getPluginName() const { return getName(); }
    virtual String  getFullName() const=0; // Full name of model.
    virtual UInt    getMinorVersion() const; // Returns minor
      version of base implementation, override for actual model.
    virtual String  getProperties() const=0; // comma delimited
    virtual String  getStates() const=0; // comma delimited
    virtual Variant getProperty(UInt index) const=0; // Return
      real/0.0 if property doesn't exist.
    virtual void    setProperty(UInt index,const Variant &p,
                                UInt restoreVersion=0); // Calls
      setValid(0).  Return true if error.
    // Allows data other than properties to be saved efficiently.
    virtual void    save(std::ostream &o) const;
    // Allows data other than properties to be restored efficiently.
    virtual void    restore(std::istream &i,UInt restoreVersion);
    virtual JointModel *clone() const=0;
    virtual Double  getMaxNormalStiffness() const=0;
    virtual Double  getMaxShearStiffness() const=0;
    virtual void    copy(const JointModel *mod);
    virtual void    run(UByte dim,State *s); // If !isValid(dim)
      calls initialize(dim,s)
```

```
    virtual void    initialize(UByte dim,State *s); // calls
      setValid(dim)
    // Optional
    virtual Double  getStressStrengthRatio(const Double &,
                                           const DVect3 &) const
                                           { return(10.0); }
    virtual void    scaleProperties(const Double &,
                                    const std::vector<UInt> &)
                                    { std::logic_error("Does not
                                    support property scaling"); }
    virtual bool    supportsStressStrengthRatio() const
                                   { return(false); }
    virtual bool    supportsPropertyScaling() const
                                   { return(false); }
    virtual void    destroy() { delete this; }
    virtual bool    isSliding(const State &) { return false; }
    virtual bool    isBonded(const State &) { return true; }

    JointModel();
    virtual ~JointModel();
    // Major version changes when the interface changes.
    static UInt getMajorVersion();
    static UInt getLibraryMinorVersion();
    // Indicates whether initializion is necessary - by dimension
    bool isValid(UByte dimVal) const { return(valid_==dimVal); }
    void setValid(UByte dimVal) { valid_ = dimVal; }
    // Indicates whether failure should be allowed
    bool canFail() const { return(can_fail_); }
    void setIfCanFail(bool b) { can_fail_ = b; }
    // Indicates whether the model was loaded as a plugin
        (defaults to false).
    bool getPlugIn() const { return plugin_; }
    void setPlugIn(bool b) { plugin_ = b; }

  private:
    UByte valid_;
    bool  can_fail_;
    bool  plugin_;
  };
} // namespace jmodels

// EoF
```

Member Functions

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

`String getName()` returns a string containing the name of the constitutive model as the user will refer to it with the **block contact cmodel assign** command. For example, **`'èxample''`** would be a valid string in C++. This name must be unique, because it is used to identify the model during the save/restore process.

`String getFullName()` returns a string containing the name of the constitutive model that is to be used on printout. The name may or may not be the same as that given by the **`Keyword`** member function, but note that long strings may be truncated on printout. An example of a valid string is **`'èxample-mohr''`**.

`String getProperties()` returns a string containing the names of model properties. This array of strings is a valid example: {**`''JKN, JKS''`**}. The given names will be those recognized by the **block contact property** command.

`String getStates()` returns a string containing state names. The names are used on printout and in plotting, to identify user-defined internal states of the model (e.g., tension). This array of strings is a valid example: {**`''slip, tension,''`**}. See the variable **`mState`** in Section 4.1.1.4.

`setProperty(UInt n, const variant &dVal)` The value of **`dVal`** supplied by the call comes from a command of the form **block contact cmodel assign = dVal**; the supplied value of **`n`** is the sequence number (starting with 1) of the property name previously specified by means of a **`getProperties()`** call. The model object is required to store the supplied value in its appropriate private memory location.

`Variant getProperty(UInt n)` A value should be returned for the model property of sequence number **`n`** (previously defined by a **`getProperties()`** call, with **`n`** = 1 denoting the first property).

`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). It is not necessary to copy data members that are recomputed when the **`Initialize()`** function is called.

`initialize(UInt uDim, State *ps)` This function is called once for each model object (i.e., for each full zone) when the **CYCLE** command is given. 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 *UDEC*) is given as **`uDim`**, and structure **`ps`** (see Section 4.1.1.4) contains current information for the contact containing the model object.

`const char *Run(UInt uDim, JState *ps)` This function is called for each contact at each cycle. The model must update the forces from displacement increments. The structure **`ps`** (see Section 4.1.1.4) contains the current force components and the computed displacement components for the contact being processed.

`Double getMaxNormalStiffness()` This function returns the maximum normal stiffness. This is used to calculate a stable timestep.

`Double getMaxShearStiffness()` This function returns the maximum shear stiffness. This is used to calculate a stable timestep.

`UInt getMinorVersion` 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 (of the same class as the current object) must be created, and a pointer to it of type **`ConstitutiveModel`** returned. This function is called whenever a model is installed in a contact.

`Double getStressStrengthRatio(const Double &sn,const DVect3 &sstr)` returns the ratio of strength to the yield stress based on the passed normal stress, **`sn`**, and the shear stress components, **`sstr`**.

`scaleProperties(const Double &val,const std::vector<UInt> &array)` scales the properties specified in the integer array by the value ***val***. This is used in the strength reduction scheme used in the factor-of-safety calculations.

`Bool supportsStressStrengthRatio()` returns **`true`** if strength ratio calculations are supported.

`Bool supportsPropertyScaling()` returns **`true`** if property scaling is supported for factor-of-safety calculations.

The model class definition should also contain a constructor that must invoke the base constructor. 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 4.6.

Example 4.6 Typical model constructor

```
JModelExample::JModelExample() :
  kn_(0),
  ks_(0),
  cohesion_(0),
  friction_(0),
  dilation_(0),
  tension_(0),
  zero_dilation_(0),
  res_cohesion_(0),
  res_friction_(0),
  res_dilation_(0),
  res_tension_(0),
  tan_friction_(0),
```

```
    tan_dilation_(0),
    tan_res_friction_(0),
    tan_res_dilation_(0)
  {  }
```

Registration of Models

Each user-written joinr constitutive model is compiled into a DLL that must be instantiated in the *UDEC* process. By convention, there are four exported functions in a DLL used as a plug-in to *UDEC*: **`getName()`**, **`getMajorVersion()`**, **`getMinorVersion()`** and **`createInstance()`**. You must also provide a stub function called **`DllMain()`**, which is called when the library is loaded and unloaded from the system. For example, here is how these functions appear for the example model:

Example 4.7 Global registration of a model object

```
#ifdef EXAMPLE_EXPORTS
int __stdcall DllMain(void *,unsigned, void *)
{
  return 1;
}

extern "C" __declspec(dllexport) const char *getName()
{
#ifdef JMODELDEBUG
  return "jmodelexampled";
#else
  return "jmodelexample";
#endif
}

extern "C" __declspec(dllexport) unsigned getMajorVersion()
{
  return MAJOR_VERSION;
}

extern "C" __declspec(dllexport) unsigned getMinorVersion()
{
  return MINOR_VERSION;
}

extern "C" __declspec(dllexport) void *createInstance()
{
  jmodels::JModelExample *m = new jmodels::JModelExample();
  return (void *)m;
}
```

```
#endif
```

Information Passed between Model and the Code during Cycling

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

Double Area_ contact area

Double normal_force_ normal force

DVect3 shear_force_ shear force

Double normal_force_inc_ normal force increment

DVect3 shear_force_inc_ shear force increment

Double normal_disp_ normal displacement

DVect3 shear_disp_ shear displacement

Double normal_disp_inc_ normal displacement increment

DVect3 shear_disp_inc_ shear displacement increment

Double dDnop_ fraction of normal displacement increment that causes contact tension or separation

getTableIndexFromID(UInt id) returns table index for specified table number.

Double getYFromX(void *index,const Double &x) returns y-value based on x from the table index.

Double getSlopeFromX(void *index,const Double &x) returns local slope of table at point x.

Double working_[max_working_] This is a working area for values that must be stored between **run()** calls.

Int iworking_[max_iworking_] This is a working area for values that must be stored between **run()** calls.

Double getTimeStep() current timestep

UInt state 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, a value of 2 (bit 1) represents the second, a value of 4 (bit 2)

represents the third, a value of 8 (bit 3) represents the fourth, etc. Any number of bits may be selected simultaneously (for example, both shear and tensile yield may occur together). See Section 4.2.2.2 for a description of the failure states and bit assignment.

`Bool isThermal` True if the thermal calculation mode is active.

`Bool isCreep` True if the creep calculation mode is active.

`Bool isFluid` True if the fluid (groundwater) calculation mode is active.

The main task of member-function **`Run()`** is to compute new forces from displacement increments. In a slipping joint, 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 whether they are currently yielding or have yielded in the past. Each contact may set the variable **`mState`**, which 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 **`JStates`** 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 bit assignment is described in Section 4.2.2.2.

State Indicators

Each contact has a state indicator. The member variable has 32 bits that can be used to represent a maximum of 31 distinct states. The state indicator bits are used by built-in constitutive models to denote slip in a contact. See Table 4.2 for bit assignment and the corresponding failure state for built-in constitutive models.

Table 4.2 Failure states and bit assignments

		Hex	Decimal	Binary
slip-n	=	0x0001	1	0000 0000 0000 0001
tension-n	=	0x0002	2	0000 0000 0000 0010
slip-p	=	0x0004	4	0000 0000 0000 0100
tension-p	=	0x0008	8	0000 0000 0000 1000
unused	=	0x0010	16	0000 0000 0001 0000
unused	=	0x0020	32	0000 0000 0010 0000
unused	=	0x0040	64	0000 0000 0100 0000
unused	=	0x0080	128	0000 0000 1000 0000
unused	=	0x0100	256	0000 0001 0000 0000
unused	=	0x0200	512	0000 0010 0000 0000
unused	=	0x0400	1024	0000 0100 0000 0000
unused	=	0x0800	2048	0000 1000 0000 0000
unused	=	0x1000	4096	0001 0000 0000 0000
unused	=	0x2000	8192	0010 0000 0000 0000
unused	=	0x4000	16384	0100 0000 0000 0000
unused	=	0x8000	32768	1000 0000 0000 0000

For user-defined constitutive models, the user can create a named state and assign any particular bit for that state, and subsequently update the contact state indicator variable. The named states in Table 4.2 are used by built-in constitutive models to update the failure states of contacts. If users use the state indicator variable to indicate failure states in their created model, they should make sure that there is no conflict with failure state constants of built-in models if they plan to use both of them in an analysis.

The code calls the constitutive model function **`Run()`** for each contact to update its force values. Typically, the state indicator is also updated in this process by the constitutive model. For built-in models, the state indicator denotes the failure state of the contact. This is updated by the constitutive model using the logical “or” (|) operation with the contact state indicator variable and the current failure state calculated by the constitutive model. The user should be certain to appropriately set or un-set all previous states updated prior to the current state calculated by the constitutive model. The state of a contact can then be checked using the logical “and” (&) operator with the state variable and desired user-defined state.

4.2.2.3 Implementation

Example Constitutive Model

Here we extract, for illustration, parts of the Mohr-Coulomb elastic/plastic model contained in files "Jmodelexample.*". Example 4.8 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 to change or print out.

Example 4.8 Class specification for model: file "Jmodelexample.h"

```
#pragma once

#include "../src/jointmodel.h"

namespace jmodels
{
  class JModelExample : public JointModel {
  public:
    JModelExample();
    virtual String  getName() const;
    virtual String  getPluginName() const { return getName(); }
    virtual String  getFullName() const;
    virtual UInt    getMinorVersion() const;
    virtual String  getProperties() const;
    virtual String  getStates() const;
    virtual Variant getProperty(UInt index) const;
    virtual void    setProperty(UInt index,const Variant &p,
                                UInt restoreVersion=0);
    virtual JModelExample *clone()
                           const { return new JModelExample(); }
    virtual Double  getMaxNormalStiffness() const { return kn_; }
    virtual Double  getMaxShearStiffness() const { return ks_; }
    virtual void    copy(const JointModel *mod);
    // If !isValid(dim) calls initialize(dim,s)
    virtual void    run(UByte dim,State *s);
    // calls setValid(dim)
    virtual void    initialize(UByte dim,State *s);
    // Optional
    virtual Double  getStressStrengthRatio(const Double &,
                                           const DVect3 &)
                                           const { return 10.0; }
    virtual void    scaleProperties(const Double &,
                       const std::vector<UInt> &)
```

```
                    { throw std::exception
                      ("Does not support property scaling"); }
    virtual bool    supportsStressStrengthRatio() const
                                          { return false; }
    virtual bool    supportsPropertyScaling() const
                                          { return false; }
  private:
    Double kn_;
    Double ks_;
    Double cohesion_;
    Double friction_;
    Double dilation_;
    Double tension_;
    Double zero_dilation_;
    Double res_cohesion_;
    Double res_friction_;
    Double res_dilation_;
    Double res_tension_;
    Double tan_friction_;
    Double tan_dilation_;
    Double tan_res_friction_;
    Double tan_res_dilation_;
    Int    kn_tab_;
    Int    ks_tab_;
  };
} // namespace models
```

Example 4.9 provides the constant definitions used by the model as well as the global instantiation of the model, as discussed in Section 4.2.2.2.

Example 4.9 Constant definition for Mohr-Coulomb model

```
static const Double dPi=3.14159265358979323846264338327950288419716939;
static const Double dDegRad = dPi / 180.0;
```

The constructor for this model was listed in Example 4.6. Example 4.10 provides listings of the member functions for initialization and execution ("running"). Also note the use of the **`State`** structure in providing displacement increments and forces. 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 this example, the 2D section is identical to the 3D section. Please refer to the file "jmodelexample.cpp" for listings of member functions **`Properties`**, **`States`**, **`GetProperties`**, **`SetProperties`**, **`Copy`** and **`SaveRestore`**.

Example 4.10 Initialization and execution sections of the Mohr-Coulomb model

```
/***************************** Initialize ***************************/
  void JModelExample::initialize(UByte dim,State *s)
  {
    JointModel::initialize(dim,s);
    tan_friction_    = tan(friction_ * dDegRad);
    tan_res_friction_ = tan(res_friction_ * dDegRad);
    tan_dilation_    = tan(dilation_ * dDegRad);
    tan_res_dilation_ = tan(res_dilation_ * dDegRad);
  }

  void JModelExample::run(UByte dim,State *s)
  {
    JointModel::run(dim,s);
    /* --- state indicator:                                    */
    /*     store 'now' info. as 'past' and turn 'now' info off ---*/
    if (s->state_ & slip_now) s->state_ ^= slip_past;
    s->state_ &= ~slip_now;
    if (s->state_ & tension_now) s->state_ ^= tension_past;
    s->state_ &= ~tension_now;

    Double kna  = kn_ * s->area_;
    Double ksa  = ks_ * s->area_;

    // normal force
    Double fn0 = s->normal_force_;
    s->normal_force_inc_ = -kna * s->normal_disp_inc_;
    s->normal_force_ += s->normal_force_inc_;

    // correction for time step in which joint opens
    //   (or goes into tension)
    // s->dnop_ is part of s->normal_disp_inc_ at
    //   which separation or tension takes place
    s->dnop_ = s->normal_disp_inc_;
    if ((fn0 > 0.0)               &&
        (s->normal_force_ <= 0.0) &&
        (s->normal_force_inc_ < 0.0))
    {
        s->dnop_ = -s->normal_disp_inc_ * fn0 / s->normal_force_inc_;
        if (s->dnop_ > s->normal_disp_inc_) s->dnop_ =
           s->normal_disp_inc_;
    }

    // tensile strength
```

```
Double ten;
if (s->state_)
  ten = -res_tension_ * s->area_;
else
  ten = -tension_ * s->area_;

// check tensile failure
bool tenflag = false;
if (s->normal_force_ <= ten)
{
  s->normal_force_  = ten;
  if (!s->normal_force_)
  {
    s->shear_force_ = DVect3(0,0,0);
    tenflag = true; // complete tensile failure
  }
  s->state_ ¯ tension_now;
  s->normal_force_inc_ = 0.0;
  s->shear_force_inc_ = DVect3(0,0,0);
}

// shear force
if (!tenflag)
{
  s->shear_force_inc_ = s->shear_disp_inc_ * -ksa;
  s->shear_force_ += s->shear_force_inc_;
  Double fsm = s->shear_force_.mag();
  // shear strength
  Double fsmax;
  if (!s->state_)
    fsmax=cohesion_ * s->area_ + tan_friction_ * s->normal_force_;
  else
  { // if residual friction is zero, take peak value
    Double resamueff = tan_res_friction_;
    if (!resamueff) resamueff = tan_friction_;
    fsmax=res_cohesion_ * s->area_ + resamueff * s->normal_force_;
  }
  if (fsmax < 0.0) fsmax = 0.0;
  //  check for slip
  if (fsm >= fsmax)
  {
    Double rat = 0.0;
    if (fsm) rat = fsmax / fsm;
    s->shear_force_ *= rat;
    s->state_ ¯ slip_now;
    s->shear_force_inc_ = DVect3(0,0,0);
```

```
          // dilation
          if (dilation_)
          {
            Double zdd = zero_dilation_;
            Double usm = s->shear_disp_.mag();
            if (!zdd) zdd = 1e20;
            if (usm < zdd)
            {
              Double dusm  = s->shear_disp_inc_.mag();
              Double dil = 0.0;
              if (!s->state_) dil = tan_dilation_;
              else
              {
                // if residual dilation is zero, take peak value
                //     Double resdileff = tan_res_dilation_;
                Double resdileff = tan_dilation_;
                if (!resdileff) resdileff = tan_dilation_;
                dil = resdileff;
              }
              s->normal_force_ += kna * dil * dusm;
            }
          } // dilation
        } // fsm>fsmax
      } // if (!tenflg)
    }

  } // namespace models
```

4.3 References

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