

4 WRITING NEW USER-DEFINED CONTACT MODELS

4.1 Constitutive-Model Support in *PFC^{2D}*

User-defined contact models are written in the C++ language, and compiled as a DLL file (dynamic link library) that can be loaded whenever it is needed.

The C++ language makes it easy to add a new contact model, compiled as a DLL (Dynamic Link Library). Using the “virtual function” support provided by C++, it is possible to have a user-written function install itself into a model library at runtime. The file containing the function is completely self-contained — the model name, property names and local data structures are all encapsulated in the file. However, once the model is installed, the host program (*PFC^{2D}*, in this case) can refer to the specific model names in commands such as **MODEL**, **PROPERTY** and **PRINT**. Further, the model and its associated properties can be manipulated by *FISH* in a way similar to the way in which the built-in properties are available to *FISH* functions.

The methodology of writing a contact model using C++ language as a form of DLL is described in [Section 4.2](#). *FISH* support functions and existing user-defined models are described in [Sections 4.3](#) and [4.4](#), respectively. Note that the DLL must be compiled using Microsoft Visual C++ (VC++) for operation in *PFC^{2D}*. The C++ source files provided with this version of *PFC^{2D}* are compiled using VC++ Version 6.0.

As an example in using the new logic, consider first a typical *PFC^{2D}* data file to perform model setup and execution — see [Example 4.1](#), contained in file “CMTEST1.DAT.”

Example 4.1 3-disk model, using built-in contact models

```

;fname: cmtest1.DAT (Use built-in, linear contact model.)
new
set random
set disk on
def setup ; contact properties
  nstiff = 1.0           ; Stiffnesses for user-defined contact model
  sstiff = 1.0
  mfric = 0.1
  nst2  = nstiff * 2.0   ; Stiffnesses for built-in model
  sst2  = sstiff * 2.0
end
setup
ball id=1 x 0.5 y 0.5    rad=0.5 ; make 3 balls equilateral triangle
ball id=2 x 1.5 y 0.5    rad=0.5
ball id=3 x 1.0 y 1.3660254 rad=0.5
prop dens 1.0
wall id=1 kn 2 ks 2 fric .1 nodes (0.0,0.0) (2.1,0.0)
wall id=2 kn 2 ks 2 fric .1 nodes (0.0,2.0) (0.0,0.0)
wall id=3 kn 2 ks 2 fric .1 nodes (2.1,0.0) (2.1,2.0)

```

```

wall id=4 kn 2 ks 2 fric .1 nodes (2.1,2.0) (0.0,2.0)
; =====
prop kn=nst2 ks=sst2 dens 1 fric=mfric ; assign ball stiffnesses for
; built-in linear contact model
; =====
plot add ball yellow angle=on
plot add wall black
plot add cforce green
wall id=4 yvel=-0.001 ; compress these 3 balls
cycle 700
wall id=4 yvel=0.0 ; achieve static equilibrium
cycle 500
plot show
print contact force
;EOF: cctest1.DAT

```

By executing this data file, a compacted assembly of 3 disks is obtained. The built-in linear contact model is invoked in this example, with slip, separation, and contact conditions arising during the run. Using the external contact model documented in [Section 4.4.1](#), identical behavior can be observed when the data file (“CMTEST2.DAT”) of [Example 4.2](#) is executed. Much of the data file is identical, but the DLL for the new contact model is loaded with the **MODEL LOAD** command, and the new contact model is invoked with the **MODEL** command; this overrides the built-in contact model and attaches a contact-model object to each existing contact. The properties associated with the user-written model are specified with the **PROPERTY** command (as usual), but the names and values relate to *contacts*, not to particles. This difference necessitates that contact stiffnesses for the user-written model are double those of the built-in model, in order to reproduce the latter’s results. The built-in model associates stiffnesses with particles. When two particles come into contact, the two stiffnesses are assumed to act *in series*, so that the common contact inherits half the stiffness of either ball (assuming that they are equal). However, stiffnesses specified for a user-written model are associated directly with contacts. One further consideration when using user-written models is the way in which the models are installed in *new contacts*. By default, new contacts will acquire the built-in contact model. In [Example 4.2](#), a **fishcall** is set up so that the creation of a new contact will automatically trigger the calling of *FISH* function **catch_contact**, which installs the user-defined model and its properties dynamically. See [Section 4.3](#) for a description of the two *FISH* intrinsic functions, **c_model()** and **c_prop()**. Apart from the command **MODEL**, one other command relates to user-defined models; the command **PRINT contact prop** lists the contact models associated with each contact and the values of contact properties.

Example 4.2 3-disk model, using the external contact model

```

;fname: cctest2.DAT (Use user-defined "cml" contact model.)
new
set random
set disk on

```

```

def setup ; contact properties
  nstiff = 1.0          ; Stiffnesses for user-defined contact model
  sstiff = 1.0
  mfric  = 0.1
  nst2   = nstiff * 2.0 ; Stiffnesses for built-in model
  sst2   = sstiff * 2.0
end
setup
ball id=1 x 0.5 y 0.5      rad=0.5 ; make 3 balls equilateral triangle
ball id=2 x 1.5 y 0.5      rad=0.5
ball id=3 x 1.0 y 1.3660254 rad=0.5
prop dens 1.0
wall id=1 kn 2 ks 2 fric .1 nodes (0.0,0.0) (2.1,0.0)
wall id=2 kn 2 ks 2 fric .1 nodes (0.0,2.0) (0.0,0.0)
wall id=3 kn 2 ks 2 fric .1 nodes (2.1,0.0) (2.1,2.0)
wall id=4 kn 2 ks 2 fric .1 nodes (2.1,2.0) (0.0,2.0)
; =====
call fishcall.FIS
def catch_contact          ; Called whenever new contact made
  count = count + 1       ; Count new contacts, for interest
  cp    = fc_arg(0)
  c_model(cp) = 'cml'
  c_prop(cp,'cml_kn')    = nstiff
  c_prop(cp,'cml_ks')    = sstiff
  c_prop(cp,'cml_fric') = mfric
end
model load cml2wrv ; PFC2D release version
model cml          ; Convert existing contacts
prop cml_kn=nstiff cml_ks=sstiff cml_fric=mfric ; to "cml" contact model
set fishcall FC_CONT_CREATE catch_contact      ; Trap new contacts
; =====
plot add ball yellow angle=on
plot add wall black
plot add cforce green
wall id=4 yvel=-0.001 ; compress these 3 balls
cycle 700
wall id=4 yvel=0.0    ; achieve static equilibrium
cycle 500
plot show
print contact force
;EOF: cmtest2.DAT

```

4.2 Implementation of User-Defined Contact Models in *PFC^{2D}*

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 simple introduction to programming in C++ is provided by Stevens (1994); it is assumed that the reader has a working knowledge of the language.

4.2.1 Base Class for Contact Models

The methodology described above is exploited in *PFC^{2D}*’s support for user-defined contact models. A base class provides a framework for actual contact models, which are classes derived from the base class. The base class, called **ContactModel**, 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 **ContactModel** can be created, and that any derived-class object *must* supply real member functions to replace each one of the member functions of **ContactModel**. [Example 4.3](#) provides a partial listing of **ContactModel**, which is contained in file “CONTMODEL.H.” Some member functions of **ContactModel** are omitted from the listing in [Example 4.3](#). These functions are used by *PFC^{2D}* to manipulate and interrogate contact models; there is no reason for user-written model to use or redefine these.

Example 4.3 Class definition for base class, **Cmodel**

```
// fname: Cmodel.txt
class ContactModel {
private:
    unsigned long ulType;
protected:
    bool    delete_flag;    // true if contact can be deleted
public:
    // Creators
    EXPORT ContactModel(unsigned long ulTypeIn, bool bRegister=false);
    EXPORT virtual ~ContactModel(void);
    // Accessors
    EXPORT virtual const char *Name(void) const=0;
    EXPORT virtual const char **PropNames(void) const=0;
    EXPORT virtual ContactModel *Clone(PropBlock *pb=0) const=0;
    EXPORT virtual double ReturnProp(int) const=0;
```

```
EXPORT virtual double KsEstimate(void) const=0;
EXPORT virtual double KnEstimate(void) const=0;
EXPORT virtual unsigned long Version(void) const = 0;
EXPORT unsigned long Type(void) const { return ulType; }
EXPORT bool OKToDelete(void) { return delete_flag; }
// Manipulators
EXPORT virtual void AcceptProp(int,double)=0;
EXPORT virtual const char *PreCycle(PropBlock &pb)=0;
EXPORT virtual void FDLaw(FdBlock &fb)=0;
void SetDelete(bool d) { delete_flag = d; }
// Save & Restore
EXPORT virtual const char *SaveRestore(ModelSaveObject *mso);
};
```

4.2.2 Member Functions

Any derived contact-model class must provide the constructor, the destructor and concrete functions to replace the virtual member functions in **ContactModel**. These functions perform the operations described below.

- const char *Name()** A pointer is returned to a character array containing the name of the contact model. For example, {"**cm1**"} would be a valid string in C++.
- const char **PropNames()** A pointer is returned to character arrays containing the names of model properties. The following string is a valid example: {"**cm1_kn**", "**cm1_ks**", "**cm1_fric**", 0}. The given names will be those recognized by the **PROPERTY** command. Note: The array of strings should be terminated by 0, as shown.
- ContactModel *Clone(PropBlock *pb=0)** A new object is created of the same class as the current object, and a pointer to it, of type **ContactModel**, is returned. The argument to **Clone** must be a pointer of structure **PropBlock pb**, which is initialized zero. (See [Section 4.2.4](#) for explanations of *PFC^{2D}*'s classes and structures.)
- double ReturnProp(int n)** A value is returned for the model property of sequence number *n* (previously defined by a **PropNames ()** call).
- double KsEstimate()** The model object must return a value for its best estimate of the current tangent shear contact-stiffness. Currently it is used by *PFC^{2D}* if there is no other source of information about stiffnesses, in order to estimate an initial timestep.
- double KnEstimate()** The model object must return a value for its best estimate of the current tangent normal contact-stiffness. Currently it is used by *PFC^{2D}* if there is no other source of information about stiffnesses, in order to estimate an initial timestep.
- unsigned long Version()** The version number of the contact 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.
- bool OKToDelete()** A *PFC^{2D}* contact is normally deleted when the distance between its two constituent particles exceeds some internally-defined limit. Deletion may be prevented by setting a model flag; the state of the flag can be tested with this function.

void AcceptProp(int n, double v)

The value of **v** supplied by the call comes from a *PFC^{2D}* command of the form **PROP name=v**; the supplied value of **n** is the sequence number of the property name previously specified by means of a **PropNames()** call. The model object is required to store the supplied value in its appropriate local memory location.

void PreCycle(PropBlock &pb)

This function is called for each model object when the *PFC^{2D}* **CYCLE** command is given, with a reference to structure **PropBlock pb**, the associated contact. The model object may perform initialization, or it may do nothing.

void Fdlaw(FdBlock &fb)

This function is called for each model object at each cycle from within *PFC^{2D}*'s contact scan. The model must update contact forces, based on contact overlap and relative contact velocities. The structure **fb** contains the current overlap and local contact velocities. The velocity components provided by **fb** — rather than the velocities of the particles — should be used since they already incorporate the contributions of particle spins and any rotation-correction terms.

SetDelete(bool d)

The delete-flag (see above discussion) may be set. The following meanings are attached to **d**: if **d=bTrue**, the contact can be deleted, at the discretion of *PFC^{2D}*. Otherwise it will not be deleted, even if the gap is very large.

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 real variables, integers and boolean values are accepted, so other data types must be converted to those. The derived-class function must call **mso->Initialize(nd,ni,nb)**, where **nd**, **ni** and **nb** are the number of **doubles**, **ints** and **bools** to be saved/restored, respectively. 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**, 0 to **ni-1** or 0 to **nb-1**, depending on whether

reals, integers and bools are being saved/restored), and **var** is the variable to be saved. There are separate **Save()** functions for processing **double**, **int** or **bool** variables. The structure of the **ModelSaveObject** class is irrelevant, except for the use of the member functions mentioned. It is defined in "CONTMODEL.H."

4.2.3 Derived Class for Contact Model

An example of the declaration of a user-defined contact model is listed in [Example 4.4](#).

Example 4.4 Class declaration for user-defined class, **CExample**

```
//fname: CExample.txt
#ifndef __UDM_EXAMPLE_H
#define __UDM_EXAMPLE_H

#include "contmodel.h"

----- Example Model -----
const unsigned long ulModelCMEEx = 100;
class CExample : public ContactModel {
private:
    double kn;
    double ks;
    double fric;
public:
    // Creators
    EXPORT CExample(Bool bRegister=bFalse);
    EXPORT ~CExample(void) { }
    // Accessors
    EXPORT const char *Name(void) const;
    EXPORT const char *PropNames(void) const;
    EXPORT ContactModel *Clone(PropBlock *pb=0) const {return
        (new CExample);}
    EXPORT double ReturnProp(int n) const;
    EXPORT double KsEstimate(void) const { return ks; }
    EXPORT double KnEstimate(void) const { return kn; }
    EXPORT unsigned long Version(void) const { return 1; }
    // Manipulators
    EXPORT void AcceptProp(int n, double v);
    EXPORT const char *PreCycle(PropBlock &) { return(0); }
    EXPORT void FDLaw(FdBlock &);
    EXPORT const char *SaveRestore(ModelSaveObject *);
```

```
};
#endif
```

The derived class must be assigned a unique numeric identifier used during the save/restore process. 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. The properties of the constitutive model are private members of the class. Examples of the definitions of the member functions of a user-written class are shown in [Example 4.5](#).

Example 4.5 *Definitions of user-defined class member functions*

```
// fname: MyExamp.txt
CExample::CExample(bool bRegister) :
    ContactModel(ulModelCMEEx, bRegister),
                kn(0.0), ks(0.0), fric(0.0) {
    SetDelete(true);
}

const char *CExample::Name(void) const {
    return("cex");
}

const char **CExample::PropNames(void) const {
    static const char *strKey[] = {
        "cex_kn", "cex_ks", "cex_fric",
        0
    };
    return(strKey);
}

double CExample::ReturnProp(int n) const {
    switch (n) {
        case 0: return kn;
        case 1: return ks;
        case 2: return fric;
        default: return 0.0;
    }
}

void CExample::AcceptProp(int n, double v) {
```

```

switch (n) {
  case 0:  kn = v;    break;
  case 1:  ks = v;    break;
  case 2:  fric = v;  break;
  default:                break;
}
}

//----- user-defined contact model "cexample" -----
void CExample::FDlaw(FdBlock& fb) {
#if DIM == 2 // 2D Section
  if (fb.u_n > 0.0) {
    fb.n_force = kn * fb.u_n;
    fb.s_force -= ks * fb.u_dot_s * fb.tdel;
    double max_s_force = fb.n_force * fric;
    if (fabs(fb.s_force) > max_s_force)
      fb.s_force = fb.s_force < 0.0 ? -fabs(max_s_force):fabs(max_s_force);
    fb.knest = kn;
    fb.ksest = ks;
    fb.skip = false;
  } else {
    fb.n_force = 0.0;
    fb.s_force = 0.0;
    fb.knest = 0.0;
    fb.ksest = 0.0;
    fb.skip = true;
  }
}
#elif DIM == 3 // ... 3D section ...
  if (fb.u_n > 0.0) {
    fb.n_force = kn * fb.u_n;
    double kst = ks * fb.tdel;
    _Dvect vec = fb_u_dot_s * kst;
    fb_s_force= fb_s_force-vec;
    double max_s_force = fb.n_force * fric; // Slip check
    double dX = fb.pts_force->x;
    double dY = fb.pts_force->y;
    double dZ = fb.pts_force->z;
    double sfmag = sqrt(dX*dX + dY*dY + dZ*dZ);
    if (sfmag > max_s_force) {
      double rat = max_s_force / sfmag;
      fb_s_force= fb_s_force* rat;
    }
    fb.knest = kn;
    fb.ksest = ks;
  }
}

```

```

        fb.skip = false;
    } else {
        fb.n_force = 0.0;
        fb.s_force.Fill(0.0);
        fb.knest = 0.0;
        fb.ksest = 0.0;
        fb.skip = true;
    }
}
#endif

const char *CExample::SaveRestore(ModelSaveObject *mso) {
    const char *str = ContactModel::SaveRestore(mso);
    if (str) return(str);
    mso->Initialize(3,0,1);
    mso->Save( 0, delete_flag);
    mso->Save( 0, kn);
    mso->Save( 1, ks);
    mso->Save( 2, fric);
    return(0);
}

/* EoF */

```

A typical example of the constructor of the user-written class is given by the following code:

```

CExample::CExample(bool bRegister) :
    ContactModel(ulModelCMEEx, bRegister),
                kn(0.0), ks(0.0), fric(0.0) {
    SetDelete(true);
}

```

The constructor is used to initialize constitutive properties of the contact model and to register the new constitutive model into *PFC^{2D}*'s model registry. Therefore, static objects of the user's model must be instantiated in order to force a class constructor to be called when the code is loaded. In this way, *PFC^{2D}* adds the new model to its list of recognized models at runtime. The following listing illustrates the mechanism by which the user-written model becomes recognized by *PFC^{2D}*. Note that the **true** argument signifies that the object is to be registered in the library of the contact models.

```

static CExample cexample(true);

```

Also note the following points.

1. The sequence number n , used in functions `void AcceptProp(int n, double v)` and `double ReturnProp(int n)`, must coincide with the sequence number of the property name in the string of the model properties used in the function `char **PropNames()`. (The number n is zero-based.)
2. In the function `FDlaw`, separate sections are provided for 2D and 3D operation, using the system-defined macro `DIM` (equal to 2 or 3, for PFC^{2D} or PFC^{3D} , respectively).

4.2.4 PFC^{2D} Global Variables, Classes and Structures

The header files “CONTMODEL.H,” and “EXDVECT3.H,” contain some global variables, function prototypes and `#defined` symbols that can be accessed by a user-written contact model. These headers should be consulted for general information about their contents — in most cases, there are explanatory comments. A list of some of the more-common global names follows.

DIM	2 for PFC^{2D} , and 3 for PFC^{3D}
_Dvect	vector type, class <code>exDvector3</code> consisting of <code>Doubles</code> used only in a 3D compilation (<code>DIM=3</code>)
ContactModel	base class for contact models (in “CONTMODEL.H”)
FdBlock	structure used to pass information between a contact model and PFC^{2D} during cycling (in “CONTMODEL.H”). Section 4.2.5 provides a list of members.
PropBlock	structure used to pass information between a contact model and PFC^{2D} when the model is created or before cycling (in “CONTMODEL.H”). Section 4.2.5 provides a list of members.
fb_u_dot_s	macro representing <code>(*fb.ptu_dot_s)</code> — used only in a 3D compilation (<code>DIM=3</code>)
fb_t_dot_rel	macro representing <code>(*fb.pttu_dot_rel)</code> — used only in a 3D compilation (<code>DIM=3</code>)
fb_s_force	macro representing <code>(*fb.pts_force)</code> — used only in a 3D compilation (<code>DIM=3</code>)
fb_moment	macro representing <code>(*fb.ptmoment)</code> — used only in a 3D compilation (<code>DIM=3</code>)

4.2.5 Parameters Passed between Model and PFC^{2D}

The most important link between *PFC^{2D}* and a user-written model is the member-function **FD-law()**, which computes the mechanical response of the model during cycling. A structure, **FdBlock** (defined in “CONTMODEL.H”), is used to transfer information to the model. The members of **FdBlock** are as follows.

double u_n	contact overlap (positive; negative, for a gap) — supplied by <i>PFC^{2D}</i>
double u_dot_n	relative normal velocity — supplied by <i>PFC^{2D}</i>
double u_dot_s	relative shear contact-velocity, used only in a 2D compilation (DIM=2) — supplied by <i>PFC^{2D}</i>
double t_dot_rel	relative angular contact-velocity, used only in a 2D compilation (DIM=2) — supplied by <i>PFC^{2D}</i>
_Dvect *ptu_dot_s	pointer of _Dvect variable, which contains components of the vector of the relative shear contact-velocity, used only in a 3D compilation (DIM=3) — supplied by <i>PFC^{2D}</i>
_Dvect *ptt_dot_rel	pointer of _Dvect variable, which contains components of the vector of the relative angular contact-velocity, used only in a 3D compilation (DIM=3) — supplied by <i>PFC^{2D}</i>
double n_force	normal force at contact
double s_force	shear force at contact, used only in a 2D compilation (DIM=2)
double moment	moment at contact, used only in a 2D compilation (DIM=2) — calculated by a model
_Dvect *pts_force	pointer of _Dvect variable, which contains components of the vector of the shear force at contact, used only in a 3D compilation (DIM=3)
_Dvect *ptmoment	pointer of _Dvect variable, which contains components of the vector of the moment at contact, used only in a 3D compilation (DIM=3) — calculated by a model
double tdel	the timestep — supplied by <i>PFC^{2D}</i>
double knest	estimated tangent normal-stiffness. The model must return a value for knest , since it is used at each step to compute the next stable timestep.
double ksest	estimated tangent shear-stiffness. The model must return a value for ksest , since it is used at each step to compute the next stable timestep.
double krest	estimated rotational-stiffness. The model must return a value for krest , since it is used at each step to compute the next stable timestep.

bool skip	specified by the model. If false , a normal contact calculation is performed. If true , the entire calculation of the contact is skipped on return from FDlaw() ; no information is updated. The latter mode is usually selected if the contact model is inactive (e.g., there is a gap), thus saving computation time.
bool bfishc_brokenn	specified by the model. If true , the fishcall FC_BOND_DEL invokes (fc_arg(1)=0) in <i>PFC^{2D}</i> . (Default is false .)
bool bfishc_brokens	specified by the model. If true , the fishcall FC_BOND_DEL invokes (fc_arg(1)=1) in <i>PFC^{2D}</i> . (Default is false .)
bool bflag	flag used to check the status of contact bond in a contact. If true , the contact bond exists in <i>PFC^{2D}</i> . The users need to specify the value when the existing contact bond is examined in their user-defined models. (Default is false .)
bool broken	flag used to check the status of contact bond. If true , the contact bond is broken in <i>PFC^{2D}</i> . The users need to specify the value when the existing contact bond is examined in their user-defined models. (Default is false .)

Also, a structure, **PropBlock** (defined in “CONTMODEL.H”), is used to communicate between *PFC^{2D}* and a user-written model, before cycling or the model is created using **PreCycle()** and **Clone()**. The members of **PropBlock** are as follows; these are all supplied by *PFC^{2D}*.

double kn_c	normal stiffness of default contact model
double ks_c	shear stiffness of default contact model
double fric_c	friction coefficient of default contact model
double n_strength	normal strength of contact bond, default contact model
double s_strength	shear strength of contact bond, default contact model
unsigned char type_gobj2	object type of 2nd object of a contact (BALL:100, WALL:101)
bool blsNear	status of contact, true if two objects are close enough to be considered valid for bond creation.

4.2.6 Using and Creating User-Written Model DLLs

DLL files may be loaded into *PFC^{2D}* 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 *PFC^{2D}* 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 — “2D_CEXAMPLE.DSW.” It contains a project called “2D_CEXAMPLE.DSP” that contains example source and header files called “CEXAMPLE.CPP” and “CEXAMPLE.H,” which are described in [Section 4.2.3](#). The user may modify these files, or delete these files and include new ones to create a new DLL. The user should also 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.

The user-defined models depend on the following files.

1. CONTMODEL.H — utility structure used to communicate with contact model
2. EXDVECT3.H — specifies a **double**s vectors class **_Dvect**, used only in a 3D compilation (**DIM=3**)

These header files should also be included in the project. These files get all unresolved definitions from “CMODEL2W(R or D)V.DLL,” to which they are linked through the import library “CMODEL2W(R or D)V.LIB.” Note that R is for release version, and D is for debug version in the file name. The resulting DLL will be placed in the “.\Release” or “.\Debug” directory, which you specified the settings. The default name of the DLL is “CEXAMPLE2W(R or D)V.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 “CEXAMPLE2W(R or D)V.DLL” is included. This model is identical to the built-in linear contact model.

4.3 FISH Support for Contact Models

The following *FISH* intrinsics exist in *PFC^{2D}*.

```
c_prop(cp,p_name)
```

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

```
val = c_prop(cp,p_name)
```

stores in **val** the floating-point value of the property named **p_name** in contact **cp**. If there is no contact model in **cp**, or the model does not possess the named property, then 0.0 is returned. Similarly,

```
c_prop(cp,p_name) = val
```

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

```
c_model(cp)
```

The action of this intrinsic depends on context. If used in the following way,

```
val = c_model(cp)
```

the returned variable **val** will be a string containing the name of the model installed in contact **cp**. Otherwise, an integer variable zero will be returned, denoting that no contact model is present. (The user's *FISH* function must check for the variable type, before doing a comparison.) If used in the following way,

```
c_model(cp) = val
```

the string contained in **val** will be taken as a model name, and the corresponding contact model *will be installed* in the contact, erasing whatever was there before. If **val** does not contain a valid model name, nothing will be done and no error condition will be triggered. The contact-model member function **Clone()** is called immediately after the model object is constructed.

4.4 Existing User Models

Several user contact models are already implemented as DLLs besides the default contact model. A brief description of these models is provided in this section.

4.4.1 A Simple Frictional Model

A simple frictional model can be compared directly with the existing model in *PFC^{2D}*, the formulation of which is provided in [Section 2](#) in **Theory and Background**. (The general solution methodology is provided in [Section 1.1](#) in **Theory and Background**.)

The normal force is derived directly from the overlap, but the shear force calculation is incremental. Using the same nomenclature as in [Section 2.1](#) in **Theory and Background**, the elastic relations are

$$F_i^n = K^n U^n n_i \quad (4.1)$$

$$\Delta F_i^s = -k^s \Delta U_i^s \quad (4.2)$$

The contact is checked for slip conditions by calculating the maximum allowable shear contact force, F_{\max}^s :

$$F_{\max}^s = \mu |F_i^n| \quad (4.3)$$

If $|F_i^s| > F_{\max}^s$, then slip is allowed to occur by setting the magnitude of F_i^s equal to F_{\max}^s , via

$$F_i^s := F_i^s (F_{\max}^s / |F_i^s|) \quad (4.4)$$

These equations are embodied in the coding of **FDlaw()**, provided in [Example 4.6](#).

The name of the model is **CM1**, and it has the following properties:

cm1_kn	normal stiffness
cm1_ks	shear stiffness
cm1_fric	friction coefficient

Example 4.6 Coding for a simple slip constitutive model

```

// fname: CM1.txt
void CM1::FDlaw(FdBlock& fb) {
#if DIM == 2 // 2D Section
    if (fb.u_n > 0.0) {
        fb.n_force = kn * fb.u_n;
        fb.s_force -= ks * fb.u_dot_s * fb.tdel;
        double max_s_force = fb.n_force * fric;
        if (fabs(fb.s_force) > max_s_force)
            fb.s_force = fb.s_force < 0.0 ? -fabs(max_s_force):fabs(max_s_force);
        fb.knest = kn;
        fb.ksest = ks;
        fb.skip = false;
    } else {
        fb.n_force = 0.0;
        fb.s_force = 0.0;
        fb.knest = 0.0;
        fb.ksest = 0.0;
        fb.skip = true;
    }
}
#elif DIM == 3 // ... 3D section ...
    if (fb.u_n > 0.0) {
        fb.n_force = kn * fb.u_n;
        double kst = ks * fb.tdel;
        _Dvect vec = fb_u_dot_s * kst;
        fb_s_force= fb_s_force- vec;
        double max_s_force = fb.n_force * fric; // Slip check
        double dX = fb.pts_force->x;
        double dY = fb.pts_force->y;
        double dZ = fb.pts_force->z;
        double sfmag = sqrt(dX*dX + dY*dY + dZ*dZ);
        if (sfmag > max_s_force) {
            double rat = max_s_force / sfmag;
            fb_s_force= fb_s_force* rat;
        }
        fb.knest = kn;
        fb.ksest = ks;
        fb.skip = false;
    } else {
        fb.n_force = 0.0;
        fb_s_force.Fill(0.0);
        fb.knest = 0.0;
        fb.ksest = 0.0;
    }
}

```

```

        fb.skip = true;
    }
}
#endif

```

4.4.2 A Simple Viscoelastic Model

Consider a simple viscoelastic model in which the shear behavior consists of a spring in series with a dashpot. The total shear velocity, \dot{u}_s , can be decomposed into elastic and viscous parts:

$$\dot{u}_s = \dot{u}_s^e + \dot{u}_s^v \quad (4.5)$$

For the two-dimensional case, the shear velocities and forces can be taken as scalars. Taking F_{s° and F'_{s° as the shear forces before and after one timestep, respectively, we can express the components of shear velocity as

$$\dot{u}_s^e = -\frac{F'_s - F_s^\circ}{k_s \Delta t} \quad (4.6)$$

and

$$\dot{u}_s^v = -\frac{F'_s + F_s^\circ}{2\eta} \quad (4.7)$$

where k_s is the shear stiffness and η is the viscosity. Substituting these expressions into [Eq. \(4.5\)](#) and rearranging,

$$F'_s = \frac{F_s^\circ \left(\frac{1}{k_s} - \frac{\Delta t}{2\eta} \right) - \dot{u}_s \Delta t}{\left(\frac{1}{k_s} + \frac{\Delta t}{2\eta} \right)} \quad (4.8)$$

For the three-dimensional case, the shear force and relative shear velocity are both vectors, and the constitutive equation becomes

$$F_i^{s'} = \frac{F_i^{s^\circ} \left(\frac{1}{k_s} - \frac{\Delta t}{2\eta} \right) - \dot{u}_i^s \Delta t}{\left(\frac{1}{k_s} + \frac{\Delta t}{2\eta} \right)} \quad (4.9)$$

Eqs. (4.8) and (4.9) are encoded into a contact model called **viscous**. The force-displacement function of the model is listed in [Example 4.7](#), for the two-dimensional section. The required properties are as follows.

vis_kn normal stiffness
vis_ks shear stiffness
vis_viscosity shear viscosity

Example 4.7 *Coding for a simple viscoelastic model — 2D only*

```
// fname: viscous.txt
void CM2::FDlaw(FdBlock &fb) { // Force/displacement law
    if (!bonded && bond_when_touch) {
        if (fb.u_n > 0.0) {
            bonded = true;
            SetDelete(false);
        }
    }
    #if DIM == 2 // ... 2D section ...
        if (bonded) {
            double c1 = fb.tdel * con2;
            double c2 = con1 - c1;
            double c3 = con1 + c1;
            fb.n_force = kn * fb.u_n;
            fb.s_force = (fb.s_force * c2 - fb.u_dot_s * fb.tdel) / c3;
            fb.knest = kn;
            fb.ksest = ks;
            fb.skip = false;
        } else {
            fb.knest = 0.0;
            fb.ksest = 0.0;
            fb.skip = true;
        }
    #elif DIM == 3 // ... 3D section ...

    #endif
}
```

4.4.3 A Simple Ductile Model

The contact model named **ductile** is intended to resemble the built-in contact model, but provide a user-specified softening slope in place of the brittle failure embodied in the contact bond.

The model has the following properties:

duc_kn	normal stiffness in compression
duc_ratks	ratio of shear stiffness to normal stiffness
duc_ratfs	ratio of maximum shear force to normal tensile force
duc_fric	friction coefficient (if bond broken)
duc_ftmax	tensile strength (a positive number)
duc_kduc	softening stiffness (a positive number)
duc_kun	normal stiffness in tension (a positive number; defaults to duc_kn)

In addition, the following “properties” record the state of the model but cannot be set by the user:

duc_softened	= $1 - T/\mathbf{ftmax}$, where T is the current (softened) tensile strength; duc_softened is equal to 0.0 if contact has never yielded
duc_broken	= 1.0 if the strength, T , has reduced to zero

When installed at a contact, the model is in its bonded state (i.e., **duc_broken** = 0.0). In this state, the contact may take tension and compression with no slip allowed, although the absolute shear force in tension is limited to a ratio of **duc_ratfs** to the absolute normal force. If the tensile normal force exceeds the tensile strength, the normal force is reduced according to the given softening stiffness (**duc_kduc**). This softening modulus is used whenever the contact displacement increment is extensile. For “unloading” increments in normal displacement (i.e., the two balls approach one another), the force-displacement path is a straight line passing through the origin and the current tensile strength. When the tensile strength reduces to zero, the contact becomes unbonded, and no further tension can be taken. Further, slip is allowed when the normal force becomes compressive. The magnitude of the shear force is not part of the yield criterion.

The fishcall **FC_BOND_DEL** is called in two circumstances: first, when the tensile strength is first exceeded (**fc_arg(1)=1**) and; second, when the bond is broken (**fc_arg(1)=0**). The value of **fc_arg(0)** is the contact pointer.

The force-displacement coding for the two-dimensional form of the simple ductile model is provided in [Example 4.8](#).

Example 4.8 Coding for a simple ductile model – 2D only

```

// fname: ductile.TXT
void CM3::FDlaw(FdBlock& fb) {
#if DIM == 2 // 2D Section
  if (fb.u_n > 0.0) { // COMPRESSIVE
    fb.n_force = kn * fb.u_n;
    double kst = kn * ratks * fb.tdel;
    fb.s_force = fb.s_force - fb.u_dot_s * kst;
    if (broken) {
      double max_s_force = fb.n_force * fric; // Slip check
      double sfmag = fabs(fb.s_force);
      if (sfmag) {
        if (sfmag > max_s_force) {
          double rat = max_s_force / sfmag;
          fb.s_force = fb.s_force * rat;
        }
      } else {
        fb.s_force = 0.0;
      }
    }
    fb.knest = kn;
    fb.ksest = kn * ratks;
    fb.skip = false;
  } else { // EXTENSILE
    if (broken) {
      fb.n_force = 0.0;
      fb.s_force = 0.0;
    } else {
      double Fn = kun * fb.u_n;
      if (Fn < FT) {
        FT = (ut - fb.u_n) * kduc - FTmax;
        if (FT > 0.0) {
          FT = 0.0;
          broken = 1.0;
          softened = 1.0;
          SetDelete(true);
          fb.bfishc_brokenn = true;
        }
      }
      kun = FT / fb.u_n;
      Fn = FT;
      if (FTmax) {
        softened = 1.0 + FT / FTmax;
      } else {
        softened = 1.0;
      }
    }
  }
}

```

```

    }
  }
  fb.n_force = Fn;
  if (broken == 0.0) {
    double kst      = kun * ratks * fb.tdel;
    fb.s_force      = fb.s_force - fb.u_dot_s * kst;
    double max_s_force = fabs(FT) * ratfs;
    double sfmag     = fabs(fb.s_force);
    if (sfmag) {
      if (sfmag > max_s_force) {
        double rat    = max_s_force / sfmag;
        fb.s_force = fb.s_force * rat;
      }
    } else {
      fb.s_force = 0.0;
    }
  } else {
    fb.s_force = 0.0;
  }
}
fb.knest = kun;
fb.ksest = kun * ratks;
fb.skip  = false;
}
}
#elif DIM == 3                // 3D section ...

#endif

```

4.4.4 A Displacement Softening Model

The contact model named **softening** is intended to be the general contact softening model. The model has the following properties:

- sof_knc** normal stiffness in compression
- sof_knt** normal stiffness in tension
- sof_ks** shear stiffness
- sof_ftmax** tensile strength, F_c^n (a positive number)
- sof_fsmax** shear strength, F_c^s (a positive number)
- sof_fric** friction coefficient

sof_rfric residual friction coefficient

sof_uplim accumulated plastic displacement for which the bond strength softens to zero (see Figure 4.1), U_{pmax}

sof_broken = 1.0 if the strengths, F_{c^k} ($k = n, s$), have reduced to zero

In addition, the following “properties” record the state of the model but cannot be set by the user:

sof_uplas currently accumulated plastic displacement (see Figure 4.1), U_p

sof_softened = U_p / U_{pmax} ; **softened** is equal to 0.0 if contact has never yielded

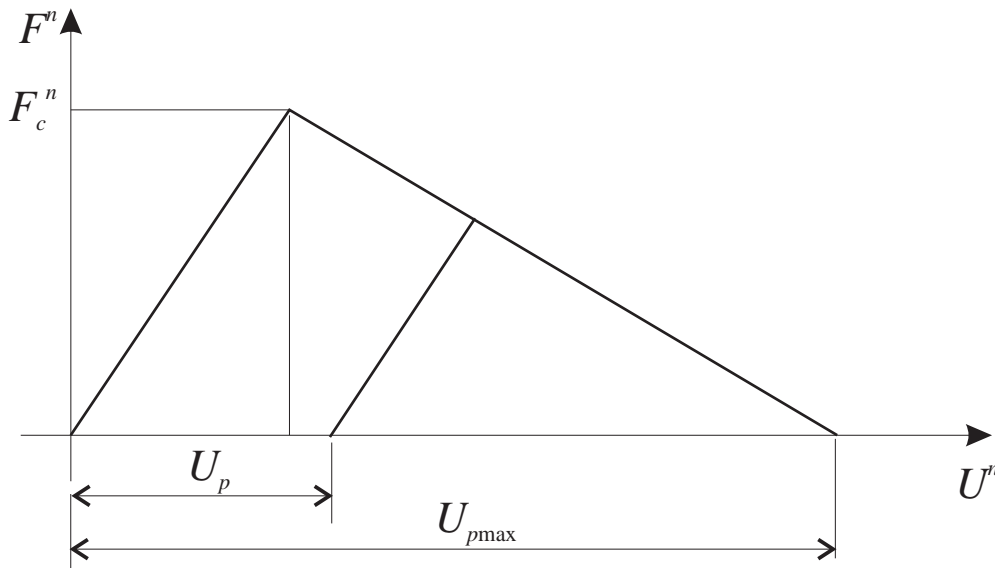


Figure 4.1 Strength-softening curve

If the contact is in tension, the contact strength, F_{max} , is calculated from the two strength parameters (i.e., F_{c^n} and F_{c^s}) as a function of the current orientation of the contact force. It is assumed that contact strength varies as a linear function of the angle α :

$$F_{max} = \left(1 - \frac{2\alpha}{\pi}\right) \cdot F_{c^n} + \frac{2\alpha}{\pi} \cdot F_{c^s}$$

where α is the angle between the directions of the contact force and the line segment connecting the centers of the balls. The yielding of the bond in tension is determined by comparing the resultant contact force — i.e.,

$$F = \sqrt{F_n^2 + F_s^2}$$

with the contact strength. The contact yields if the contact force is larger than the contact strength:

$$F > F_{\max}$$

In the case of yielding of contact bonds, the increment of contact displacements, ΔU^k ($k = n, s$), can be decomposed into elastic and plastic contact displacement increments:

$$\Delta U^k = \Delta U_e^k + \Delta U_p^k$$

The force increment, ΔF^k , is a function of the increment of the elastic displacement only — i.e.,

$$\Delta F^k = K^k \Delta U_e^k \tag{4.10}$$

where

$$\Delta U_e^k = \Delta U^k - \Delta U_p^k \tag{4.11}$$

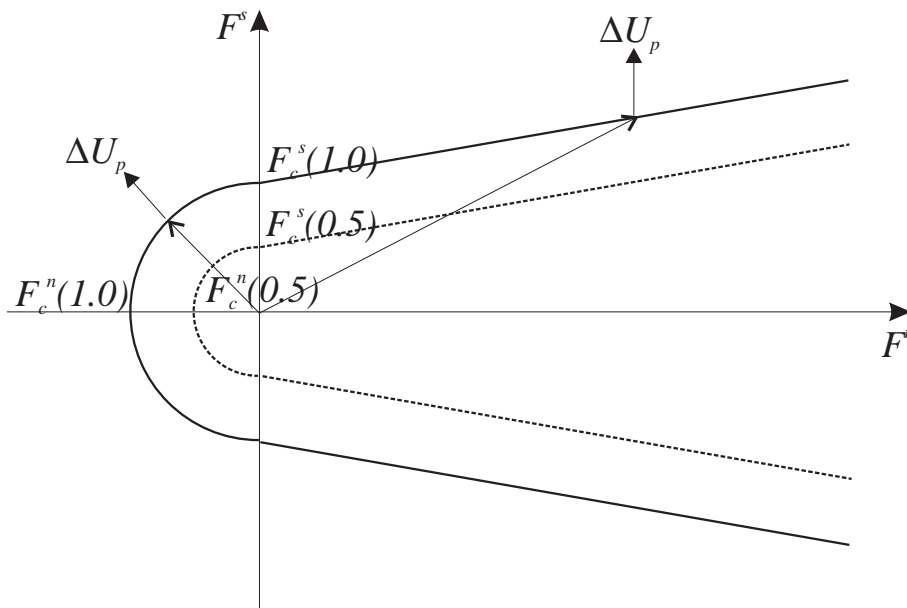


Figure 4.2 Contact yield condition

The elastic (or plastic) displacement increment can be determined using the consistency condition (i.e., $F - F_{\max} = 0$). The increment of the plastic displacements obeys the “flow rule.” It is

assumed that, if the normal force is tensile, the plastic displacement increment is always in the direction of the resultant contact force, as it is shown in [Figure 4.2](#):

$$\Delta U_p^k = \Delta\lambda \frac{F^k}{F} \quad (4.12)$$

where $\Delta\lambda$ is the plastic multiplier. The plastic multiplier can be determined from the consistency condition if the contact force is expressed using [Eqs. \(4.10\), \(4.11\) and \(4.12\)](#).

If the normal contact force is compressive, the maximum contact shear force is defined by the slip condition:

$$F_{\max}^s = \mu |F^n| + F_c^s$$

If slip takes place, the plastic shear displacement is assumed to be in the direction of the shear force (see [Figure 4.2](#)).

This is a softening model, and the contact strengths are functions of the accumulated plastic displacements:

$$F_c^k(U_p/U_{p\max}) = F_c^k \left(1 - \frac{U_p}{U_{p\max}} \right)$$

where $U_{-p} = \sum |\Delta U_{-p}|$.

The normal force in this model is not calculated directly from the overlap of the balls. Instead, the normal force is calculated incrementally, by adding the force increment (a function of the contact normal stiffness and the elastic displacement increment) to the current contact force. This modification has the following consequences:

1. The contact forces can be initialized at any stage of the simulation.
2. If this contact model is assigned to the contact involving balls that do not touch, or to the newly (during the simulation) created contacts, the contact force will be generated as a consequence of subsequent movement, although the balls are still apart. The possible ways to resolve this problem are:
 - (a) The contact model should be assigned to contacts with nonzero contact forces only.
 - (b) The contacts with zero contact forces or newly created contacts should be assigned the softening model with the parameter **sof_broken=1**.

The fishcall **FC_BOND_DEL** is called in two circumstances: first, when the tensile strength is first exceeded (**fc_arg(1)=1**); and second, when the bond is broken (**fc_arg(1)=0**). The value of **fc_arg(0)** is the contact pointer.

The force-displacement coding for the two-dimensional form of the displacements softening model is provided in [Example 4.9](#).

Example 4.9 Coding for a displacement softening model – 2D only

```

// fname: soften.TXT
----- built-in, user-defined contact model "softening" -----
void CM4::FDlaw(FdBlock& fb) {
#if DIM == 2 // 2D Section
    double dun;
    double kst = ks * fb.tdel;
    double dus = fb.u_dot_s * kst;
    if (fb.n_force > 0.0) { // COMPRESSIVE
        double knct = knc * fb.tdel;
        dun = fb.u_dot_n * knct;
    }
    else {
        double knctt = knt * fb.tdel;
        dun = fb.u_dot_n * knctt;
    }
    if ((broken != 1.0) | ((broken == 1.0) && (fb.u_n > 0.0))) {
        fb.n_force -= dun;
        fb.s_force -= dus;
    }
    else {
        fb.n_force = 0.;
        fb.s_force = 0.;
    }
    if (fb.n_force > 0.0) { // COMPRESSIVE
        double Dcoef;
        double max_s_force = fb.n_force * (fric * (1. - softened)
            + rfric * softened) + FSmax * (1.0 - softened);
        double max_rs_force =
            fb.s_force < 0.0 ? -fabs(fb.n_force*rfric):fabs(fb.n_force*rfric);
        double sfmag = fabs(fb.s_force);
        if (sfmag > max_s_force) {
            Dcoef = broken == 1 ? 0. : FSmax/uplim;
            double duplas = (sfmag - max_s_force)/(ks - Dcoef);
            fb.s_force -=
                fb.s_force < 0.0 ? -fabs(ks*fabs(duplas)):fabs(ks*fabs(duplas));
            if (fb.s_force*max_rs_force<0|fabs(fb.s_force)<fabs(max_rs_force))
                fb.s_force = max_rs_force;
        }
    }
}

```

```

        uplas += fabs(duplas);
    }
    fb.knest = knc;
    fb.ksest = ks;
    fb.skip = false;
}
else {
    // EXTENSILE
    if (broken == 1.0) {
        fb.n_force = 0.;
        fb.s_force = 0.;
    }
    else {
        double Fn = fb.n_force;
        double Fs = fb.s_force;
        double fmag = sqrt(Fn * Fn + Fs * Fs);
        double alpha = 2.0 * acos(fabs(Fn / fmag)) / dPi;
        double Fmax = (1.0 - alpha) * FTmax + alpha * FSmax;
        double max_force = Fmax * (1.0 - softened);
        if (fmag > max_force) {
            double CoefA = knt*knt*Fn*Fn/(fmag*fmag)+ks*ks*Fs*Fs/(fmag*fmag)
                - Fmax*Fmax/(uplim*uplim);
            double CoefB = 2.*Fmax*Fmax*(1.0-softened)/uplim-2.*knt*Fn*Fn/fmag
                - 2.*ks*Fs*Fs/fmag;
            double CoefC = fmag * fmag - max_force * max_force;
            if (CoefA != 0. && (CoefB*CoefB-4.*CoefA*CoefC)>=0.) {
                double duplas=0.5*(-CoefB-sqrt(CoefB*CoefB-4.*CoefA*CoefC))/CoefA;
                fb.s_force -= fb.s_force<0.0 ? -fabs(ks*fabs(duplas*Fs)/fmag)
                    : fabs(ks*fabs(duplas*Fs)/fmag);

                if (Fs * fb.s_force < 0.) {
                    fb.s_force = 0.;
                }
                fb.n_force -= fb.n_force < 0.0 ? -fabs(knt*fabs(duplas*Fn)/fmag)
                    : fabs(knt*fabs(duplas*Fn)/fmag);

                if (fb.n_force > 0.) {
                    fb.n_force = 0.;
                }
                uplas += fabs(duplas);
            }
            else {
                fb.s_force = 0.;
                fb.n_force = 0.;
                uplas = uplim;
            }
        }
    }
}
fb.knest = knt;

```

```
    fb.ksest = ks;
    fb.skip  = false;
}
if (broken != 1.0) {
    if ((softened == 0.) & (uplas != 0.)) {
        fb.bfishc_brokens = true;
    }
    softened = uplas/uplim < 1.0 ? uplas/uplim : 1.0;
    if (softened == 1.0) {
        broken = 1.0;
        SetDelete(true);
        fb.bfishc_brokenn = true;
        fb.bflag = false;
        fb.broken = true;
    }
}
}
}
#elif DIM == 3                // 3D section ...

#endif
```

4.5 Optional Models

4.5.1 A Hysteretic Damping Model

The contact model **hysdamp** is intended to introduce energy dissipation by hysteretic damping to a linear contact model with frictional slip.

The model has the following property.

hys_knm	normal stiffness, the average of the normal stiffness on loading, Kn_load , and on unloading, Kn_unload
hys_dampn	the ratio of normal stiffness on loading, Kn_load , to that on unloading, fishfontKn_unload, ($0.4 \leq \mathbf{hys_dampn} \leq 1.0$ (with tensile force); $0.05 \leq \mathbf{fishfont}\mathbf{hys_dampn} \leq 1.0$ (without tensile force); default 0.8)
hys_ks	shear stiffness
hys_fric	friction coefficient
hys_nstr	contact bond normal strength [force]
hys_sstr	contact bond shear strength [force]
hys_notension	switch (0: with tensile force (default); 1: without tensile force)
hys_inheritprop	switch (0: the model does not inherit properties from <i>PFC^{2D}</i> , (default); 1: the model inherits the properties)

The normal stiffness on loading, **Kn_load**, and on unloading, **Kn_unload**, used in this model are calculated (Eqs. (4.13) and (4.14)) using **hys_dampn** and **hys_knm**.

$$K_{n_load} = \frac{2 \cdot \mathbf{hys_dampn} \cdot \mathbf{hys_knm}}{(1 + \mathbf{hys_dampn})} \quad (4.13)$$

$$K_{n_unload} = \frac{2 \cdot \mathbf{hys_knm}}{(1 + \mathbf{hys_dampn})} \quad (4.14)$$

where **hys_knm** is taken as the average of **Kn_load** and **Kn_unload**.

Note that if **hys_inheritprop** is set to 1, the model sets the parameters, **hys_knm**, **hys_ks**, **hys_fric**, **hys_nstr** and **hys_sstr** to the values of normal stiffness, shear stiffness, friction

coefficient, contact bond normal strength and contact bond shear strength that associated with contacts before cycling, even though these parameters are specified explicitly. If `hys_inheritprop` is set to 0 (default), users must specify these values (`hys_knm` at minimum). Otherwise, *PFC^{2D}* notices the error message before cycling.

In the hysteretic damping model, normal stiffness on unloading is greater than that on loading (see [Figure 4.3](#)). The hysteretic damping is independent of the relative velocity before and after contact between two entities. It is suggested that the ratio between the two stiffnesses, `hys_dampn`, should be determined with a parametric pretest to obtain a measurable quantity, such as the restitution coefficient (see [Figure 4.4](#)), which illustrates the results of drop tests. Also, the local-damping coefficient should be set to zero in advance by the command `PROP damp` or the *FISH* function `b_damp(bp)` (see the `zero_damp` function in [Example 4.10](#)). The data file used for the drop tests is shown in [Example 4.10](#).

This damping model is applicable to impact problems in which particle movement dominates, but should not be used for problems of compact particle assemblies.

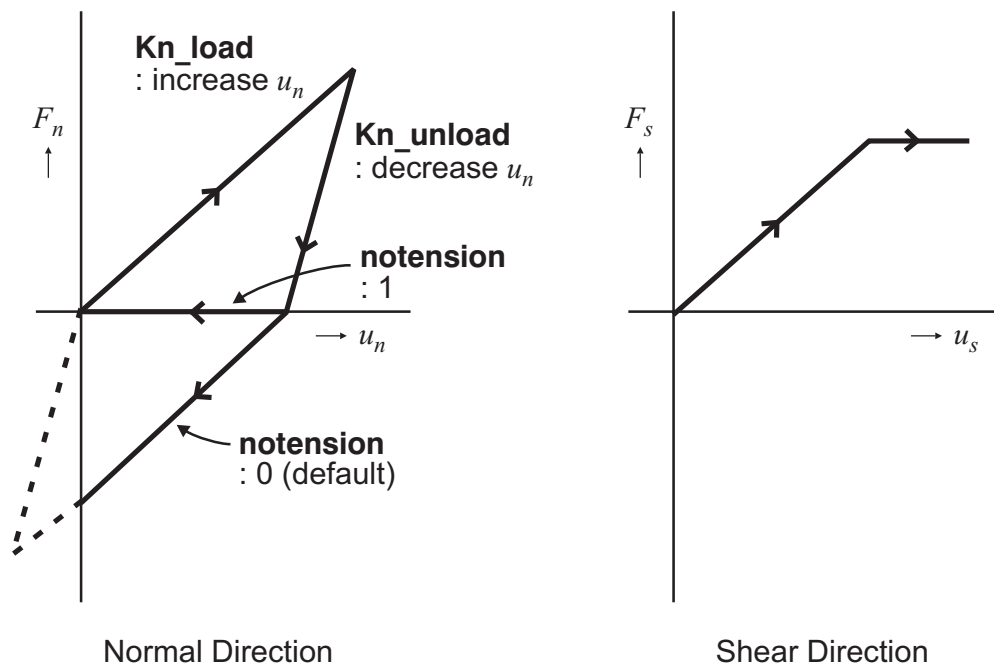


Figure 4.3 *Hysteretic response*

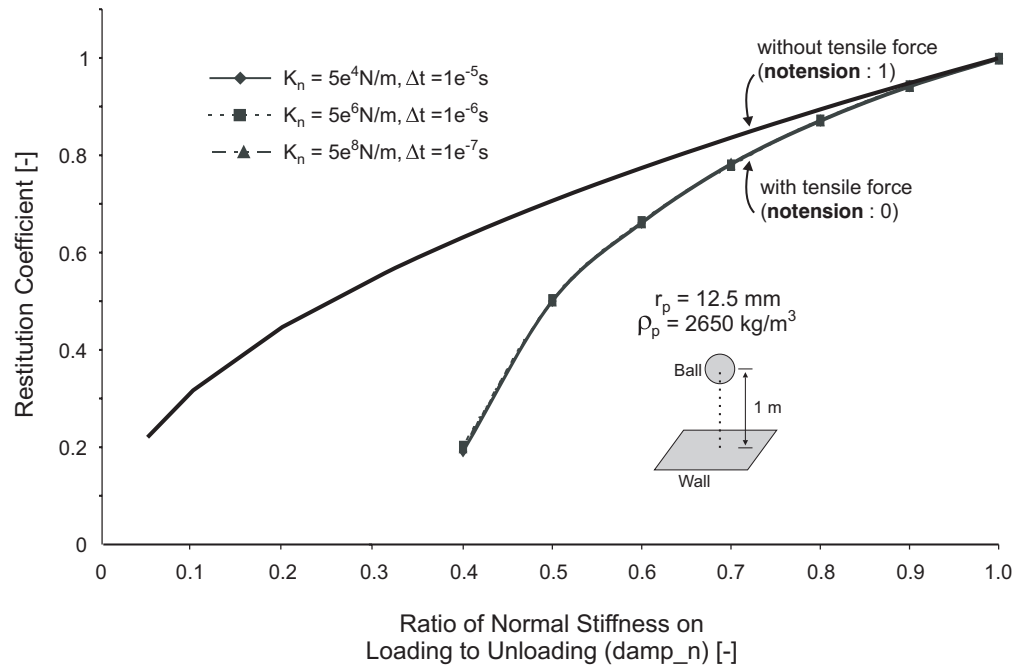


Figure 4.4 Relation between restitution coefficient and ratio of normal stiffness on loading to unloading, `hys_dampn`

Example 4.10 Data file for drop tests

```

; fname: drop2dhys.DAT
new
set dt max 1.0e-4
set pinterval 200
config cppudm
model load hys2wrv.dll
;-----
def make_ball
  command
    ball id = 1 rad = 0.04 x = 0.0 y = 1.0 ;0.03 ; 1.0
    prop dens = 2600 kn = 1.0e6 ks = 1.0e6 fric = 0.0
  end_command
end
;-----
def zero_damp
  bp = ball_head
  loop while bp # null
    b_damp(bp) = 0.0
    bp = b_next(bp)
  end_loop

```

```

end
;-----
def catch_contact_hys
  cp = fc_arg(0);
  c_model(cp) = 'hysdamp'
  c_prop(cp,'hys_dampn')= setv
  c_prop(cp,'hys_notension')= 1 ; 0 if you want tensile force
  c_prop(cp,'hys_inheritprop')= 1
end
;-----
def plot_view
  command
    title 'Drop Tests with Hysteretic Damping (damp_n: 1.0, 0.8, 0.5)'
    plot create 1
    plot add axes black
    plot add ball yellow
    plot add wall lblue
    plot add cf white
  end_command
end

model hysdamp
set fishcall 6 catch_contact_hys
plot_view

wall id = 1 kn = 1.0e6 ks = 1.0e6 fric = 0.0 &
  nodes (-0.5,0.0) (0.5,0.0)
set grav 0 -9.8

; ----- case damp_n = 1.0 -----
make_ball
zero_damp
set setv = 1.0
plot show
cycle 30000
del ball 1
; ----- case damp_n = 0.8 -----
make_ball
zero_damp
set setv = 0.8
cycle 70000
del ball 1
; ----- case damp_n = 0.5 -----
make_ball
zero_damp
set setv = 0.5

```

```

cycle 20000
return

```

4.5.2 Burger's Model

The Burger's model shown by Figure 4.5 is implemented to simulate creep mechanisms in *PFC^{2D}*. The model contains the Kelvin model and the Maxwell model, which are connected in series in both the normal and shear directions, respectively, at a contact point.

The model has the following properties.

- bur_knk** normal stiffness for Kelvin section (K_{k_n})
- bur_cnk** normal viscosity for Kelvin section (C_{k_n})
- bur_knm** normal stiffness for Maxwell section (K_{m_n})
- bur_cnm** normal viscosity for Maxwell section (C_{m_n})
- bur_ksk** shear stiffness for Kelvin section (K_{k_s})
- bur_csk** shear viscosity for Kelvin section (C_{k_s})
- bur_ksm** shear stiffness for Maxwell section (K_{m_s})
- bur_csm** shear viscosity for Maxwell section (C_{m_s})
- bur_fric** friction coefficient (f_s)
- bur_notension** switch (O: with tensile force (default); 1: without tensile force)

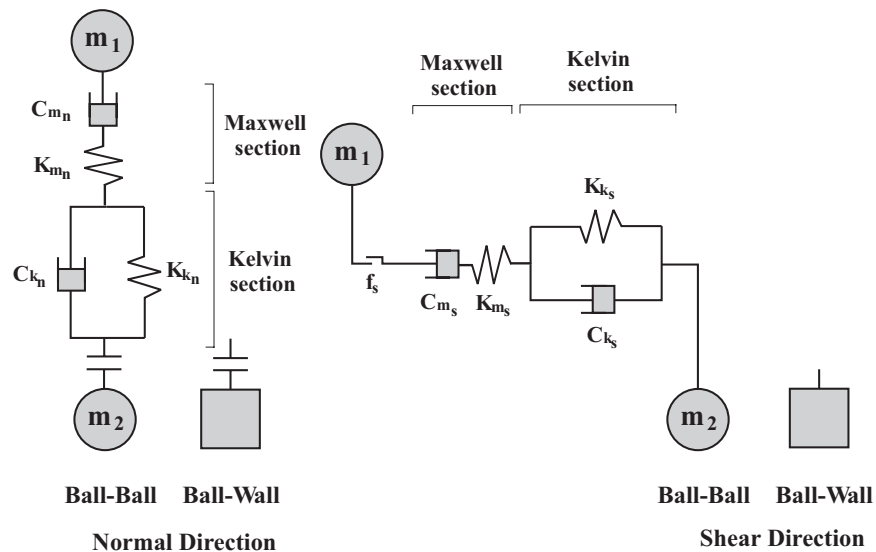


Figure 4.5 The Burger's model in PFC

4.5.2.1 Numerical Scheme

The total displacement of the Burger's model, u , is the sum of the displacement of the Kelvin section (u_k) and that of the Maxwell section (u_{m_K}, u_{m_C}) of the model, given by Eq. (4.15). Note that symbols \pm and \mp correspond to the cases of normal direction and shear direction. (For example, \pm means + for normal direction and - for shear direction.)

$$u = u_k + u_{m_K} + u_{m_C} \quad (4.15)$$

The first and second derivatives of Eq. (4.15) are given by Eqs. (4.16) and (4.17):

$$\dot{u} = \dot{u}_k + \dot{u}_{m_K} + \dot{u}_{m_C} \quad (4.16)$$

$$\ddot{u} = \ddot{u}_k + \ddot{u}_{m_K} + \ddot{u}_{m_C} \quad (4.17)$$

The contact forces, f , using the Kelvin section and the first derivative, are given by Eqs. (4.18) and (4.19):

$$f = \pm K_k u_k \pm C_k \dot{u}_k \quad (4.18)$$

$$\dot{f} = \pm K_k \dot{u}_k \pm C_k \ddot{u}_k \quad (4.19)$$

Also, using stiffness K_m and viscosity C_m of the Maxwell section,

$$f = \pm K_m u_{m_K} \quad (4.20)$$

$$\dot{f} = \pm K_m \dot{u}_{m_K} \quad (4.21)$$

$$\ddot{f} = \pm K_m \ddot{u}_{m_K} \quad (4.22)$$

$$f = \pm C_m \dot{u}_{m_C} \quad (4.23)$$

$$\dot{f} = \pm C_m \ddot{u}_{m_C} \quad (4.24)$$

Using Eqs. (4.16) through (4.24), the second-order differential equation for contact force f is given by

$$f + \left[\frac{C_k}{K_k} + C_m \left(\frac{1}{K_k} + \frac{1}{K_m} \right) \right] \dot{f} + \frac{C_k C_m}{K_k K_m} \ddot{f} = \pm C_m \dot{u} \pm \frac{C_k C_m}{K_k} \ddot{u} \quad (4.25)$$

From Eq. (4.18) of the Kelvin section,

$$\dot{u}_k = \frac{-K_k u_k \pm f}{C_k} \quad (4.26)$$

By using a central difference approximation of the finite difference scheme for the time derivative and taking average values for u_k and f ,

$$\frac{u_k^{t+1} - u_k^t}{\Delta t} = \frac{1}{C_k} \left[-\frac{K_k (u_k^{t+1} + u_k^t)}{2} \pm \frac{f^{t+1} + f^t}{2} \right] \quad (4.27)$$

therefore,

$$u_k^{t+1} = \frac{1}{A} \left[B u_k^t \pm \frac{\Delta t}{2C_k} (f^{t+1} + f^t) \right] \quad (4.28)$$

where

$$A = 1 + \frac{K_k \Delta t}{2C_k} \quad (4.29)$$

$$B = 1 - \frac{K_k \Delta t}{2C_k} \quad (4.30)$$

For the Maxwell section, the displacement and the first derivative are given by Eqs. (4.31) and (4.32):

$$u_m = u_{mK} + u_{mC} \quad (4.31)$$

$$\dot{u}_m = \dot{u}_{mK} + \dot{u}_{mC} \quad (4.32)$$

Substituting Eqs. (4.21) and (4.23) into Eq. (4.32),

$$\dot{u}_m = \pm \frac{\dot{f}}{K_m} \pm \frac{f}{C_m} \quad (4.33)$$

By using a central difference approximation of the finite difference scheme and taking the average value for f ,

$$\frac{u_m^{t+1} - u_m^t}{\Delta t} = \pm \frac{f^{t+1} - f^t}{K_m \Delta t} \pm \frac{f^{t+1} + f^t}{2C_m} \quad (4.34)$$

therefore,

$$u_m^{t+1} = \pm \frac{f^{t+1} - f^t}{K_m} \pm \frac{\Delta t (f^{t+1} + f^t)}{2C_m} + u_m^t \quad (4.35)$$

The total displacement and the first derivative of the Burger's model are given by Eqs. (4.36) and (4.37):

$$u = u_k + u_m \quad (4.36)$$

$$\dot{u} = \dot{u}_k + \dot{u}_m \quad (4.37)$$

By using the finite difference scheme for the time derivative,

$$u^{t+1} - u^t = u_k^{t+1} - u_k^t + u_m^{t+1} - u_m^t \quad (4.38)$$

Substituting Eqs. (4.28) and (4.35) into Eq. (4.38), the contact force, f^{t+1} , is given by Eq. (4.39):

$$f^{t+1} = \pm \frac{1}{C} \left[u^{t+1} - u^t + \left(1 - \frac{B}{A} \right) u_k^t \mp Df^t \right] \quad (4.39)$$

where

$$C = \frac{\Delta t}{2C_k A} + \frac{1}{K_m} + \frac{\Delta t}{2C_m} \quad (4.40)$$

$$D = \frac{\Delta t}{2C_k A} - \frac{1}{K_m} + \frac{\Delta t}{2C_m} \quad (4.41)$$

Contact force f^{t+1} can be calculated from known values for u^{t+1} , u^t , u_k^t and f^t .

4.5.2.2 Simple Examples

Figure 4.6 shows the time history of the contact force in the normal direction for step input $u(t) = 0.01$. The FISH program is listed in Example 4.11, in which two balls are created under an overlap condition, then fixed in position and spin. The result shows that the contact force exponentially decreases as time increases, which coincides with the analytical solution, Eq. (4.42), as shown by Figure 4.7.

$$f(t) = A_1 \exp(z_1 t) + A_2 \exp(z_2 t) \quad (4.42)$$

where

$$A_1 = \frac{b_2 z_1 + b_1}{a_2(z_1 - z_2)} \quad (4.43)$$

$$A_2 = \frac{b_2 z_2 + b_1}{a_2(z_2 - z_1)} \quad (4.44)$$

$$a_1 = \frac{C_k}{K_k} + C_m \left(\frac{1}{K_k} + \frac{1}{K_m} \right) \quad (4.45)$$

$$a_2 = \frac{C_k C_m}{K_k K_m} \quad (4.46)$$

$$b_1 = \pm C_m \quad (4.47)$$

$$b_2 = \pm \frac{C_k C_m}{K_k} \quad (4.48)$$

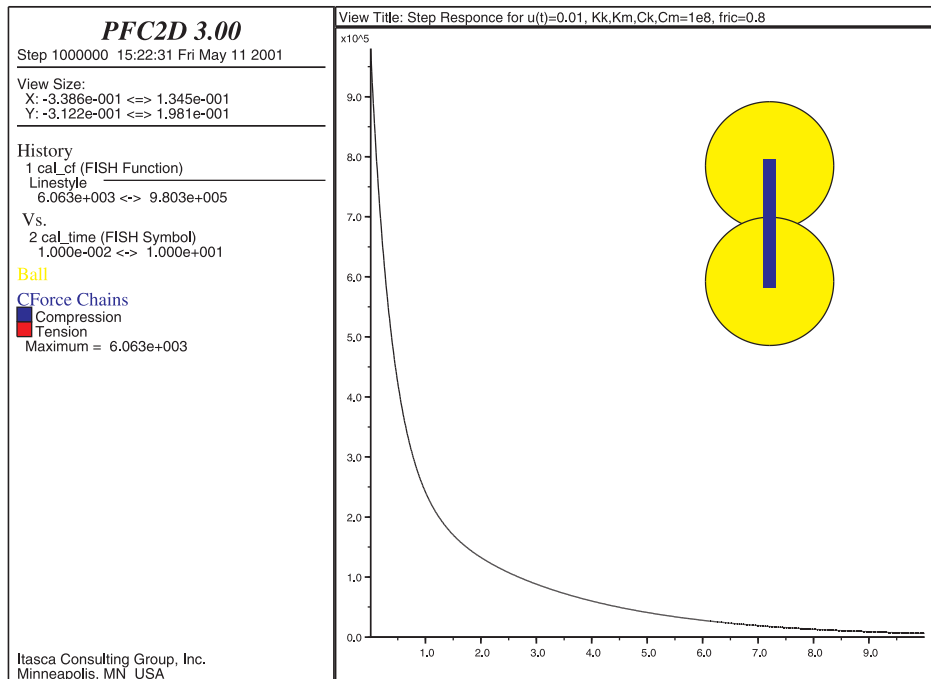


Figure 4.6 Time history of normal contact force with two balls, PFC^{2D}

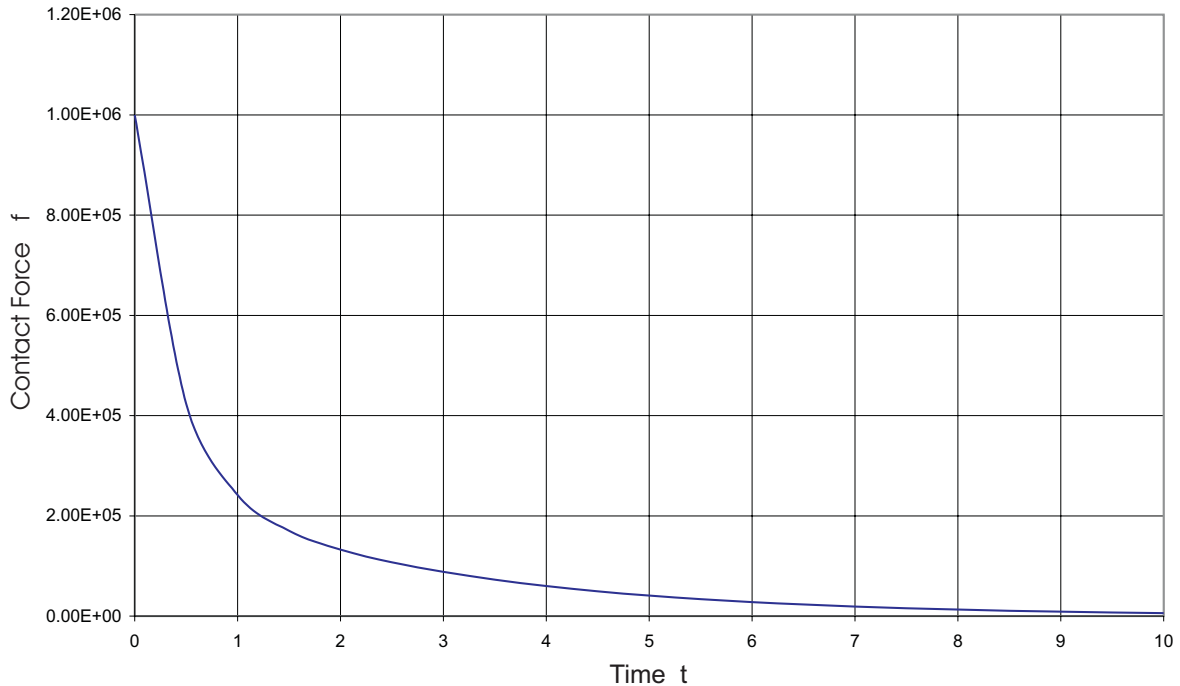


Figure 4.7 Analytical solution for step input $u(t) = 0.01$, ($K_k = K_m = C_k = C_m = 1.0 \times 10^8$)

Example 4.11 Data file for simple stress relaxation test

```

; fname: burger2d.DAT
new
set dt max 1e-5
;----- FISH function -----
def set_param
  k_set = 1.0e8
  c_set = 1.0e8
  f_set = 0.0
end
set_param
;-----
def catch_contact_bur
  cp = fc_arg(0);
  c_model(cp) = 'burger'
  c_prop(cp, 'bur_knk') = k_set
  c_prop(cp, 'bur_cnk') = c_set
  c_prop(cp, 'bur_knm') = k_set
  c_prop(cp, 'bur_cnm') = c_set
  c_prop(cp, 'bur_ksk') = k_set
  c_prop(cp, 'bur_csk') = c_set
  c_prop(cp, 'bur_ksm') = k_set
  c_prop(cp, 'bur_csm') = c_set
  c_prop(cp, 'bur_fric') = f_set
end
;-----
def cal_cf
  cal_cf = c_nforce(contact_head)
  cal_time = time - time0
end
;-----
def reset_time
  time0 = time
end
; ----- main -----
; config & load DLL for the Burger's model
config cppudm
model load bur2wrv ; 2d VC++ Release version
; model load bur2wdv ; 2d VC++ Debug version
; set the Burger's model
model burger
set fishcall 6 catch_contact_bur
set fishcall 0 cal_cf
; make BALL
ball id = 1 rad = 0.05 x = 0.0 y = 0.0
ball id = 2 rad = 0.05 x = 0.0 y = 0.09

```

```
prop dens = 2600
fix x y spin
; set property
prop bur_knk = k_set bur_cnk = c_set
prop bur_knm = k_set bur_cnm = c_set
prop bur_ksk = k_set bur_csk = c_set
prop bur_ksm = k_set bur_csm = c_set
prop bur_fric = f_set
; history data
his id=1 cal_cf
his id=2 cal_time
hist nstep = 1000
; cycle until 10 s
reset_time
cycle 1000000
```

4.6 References

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

Revised 15 August 2001

