EDEM 2.4 Programming Guide

Revision 1

Copyrights and Trademarks

Copyright © 2011 DEM Solutions. All rights reserved.

Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of those agreements. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission.

DEM Solutions
49 Queen Street
Edinburgh
EH2 3NH
UK

www.dem-solutions.com

EDEM® incorporates CADfix translation technology. CADfix is owned, supplied by and Copyright © TranscenData Europe Limited, 2007. All Rights Reserved. This software is based in part on the work of the Independent JPEG Group. EDEM uses the Mersenne Twister random number generator, Copyright © 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved. EDEM includes CGNS (CFD General Notation System) software. See the Online Help for full copyright notice.

EDEM® and Particle Factory® are registered trademarks of DEM Solutions. All other brands or product names are the property of the respective owners.

Contents

Overview ... 4
  EDEM User Defined Library Development Process ... 4
  EDEM Simulation Sequence ... 5
  EDEM API Files ... 6
Contact Model User Defined Libraries ... 8
  Contact Model API v2.2.0 ... 9
  Developing a Contact Model User Defined Library ... 10
    Using the New Contact Model ... 14
    Using the EDEM Analyst to Verify the Contact Model ... 15
  Particle Body Force User Defined Libraries ... 18
  Particle Body Force API v2.1.0 ... 19
  Developing a Particle Body Force User Defined Library ... 20
    Using the New Particle Body Force ... 23
Particle Factory User Defined Libraries ... 24
  Particle Factory API v2.0.0 ... 25
  Developing a Particle Factory User Defined Library ... 26
    Using the New Particle Factory ... 31
  Custom Properties ... 32
  Using Custom Properties ... 33
    Developing a UDL that uses Custom Particle Properties ... 33
    Create and Compile the Custom Particle Property ... 33
    Using the New Custom Particle Property ... 36
Glossary ... 37
Appendix A: Code Used in Examples ... 39
    CCohesionModel.h ... 39
    CCohesionModel.cpp ... 42
    CForceExample.h ... 45
    CForceExample.cpp ... 47
Index ... 49

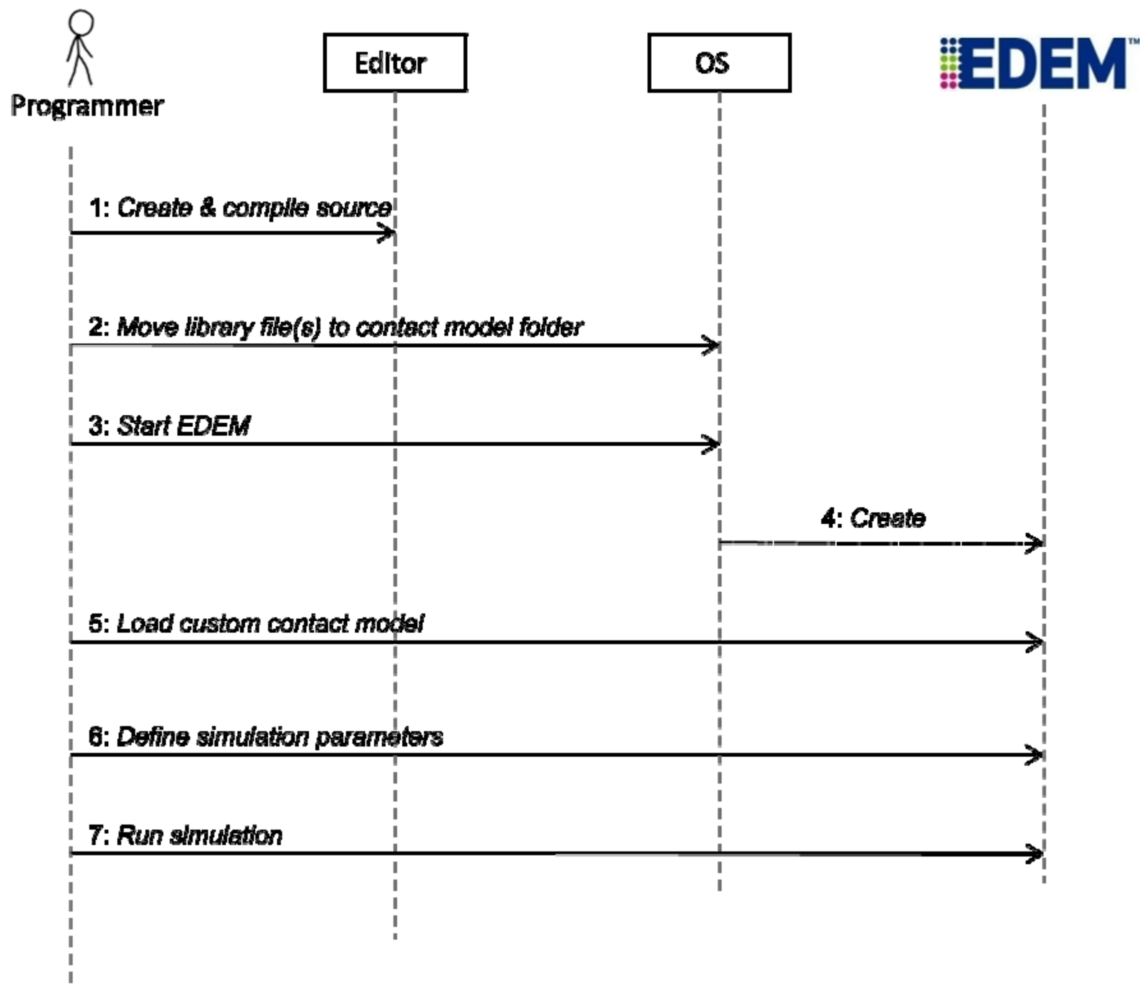
Overview

This document describes the EDEM® API, which you can use to code and implement your own contact physics, external couplings and particle-generation factories in the form of a *User Defined Library* (UDL) which operates as a plugin to EDEM®. Read this document along with the API reference help at `\src\Api\Help\index.html`

EDEM User Defined Library Development Process

The figure below depicts a high-level sequence diagram showing the steps to write and use a UDL contact model. The same steps apply for Particle Body Force and Particle Factory UDLs.

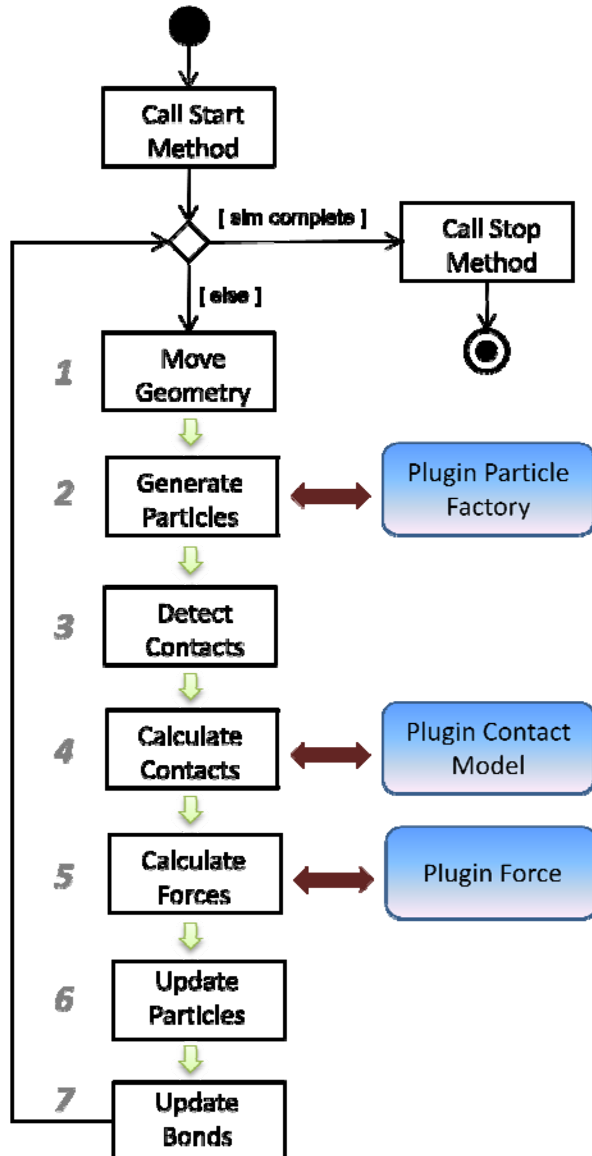
The programmer typically adapts an existing contact model source file (available from the DEM Solutions website) using a development environment or editor. Once compiled, the new contact model is loaded into EDEM for use in simulations.



EDEM Simulation Sequence

The figure below shows when UDL libraries are called during EDEM's simulation sequence. Depending on where the UDL lies in the chain, each stage connects to the relevant UDL (if applicable) before performing its main function (such as calculating contacts).

1. Move geometry.
2. Generate new particles by calling each registered particle factory in turn (some of which may be UDLs).
3. Detect contacts.
4. Invoke the contact model chain for each contact. Each contact model in the chain is executed in order, irrespective of whether it is a UDL or built-in model. Note there are two different contact model chains: one for particle-to-particle contacts and one for particle-to-geometry contacts. Your UDL can be loaded into one or both of these as required.
5. Apply the Particle Body Force chain to each particle in the system. Particle Body Force models in the chain are executed in order, irrespective of whether it is a UDL or built-in model.
6. Update particles based on the forces applied to them in this timestep.
7. Update bond information (if applicable).
8. If there are more timesteps, repeat from step 1 otherwise finish.



EDEM API Files

EDEM 2.4 includes a new v2.2.0 C++ API. Use this to get the normalOverlap calculated from the physical radius of particles. This version supports multi-threading. You can also continue to use any legacy API from previous versions of EDEM. Legacy versions of the IPluginContactModel will continue to receive the normalOverlap calculated from the contact radius of particle surfaces. For full details on the legacy API, refer to the *EDEM 2.1.2 Programming Guide*, available from the customer area of our website.

The tables below gives an overview of EDEM's API source file location and files. For full details, refer to the API reference help at `\src\Api\Help\index.html`

Example source files (which you can use as the basis for writing your own UDLs) are available from the customer area of our website.

API Directories

Directory	Description
`src/Api`	v2.0.0 (and later) API files for contact models, particle body forces, and factories.
`src/Api/Help`	Online reference help.
`src/Misc`	Additional files that may be of use when developing UDLs. The Helpers.h file is an updated version of sharedTypes.h from the legacy API.
`src/LegacyApi`	Older API files as shipped with the previous version of EDEM.

Core API Header Files

Header File	Description
ApiIds.h	Contains unique ID numbers for all supported APIs.
ApiTypes.h	Contains constants and declarations used by the EDEM generic API.
IApi.h	Defines the IApi interface. This is the base interface for all non-UDL APIs.
IApiManager_1_0.h	Defines the IApiManager_1_0 interface that provides the ability to allocate and initialize various APIs for use by UDLs.
ICustomPropertyDataApi_1_0.h	Provides basic access to custom property data and delta values.
ICustomPropertyManagerApi_1_0.h	Provides basic access to custom property meta-data for one of four different custom property collections (particle, geometry element, contact, simulation).
IFieldApi_1_0.h	Defines the IFieldAPI_1_0 interface that provides basic access to field data via a series of query methods.
PluginConstants.h	Contains constants and declarations used by Contact Model, Particle Body Force, and Factory UDLs.

Contact Model Header Files

Header File	Description
IPluginContactModel.h	Defines the IPluginContactModel interface that all Contact Model UDLs versioned interfaces derive from.
IPluginContactModelV2_2_0.h	Defines the IPluginContactModelV2_2_0 versioned interface for contact models. Contains all the main methods to implement.
PluginContactModelCore.h	Core interface for contact models. Implement the methods contained here to enable EDEM to access your UDL.

Particle Body Force Header Files

Header File	Description
IPluginParticleBodyForce.h	Defines the IPluginParticleBodyForce interface that all Particle Body Force versioned interfaces derive from.
IPluginParticleBodyForceV2_1_0.h	Defines the IPluginParticleBodyFroceV2_1_0 versioned interface for particle body forces. Contains all the main methods to implement.
PluginParticleBodyForceCore.h	Core interface for Particle Body Force UDLs. Implement the methods contained here to enable EDEM to access your UDL.

Factory Header Files

Header File	Description
IPluginParticleFactory.h	Defines the PluginParticleFactory UDL interface that all Factory versioned interfaces derive from.
IPluginParticleFactoryV2_0_0.h	Defines the IPluginParticleFactoryV2_0_0 versioned interface for factories. Contains all the main methods to implement.
PluginParticleFactoryCore.h	Core interface for factories. Implement the methods contained here to enable EDEM to access your UDL.

Other Header Files

Header File	Description
CGenericFileReader.h	Provides a generic way to read configuration files. You will also need to compile the CGenericFileReader.cpp file into your UDLs shared library along with other source files.
Helpers.h	Includes various utility classes to help with vector and matrix calculations.

Contact Model User Defined Libraries

Using the EDEM API, you can write and compile custom contact models as a UDL written in C/C++. A Contact Model UDL consists of a shared library file and optionally a file containing preferences.

Overview of Creating a New Contact Model

1. Create a directory to contain the new contact model project.
2. Copy the following header files into the directory:
   - IPluginContactModel.h
   - PluginContactModelCore.h
   - IPluginContactModelV2_2_0.h
   - PluginConstants.h
3. If required, also copy the following optional header files:
   - CGenericFileReader.h
   - Helpers.h
4. Create a new class (CNewContactModel) derived from the contact model interface you want to use (e.g. IPluginContactModelV2_2_0).
5. Save the class declaration to a header file (e.g. CNewContactModel.h). This should consist of declarations of the methods in the chosen interface, any extra methods required, and any variables.
6. Implement all of the methods defined in your header file and save as a cpp file (e.g. CNewContactModel.cpp).
7. Create a new .cpp file with implementations of the methods in PluginContactModelCore.h. For example, save as NewContactModel.cpp
8. Compile all of your src files and link them together in to .dll (Windows) or .so (Linux) library files.
9. Be sure the library and optional preferences file are in the contact model folder (as specified with Tools > Options > File Locations).
10. Start EDEM then select the required contact model category from the Interaction pulldown in the Physics section.
11. Click the **+** drop-down list then select your new contact model.

For a detailed example, refer to *Developing a Contact Model User Defined Library* on page 10.

Contact Model API v2.2.0

The contact model interface IPluginContactModelV2_2_0 provides these methods:

Method	Description	Type
getDetailsForProperty	Retrieves details for a given property. These properties will then be registered with the system if they do not clash with any existing properties.	Setup
getNumberOfRequiredProperties	Returns the number of custom properties this UDL wants to register with the system.	Setup
getPreferenceFileName	Retrieves the name of the config file used by the UDL.	Setup
isThreadSafe	If the UDL's calculateForce() method is thread-safe then this method should return true. Thread safe programming requires a number of conventions and restrictions to be followed. If in doubt set this to return false.	Setup
setup	Initializes the plugin by reading any config files, opening temporary files, generating data structures or any other setup work.	Setup
usesCustomProperties	Indicates whether the UDL wants to register or receive custom property data.	Setup
starting	Called to indicate processing is about to begin and the model should allocate any temporary storage and retrieve any required file/API/socket handles.	Simulation
calculateForce	Called when two elements are in contact with each other. These may be surfaces (in-contact when their contact radii cross), or a surface and a geometry element (when the contact radius of the surface touches the surface geometry). Also used to set contact forces and deltas for custom properties.	Simulation
stopping	Called to indicate processing is finished and that the model should free any temporary storage and close/release file/api/socket handles.	Simulation

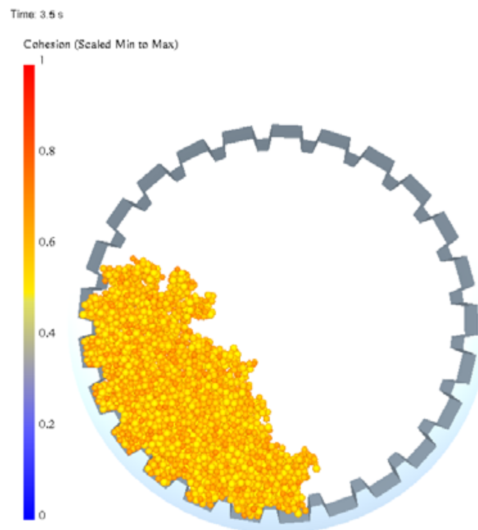
Refer to the online API documentation for full details.

Developing a Contact Model User Defined Library

This section describes how to create a new cohesion model UDL.

The following example is slightly different from the cohesion model provided in the programming examples section of the DEM Solutions website. This simpler example does not store the energy density values for different particle types combination. The cohesion model includes time-dependent particle-to-particle and particle-to-geometry cohesion.

For MS Windows, this example uses Microsoft Visual C++ 2008 Express Edition to compile the contact models. See http://www.microsoft.com/express/download/. For Linux, this example uses the standard GNU Compiler Collection (GCC).



Step 1: Prepare the Header Files and Derive Classes

1. Create a new folder then copy the following header files from `src\Api\ContactModels` and `src\Api\Core` into your project folder.
    - IPluginContactModel.h
    - PluginContactModelCore.h
    - IPluginContactModelV2_2_0.h
    - PluginConstants.h
    - Helpers.h
2. Create a file **cohesion_prefs.txt** in your working directory. Edit the file to contain the lines:

    ```
    particle:particle 4e5
    particle:mill 6e5
    ```

    This indicates a particle-particle cohesion with an energy density of 4e5 J/m^3 and a particle-mill cohesion of 6e5 J/m^3.

Step 2: Implement core UDL methods

1. Derive a class from IPluginContactModelV2_2_0 (found in the header file IPluginContactModelV2_2_0.h) and save it as CCohesionModel.h. Refer to the CCohesionModel.h code extract below (the complete file is shown in *Appendix A: Code Used in Examples* on page 39).

```
#if !defined(ccohesionmodel_h)
#define ccohesionmodel_h
#include <string>
#include <fstream>
#include "IPluginContactModelV2_2_0.h"

using namespace std;

class CCohesionModel : public NApiCm::IPluginContactModelV2_2_0
{
    public:

        /**
         * Constructor, does nothing
         */
        CCohesionModel();

        /**
         * Destructor, does nothing
         */
        virtual ~CCohesionModel();

        /**
         * Sets prefFileName to the empty string as this plugin has no
         * configuration
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual void getPreferenceFileName(char
prefFileName[NApi::FILE_PATH_MAX_LENGTH]);


        /*******
         * Insert the rest of the code here
         *******/

        /**
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual bool getDetailsForProperty(
                            unsigned int                       propertyIndex,
                            NApi::EPluginPropertyCategory      category,
                            char
name[NApi::CUSTOM_PROP_MAX_NAME_LENGTH],
                            NApi::EPluginPropertyDataTypes&    dataType,
                            unsigned int&                      numberOfElements,
                            NApi::EPluginPropertyUnitTypes&    unitType);

              /**
         * Name of the preferences file to load bond information from
         */
        static const std::string PREFS_FILE;
```

```
            private:
                  double m_energyDensity;
};

#endif
```

2. Implement all required methods in PluginContactModelCore.h. Save as CohesionModel.cpp as shown below:

```
#include "PluginContactModelCore.h"
#include "CCohesionModel.h"

using namespace NApiCm;

EXPORT_MACRO IPluginContactModel* GETCMINSTANCE()
{
    return new CCohesionModel();
}

EXPORT_MACRO void RELEASECMINSTANCE(IPluginContactModel* plugin)
{
    if (0 != plugin)
    {
        delete plugin;
    }
}

EXPORT_MACRO int GETCMINTERFACEVERSION()
{
    static const int INTERFACE_VERSION_MAJOR = 0x02;
    static const int INTERFACE_VERSION_MINOR = 0x02;
    static const int INTERFACE_VERSION_PATCH = 0x00;

    return (INTERFACE_VERSION_MAJOR << 16 |
            INTERFACE_VERSION_MINOR << 8 |
            INTERFACE_VERSION_PATCH);
}
```

3. Implement all methods in CCohesionModel.h and save it as CCohesionModel.cpp. Refer to the code extract below (the complete file is shown in *Appendix A: Code Used in Examples* on page 39) which shows the implementation of the physics part in the `ECalculateResult CCohesionModel::calculateForce` method.

```
    // This contact model assumes that the Physical and the Contact radius
    // are the same

    // The unit vector from element 1 to the contact point
    CSimple3DPoint  contactPoint = CSimple3DPoint(contactPointX, contactPointY,
contactPointZ);
    CSimple3DVector unitCPVect   = contactPoint - CSimple3DPoint(elem1PosX,
elem1PosY, elem1PosZ);
    unitCPVect.normalise();

    // Cohesion
    // R^2 - (R - d)^2

    double nCohesiveFactor         = m_energyDensity;
    double nRadiusOfOverlapSquared = 2 * elem1PhysicalCurvature * normalOverlap;
```

```
double nArea                     = PI * nRadiusOfOverlapSquared;
CSimple3DVector F_cohesive       = unitCPVect * nCohesiveFactor * nArea;
```

The calculation of the cohesion force as a function of time is shown below:

```
calculatedNormalForceX = F_cohesive.dx();
calculatedNormalForceY = F_cohesive.dy();
calculatedNormalForceZ = F_cohesive.dz();

return eSuccess;
```

Step 3: Create the Library File

If you are compiling on MS Windows:

1. Start the compiler's command prompt:

   Start > All Programs > Visual C++ 9.0 Express Edition > Visual Studio Tools > Visual Studio 2008 Command Prompt

2. Browse to your working directory using the cd command.
3. Compile using the following command:

   ```
   cl /O2 /DWIN32 /LD CohesionModel.cpp CCohesionModel.cpp
   ```

   The file C*ohesionModel.dll* is created in your working directory.

If you are compiling on Linux:

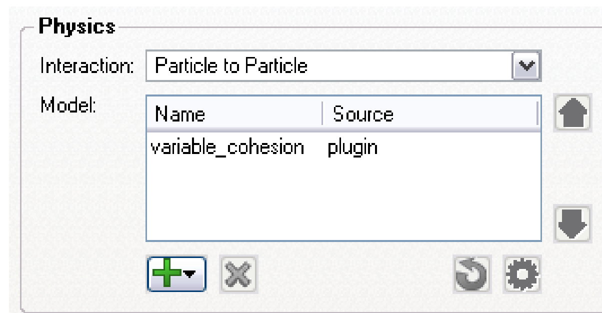
1. Open a terminal window and browse to your working directory.
2. Compile using the following command:

   ```
   gcc -O2 -shared -fPIC CohesionModel.cpp CCohesionModel.cpp -o
   CohesionModel.so
   ```

   The file *CohesionModel.so* is created in your working directory.

Using the New Contact Model

1. Start EDEM then open a simulation that will use the new contact model. You can download an example simulation (`cohesion_input.dem`) from the customer area of the DEM Solutions website.
2. Load your new contact model. Select Tools > Options > File Locations > Contact Models then navigate to the location of your newly created dll.
3. Select *Particle to Particle* from the Interaction pulldown in the Physics section.
4. Click the **+** button then select the new cohesion contact model.
5. If your contact model includes code similar to Hertz-Mindlin (in other words, code to calculate base contact forces), remove the built-in Hertz-Mindlin (no slip) contact model from the list.



6. Select *Particle to Geometry* from the Interaction pulldown in the Physics section.
7. Click the **+** button then select the cohesion contact model.
8. If your contact model includes code similar to Hertz-Mindlin (in other words, code to calculate base contact forces), remove the built-in Hertz-Mindlin (no slip) contact model from the list.
9. Click the Geometry tab.
10. Be sure the periodic boundary is enabled in the Y axis.
11. Select File > Save.

Step 2: Run the Simulation

The particles have already been created in this simulation and the model has been simulated until a steady state was achieved.

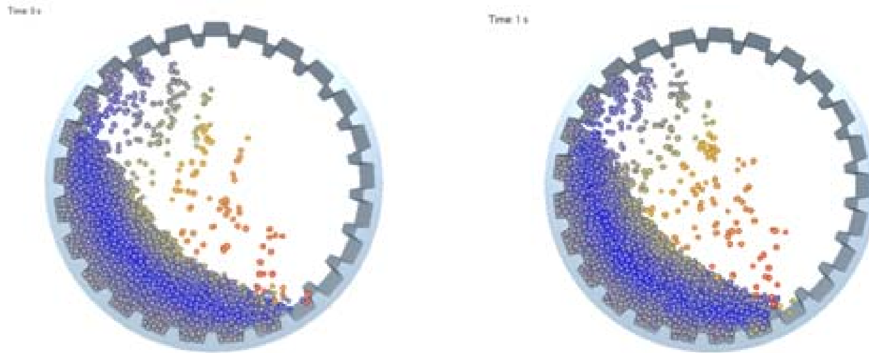
1. Click on the Simulator button.
2. Click the Start progress button to start processing the simulation.

Using the EDEM Analyst to Verify the Contact Model

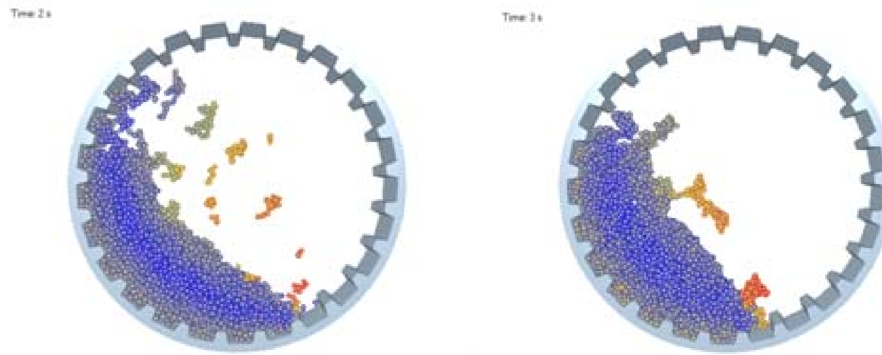
Step 1: Visualizing the Results

Run through the simulation to verify the changing behavior of the particles over time.

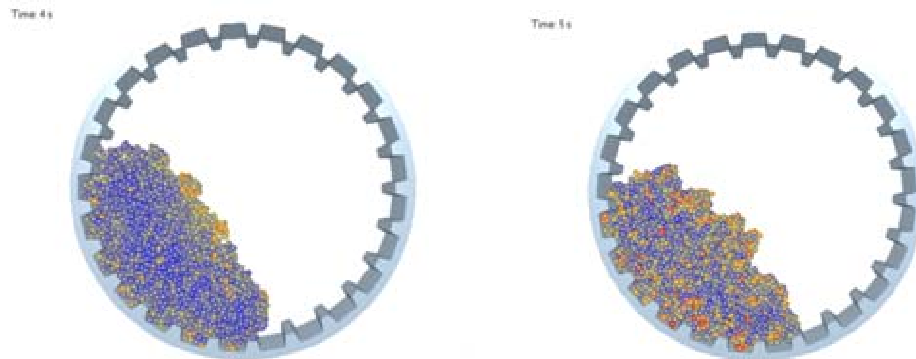
0s to 1s: Simulation behavior has not changed significantly from steady state:



2s to 3s: Particle clumping occurs as the particles are thrown from the mill lifters:



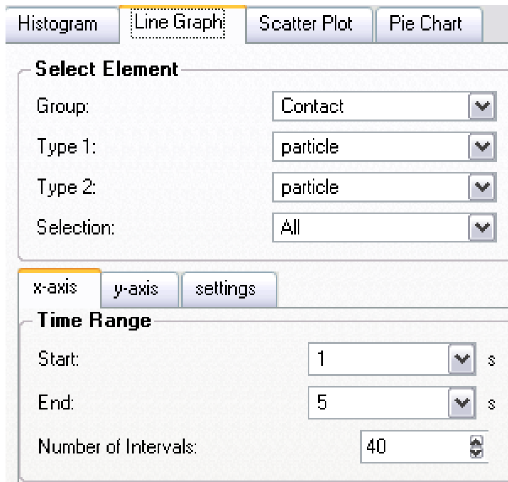
4s to 5s: The particle cohesion is high enough to prevent the particles from separating in the mill; the particles act as a single mass:



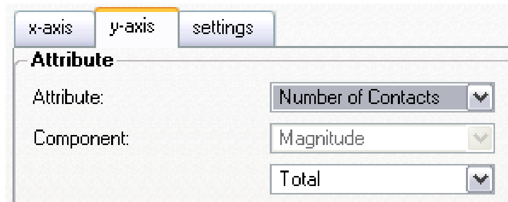
Step 2: Plotting Contact Graphs

Plot some contact graphs to see how the contact behavior changes with the increasing cohesion.

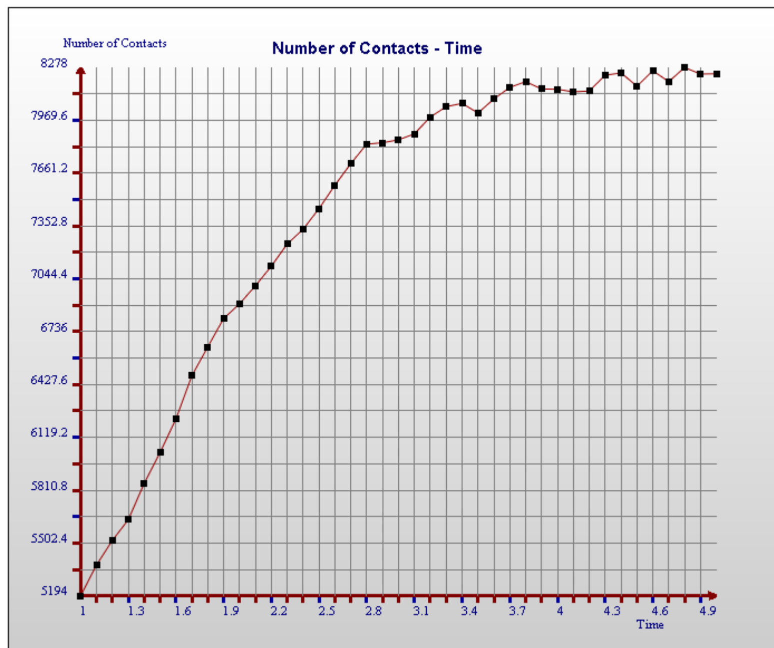
1. Click the Analyst button on the toolbar then click the Create Graph button.
2. Select the Line Graph tab then set the element and x-axis details as follows:



3. Click the Y-Axis tab then select *Number of Contacts* as the Attribute to plot:

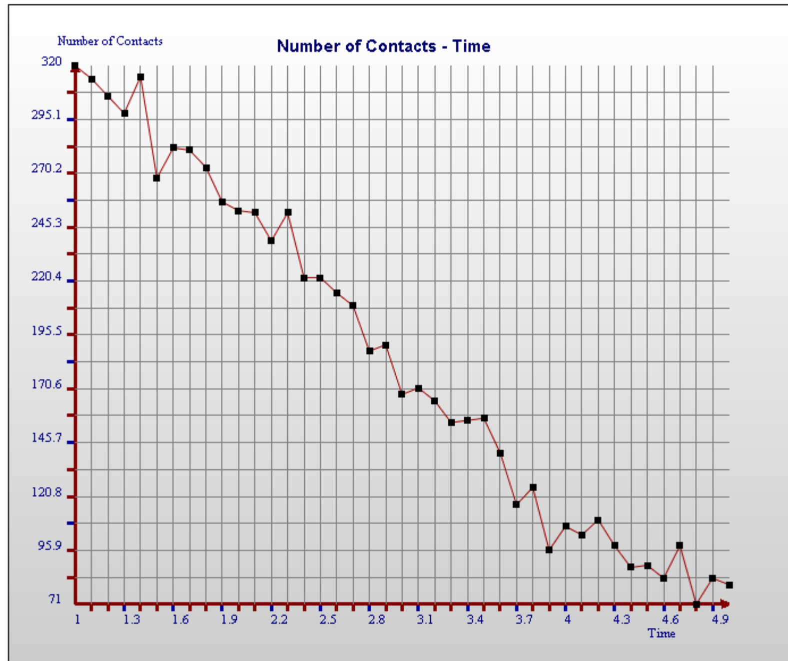


4. Click the Create Graph button to plot the graph.



The particle-to-particle contact graph shows the number of contacts increasing with time until a saturation point is reached.

5. Change the Type 2 option from particle to mill then create a new graph:



The graph shows that the number of particles in contact with the geometry decreases over time; the particle flow has change from individual particles that impact on the geometry to a single mass that rolls in the mill. This single mass has reduced impact on the mill geometry when compared to the individual particles.

Particle Body Force User Defined Libraries

Using the EDEM API, you can write and compile custom particle body forces as a UDL written in C/C++. A Particle Body Force UDL consists of a shared library file and optionally a file containing preferences.

Overview of Creating a New Particle Body Force UDL

1. Create a directory to contain the new Particle Body Force project.
2. Copy the following header files into the directory:
    - IPluginParticleBodyForce.h
    - PluginParticleBodyForceCore.h
    - IPluginParticleBodyForceV2_1_0.h
    - PluginConstants.h
    - Helpers.h
3. Create a new class (CNewParticleBodyForce) derived from the particle body force interface you want to use (e.g. IPluginParticleBodyForceV2_1_0).
4. Save the class declaration (consisting of declarations of the methods in the chosen interface and any extra methods needed for your functionality and any variables) to a header file (e.g. CNewParticleBodyForce.h)
5. Implement all of the methods defined in your header file and save as a cpp file (e.g. CNewParticleBodyForce.cpp).
6. Create a new .cpp file with implementations of the methods in PluginParticleBodyForceCore.h. Save as e.g. NewParticleBodyForce.cpp
7. Use command line or create a makefile (or equivalent) to build and compile .dll (Windows) or .so (Linux) library files.
8. Be sure the library and optional preferences file are in the particle body force model folder (as specified with Tools > Options > File Locations).
9. Start EDEM then select the required particle body force model category from the Interaction pulldown in the Physics section.
10. Click the **+** drop-down list then select your new particle body force model.

For a detailed example, refer to *Developing a Particle Body Force User Defined Library* on page 20.

Particle Body Force API v2.1.0

The interface IPluginParticleBodyForceV2_1_0 provides these methods:

Method	Description	Type
getDetailsForProperty	Retrieves details for a given property. These properties will then be registered with the system if they do not clash with any existing properties.	Setup
getNumberOfRequiredProperties	Returns the number of custom properties this UDL wants to register with the system.	Setup
getPreferenceFileName	Retrieves the name of the config file used by the UDL.	Setup
isThreadSafe	If the UDL calculateForce() method is thread-safe then this method must return true. Thread safe programming requires a number of conventions and restrictions to be followed. If in doubt set this to return false.	Setup
setup	Initializes the UDL by giving it a chance to read any config files, open temporary files, generate data structures or any other one off setup work.	Setup
usesCustomProperties	Indicates whether the UDL wishes to register or receive custom property data.	Setup
starting	Called to indicate processing is about to begin and the model should allocate any temporary storage and retrieve any file/api/socket handles it may need.	Simulation
externalForce	Called every single time step for every single particle. It allows the user to add a particle body forces (e.g. electromagnetic force, drag force) to particles.	Simulation
stopping	Called to indicate processing is finished and that the model should free any temporary storage and close/release file/api/socket handles.	Simulation

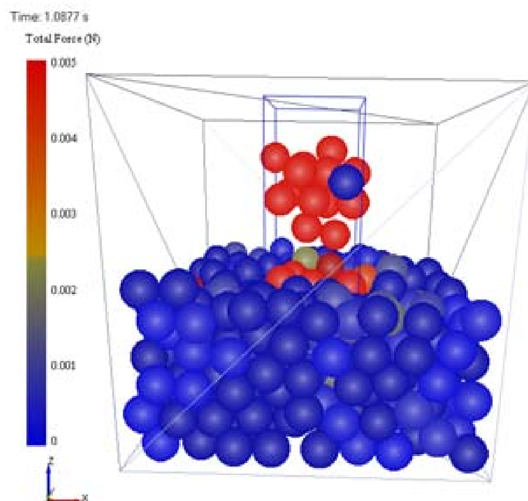
Refer to the online API documentation for full details.

Developing a Particle Body Force User Defined Library

This section describes how to create a Particle Body Force UDL that applies a particle body force to particles in a box. This is analogous of the drag force on particles from an upwards-moving vertical jet of air translating below the box of particles. The box is 100mm x 100 m x 100 mm and is centered at (0,0,0). The box is half full with 460 spherical particles of 5mm radius. The UDL:

- Applies a new force ("**forceZ**") to all particles
- Applies the force only to particles in a specific area
- Decreases the force with height
- Translates the force in the X-axis

NOTE: This example uses Microsoft Visual C++ 2008 Express Edition to compile the contact models. See http://www.microsoft.com/express/download/.



Step 1: Prepare the header files and derive classes

1. Select a working directory on the local machine (e.g. **C:\New_Force**).
2. Copy the following files from src\Api\ParticleBodyForce to your working directory:

    IPluginParticleBodyForce.h
    IPluginParticleBodyForceV2_1_0.h
    PluginParticleBodyForceCore.h
    PluginConstants.h
    Helpers.h

3. Create a file **force_example.txt** in your working directory. Edit the file to contain the lines:

    ```
    1000
    1.1
    ```

    This indicates the force in Z (1000 N) and the velocity (1.1 m/s).

Step 2: Implement core UDL methods

1. Derive a new class from IPluginParticleBodyForceV2_1_0 (found in the header file IPluginParticleBodyForceV2_1_0.h) and save it as C*ForceExample.h*. Refer to the code extract below (the complete file is shown in *Appendix A: Code Used in Examples* on page 39).

```
#if !defined(cforceexample_h)
#define cforceexample_h

#include "IPluginParticleBodyForceV2_1_0.h"
#include <string>
#include <fstream>

class CForceExample : public NApiPbf::IPluginParticleBodyForceV2_1_0
{
       public:
            /**
            * Name of the preferences file to load information
            */
            static const std::string PREFS_FILE;

            /**
             * Constructor, does nothing
             */
            CForceExample();

            /**
             * Destructor, does nothing
             */
            virtual ~CForceExample();

            /**
             * Retrieves the name of the config file used by the plugin.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual void getPreferenceFileName(char
prefFileName[NApi::FILE_PATH_MAX_LENGTH]);

            /**
             * Multithread
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual bool isThreadSafe();


            /*******
             * Complete the rest of the code here
             *******/

            private:

                           double m_forceZ;    /* Force in the z-direction */
                           double m_velocity;  /* Velocity */

};

#endif
```

2. Implement all required methods in *PluginParticleBodyForceCore.h* in a new .cpp file and save it as *ForceExample.cpp*. See below:

```
#include "PluginParticleBodyForceCore.h"
#include "CForceExample.h"

using namespace NApiPbf;

EXPORT_MACRO IPluginParticleBodyForce* GETPBFINSTANCE()
{
    return new CForceExample();
}

EXPORT_MACRO void RELEASEPBFINSTANCE(IPluginParticleBodyForce* plugin)
{
    if (0 != plugin)
    {
        delete plugin;
    }
}

EXPORT_MACRO int GETEFINTERFACEVERSION()
{
    static const int INTERFACE_VERSION_MAJOR = 0x02;
    static const int INTERFACE_VERSION_MINOR = 0x01;
    static const int INTERFACE_VERSION_PATCH = 0x00;

    return (INTERFACE_VERSION_MAJOR << 16 |
            INTERFACE_VERSION_MINOR << 8 |
            INTERFACE_VERSION_PATCH);
}
```

3. Implement all methods in *CForceExample.h* and save it as *CForceExample.cpp*. Refer to the code extract below (the complete file is shown in *Appendix A: Code Used in Examples* on page 39) which shows the implementation of the physics part in the `ECalculateResult exernalForce` method. The external force operates in the direction of the vertical (z) axis, decreasing with respect to the height of the particles. It translates in the x direction over time.

```
    double height;
    double startpoz1;
    double startpoz2;

    //Translate the force in the x-axis
    height = 1.0 + (1000 * (posZ + 50e-3));
    startpoz1 = -0.03 + ((time - 1.0) * m_velocity);
    startpoz2 = 0.0 + ((time - 1.0) * m_velocity);

    if(posX >= startpoz1 && posX <= startpoz2 && posY >= -0.015 && posY <=
0.015)
    {
        //Decreases the force with height
        calculatedForceZ = (m_forceZ / height);
    }

    return eSuccess;
```

Step 3: Create the Library File

If you are compiling on MS Windows:

1. Start the compiler's command prompt:

   Start > All Programs > Visual C++ 9.0 Express Edition > Visual Studio Tools > Visual Studio 2008 Command Prompt

2. Browse to your working directory using the cd command.
3. Compile using the following command:

   ```
   cl /O2 /DWIN32 /LD ForceExample.cpp CForceExample.cpp
   ```

4. The file *ForceExample.dll* is created in your working directory.

If you are compiling on Linux:

1. Open a terminal window and browse to your working directory.
2. Compile using the following command:

   ```
   gcc -O2 -shared -fPIC ForceExample.cpp CForceExample.cpp -o
   ForceExample.so
   ```

   The file *ForceExample.so* is created in your working directory.

Using the New Particle Body Force

1. Start EDEM then open a simulation that will use the new particle body force. You can download an example simulation from the DEM Solutions website.
2. Load your new particle body. Select Tools > Options > File Locations > Particle Body Force then navigate to the location of your newly created dll.
3. Select *Particle Body Force* from the Interaction pulldown in the Physics section.
4. Click the **+** button then select your custom Particle Body Force UDL.
5. Select File > Save.

Particle Factory User Defined Libraries

Using the EDEM API, you can write and compile a custom Particle Factory for creating and initializing particles. A Particle Factory UDL consists of a shared library file and optionally a file containing preferences.

Overview of Creating a New Particle Factory

1. Create a directory to contain the new Particle Factory project.
2. Copy the following header files into the directory:
   - IPluginParticleFactory.h
   - PluginParticleFactoryCore.h
   - IPluginParticleFactoryV2_0_0.h
   - PluginConstants.h
   - Helpers.h
3. Create a new class (CNewParticleFactory) derived from the factory interface you want to use (e.g. IPluginParticleFactoryV2_0_0).
4. Save the class declaration to a header file (e.g. CNewParticleFactory.h). This should consist of declarations of the methods in the chosen interface, any extra methods required and any variables.
5. Implement all of the methods defined in your header file and save as a cpp file (e.g. CNewParticleFactory.cpp).
6. Create a new .cpp file with implementations of the methods in PluginParticleFactoryCore.h. For example, save as NewParticleFactory.cpp
7. Compile all of your src files and link them together in to .dll (Windows) or .so (Linux) library files.
8. Be sure the library and optional preferences file are in the factories folder (as specified with Tools > Options > File Locations).
9. Start EDEM then switch to the Factories tab. Click the Import button then select the new factory from the Import Factory popup. Click OK.

Particle Factory API v2.0.0

The contact model interface IPluginParticleFactoryV2_0_0 provides these methods:

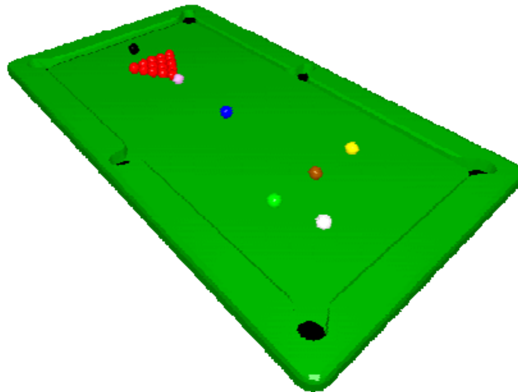
Method	Description	Type
getDetailsForProperty	Retrieves details for a given property. These properties will then be registered with the system if they do not clash with any existing properties.	Setup
getNumberOfRequiredProperties	Returns the number of custom properties this UDL wants to register with the system.	Setup
getPreferenceFileName	Retrieves the name of the config file used by the UDL.	Setup
setup	Initializes the UDL by reading any config files, opening temporary files, generating data structures or any other setup work.	Setup
usesCustomProperties	Indicates whether the UDL wishes to register or receive custom property data.	Setup
starting	Called to indicate processing is about to begin and the model should allocate any temporary storage and retrieve any required file/API/socket handles.	Simulation
createParticle	Called when a new particle is to be created and passed back to EDEM for addition to the simulation. This method is called at least once per timestep.	Simulation
stopping	Called to indicate processing is finished and that the model should free any temporary storage and close/release file/api/socket handles.	Simulation

Refer to the online API documentation for full details.

Developing a Particle Factory User Defined Library

This section describes how to create a new Particle Factory that generates snooker balls and takes a shot on a snooker table.

For MS Windows, this example uses Microsoft Visual C++ 2008 Express Edition to compile the factory. See http://www.microsoft.com/express/download/. For Linux, this example uses the standard GNU Compiler Collection (GCC).



Step 1: Prepare the header files and derive classes

Create a new folder then copy the following header files from `src\Api\Factories` and `src\Api\Core` into your project folder.

- IPluginParticleFactory.h
- PluginParticleFactoryCore.h
- IPluginParticleFactoryV2_0_0.h
- PluginConstants.h
- Helpers.h

Step 2: Implement core UDL methods

1. Derive a class from IPluginParticleFactoryV2_0_0 (found in the header file IPluginParticleFactoryV2_0_0.h) and save it as CSnooker.h.

```
#if !defined(csnooker_h)
#define csnooker_h

#include "IPluginParticleFactoryV2_0_0.h"

class CSnooker : public NApiFactory::IPluginParticleFactoryV2_0_0
{
    public:
        /**
         * Constructor, does nothing
         */
        CSnooker();
```

```
        /**
         * Destructor, does nothing
         */
        virtual ~CSnooker();

        // Implementation of method from IPluginParticleFactory_V2_0_0
        virtual NApi::ECalculateResult createParticle(
                                           double  time,
                                           bool&   particleCreated,
                                           bool&   additionalParticleRequired,
                                           char
type[NApi::API_BASIC_STRING_LENGTH],
                                           double& scale,
                                           double& posX,
                                           double& posY,
                                           double& posZ,
                                           double& velX,
                                           double& velY,
                                           double& velZ,
                                           double& angVelX,
                                           double& angVelY,
                                           double& angVelZ,
                                           double  orientation[9],
                                           NApiCore::ICustomPropertyDataApi_1_0*
propData);

    private:
        /**
         * Number of particles created so far
         */
        unsigned int m_numberCreated;

};

#endif
```

2. Implement all required methods in PluginParticleFactoryCore.h. Save as Snooker.cpp as shown below:

```
#include "PluginParticleFactoryCore.h"
#include "CSnooker.h"

using namespace NApiFactory;

EXPORT_MACRO IPluginParticleFactory* GETFACTORYINSTANCE()
{
    return new CSnooker();
}

EXPORT_MACRO void RELEASEFACTORYINSTANCE(IPluginParticleFactory* instance)
{
    if (0 != instance)
    {
        delete instance;
    }
}

EXPORT_MACRO int GETFACTINTERFACEVERSION()
{
    static const int INTERFACE_VERSION_MAJOR = 0x02;
```

```
    static const int INTERFACE_VERSION_MINOR = 0x00;
    static const int INTERFACE_VERSION_PATCH = 0x00;

    return (INTERFACE_VERSION_MAJOR << 16 |
            INTERFACE_VERSION_MINOR << 8 |
            INTERFACE_VERSION_PATCH);
}
```

3. Implement all methods in CSnooker.h and save it as CSnooker.cpp.

```
#include "CSnooker.h"

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

using namespace NApi;
using namespace NApiCore;
using namespace NApiFactory;

CSnooker::CSnooker() :
          m_numberCreated(0)
{

}

CSnooker::~CSnooker()
{

}

NApi::ECalculateResult CSnooker::createParticle(
                                   double  time,
                                   bool&   particleCreated,
                                   bool&   additionalParticleRequired,
                                   char    type[API_BASIC_STRING_LENGTH],
                                   double& scale,
                                   double& posX,
                                   double& posY,
                                   double& posZ,
                                   double& velX,
                                   double& velY,
                                   double& velZ,
                                   double& angVelX,
                                   double& angVelY,
                                   double& angVelZ,
                                   double  orientation[9],
                                   ICustomPropertyDataApi_1_0* propData)
{

    if(time > 1e-7 || m_numberCreated >=22)
    {
        particleCreated = false;
        additionalParticleRequired = false;
        return eSuccess;
    }

    // Mark that we are creating a particle
    particleCreated = true;

    // Set some fixed values for all particles we create
    velX = 0.0;
```

```
velY = 0.0;
velZ = 0.0;

angVelX = 0.0;
angVelY = 0.0;
angVelZ = 0.0;

for (unsigned int i = 0; i < 9; i++)
{
    orientation[i] = 0.0;
}

scale = 1.0;

// Z position is the same for all aprticles (just above the table
// X and Y vary
posZ = 0.025;

// Set particle specific parameters (location)
switch(m_numberCreated)
{
    //reds
    default:
        strcpy(type,"red");
        if(m_numberCreated <= 4)
        {
            posY = 0.55 - 0.05*(4-m_numberCreated);
            posX = 0.324;
        }
        if(m_numberCreated > 4 && m_numberCreated <=8)
        {
            posY = 0.525 - 0.05*(8-m_numberCreated);
            posX = 0.368;
        }
        if(m_numberCreated > 8 && m_numberCreated <= 11)
        {
            posY = 0.5 - 0.05*(11-m_numberCreated);
            posX = 0.412;
        }
        if(m_numberCreated > 11 && m_numberCreated <= 13)
        {
            posY = 0.475 - 0.05*(13-m_numberCreated);
            posX = 0.456;
        }
        if(m_numberCreated == 14)
        {
            posY = 0.45;
            posX = 0.5;
        }
        break;

     //black
    case(15):
        strcpy(type,"black");
        posY = 0.45;
        posX = 0.15;
        break;

    //pink
    case(16):
        strcpy(type,"pink");
        posY = 0.45;
        posX = 0.55;
```

```
            break;

        //blue
        case(17):
            strcpy(type,"blue");
            posY = 0.45;
            posX = 0.9;
            break;

        //brown
        case(18):
            strcpy(type,"brown");
            posY = 0.45;
            posX = 1.4;
            break;

        //green
        case(19):
            strcpy(type,"green");
            posY = 0.25;
            posX = 1.4;
            break;

        //yellow
        case(20):
            strcpy(type,"yellow");
            posY = 0.65;
            posX = 1.4;
            break;

        //white
        case(21):
            strcpy(type,"white");
            posY = 0.3;
            posX = 1.6;

            // Override the starting velocity
            // We only start with the white ball moving
            velX = -2.5;
            velY = 0.2;
            break;
    }

    // Mark the particle as created
    m_numberCreated++;

    // We are creating 21 particles, if we've done them all
    // then no more are required
    additionalParticleRequired = (22 != m_numberCreated);

    return eSuccess;
}
```

Step 3: Create the Library File

If you are compiling on MS Windows:

1. Start the compiler's command prompt:

   Start > All Programs > Visual C++ 9.0 Express Edition > Visual Studio Tools > Visual Studio 2008 Command Prompt

2. Browse to your working directory using the cd command.
3. Compile using the following command:

   ```
   cl /O2 /DWIN32 /LD Snooker.cpp CSnooker.cpp
   ```

   The file *Snooker.dll* is created in your working directory.

If you are compiling on Linux:

1. Open a terminal window and browse to your working directory.
2. Compile using the following command:

   ```
   gcc -O2 -shared -fPIC Snooker.cpp CSnooker.cpp -o snooker.so
   ```

   The file *snooker.so* is created in your working directory.

Using the New Particle Factory

1. Start EDEM then open a simulation that will use the new factory. You can use the supplied `Snooker_Factory.dem` simulation in the examples folder.
2. Switch to the Factories tab.
3. Delete the in-built *libSnooker* factory then click the Import button.
4. Select your new Snooker library then click OK.
5. Select File > Save.

Custom Properties

With custom properties you can dynamically define custom attributes to use in your simulation. Custom properties store additional attributes (for example, contact duration) for further post-processing. Custom properties can be graphed and exported just like any other attribute. When loaded, contact models, particle body forces, and coupled applications can all use and share data about these new custom properties.

Custom Property Category	Example
Contact	Contact duration, highest force during contact
Geometry	Cumulative impact force, wear
Particle	Residence time, temperature
Simulation	Stickiness of a surface

A custom property is a named group of one or more numeric (double precision floating point) values with an associated unit type. You can define any number of custom properties. Each property has the following attributes:

Attribute	Description
Name	The unique name of the custom property.
State	When properties are first added, they are defined as *tentative*. Tentative properties have not yet had space allocated for them. When simulation starts, tentative properties are finalized and space allocated for them.
Storage Type	The properties data type (currently always a C++ double).
Number of Elements	Indicates the number of elements. Usually a property has one element. A property with a position in 3D space would have three elements (X, Y, and Z).
Units	The unit type, for example length, temperature, charge, velocity etc. If the property has no unit, this is set to None.
Values	Indicates the property’s initial values. Double-click to change the value (provided the custom property is either still tentative or is a simulation property). For custom simulation properties, the Values area also indicates the value for the current timestep.

Any UDL that uses a custom property with the same name and category will share the property’s value(s) and have the ability to manipulate them, provided the two UDL definitions (name, units, number of elements and data type) are identical. If two UDLs have differing definitions then the first UDL only will load. The second UDL will not load and an error is given.

Using Custom Properties

To use custom properties:

- Indicate your UDL wants to use custom properties by setting its usesCustomProperties() method to return true.
- Register each custom property you want to use. For each custom property category (contact, particle etc.), use the methods getNumberOfRequiredProperties() and getDetailsForProperty() to indicate the name, number of elements and unit type of each property. When the UDL is loaded, EDEM calls getNumberOfRequiredProperties() once for each category. getDetailsForProperty() is called once for each property you are registering.

When EDEM starts processing, tentative properties are finalized and EDEM allocates storage for each new property.

Developing a UDL that uses Custom Particle Properties

This section describes how to develop the custom particle property called *Residence Time*. The property measures how long a particle has existed in a simulation.

NOTE: This example uses Microsoft Visual C++ 2008 Express Edition to compile the contact models. See http://www.microsoft.com/express/download/.

1. Select a working directory on the local machine (e.g. **C:\UDPA**).
2. Copy the following files from the EDEM\src folder to your working directory:

   IPluginParticleBodyForce.h
   IPluginParticleBodyForceV2_1_0.h
   PluginParticleBodyForceCore.h
   PluginConstants.h
   Helpers.h

Create and Compile the Custom Particle Property

Step 1: Implement all methods

Create a new Particle Body Force UDL as normal.

1. Derive from the IPluginParticleBodyForceV2_1_0 interface and implement all of its methods. Call the new class CResidenceTime.
2. Implement the functions from PluginParticleBodyForceCore.h in a separate source file (e.g. ResidenceTime.cpp).

Step 2: Define custom property details

1. Define the getNumberOfRequiredProperties() method to register one custom particle property.
2. Define the getDetailsForProperty() method to retrieve details for the particle property.
3. Set the unitType of the property to time.

Refer to the code example below.

```
unsigned int CResidenceTime::getNumberOfRequiredProperties(
                                const NApi::EPluginPropertyCategory category)
{
    // this plug-in registers 1 custom particle property
    if (eParticle == category)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

bool CResidenceTime::getDetailsForProperty(
                         unsigned int                      propertyIndex,
                         NApi::EPluginPropertyCategory     category,
                         char
name[NApi::CUSTOM_PROP_MAX_NAME_LENGTH],
                         NApi::EPluginPropertyDataTypes& dataType,
                         unsigned int&                     numberOfElements,
                         NApi::EPluginPropertyUnitTypes& unitType)
{
    if (eParticle == category &&
        0 == propertyIndex)
    {
        strcpy(name, RESIDENCE_TIME_PROPERTY.c_str());
        dataType         = eDouble;
        numberOfElements = 1;
        unitType         = eTime;
        return true;
    }
    else
    {
        return false;
    }
}
```

Step 3: Define how manipulate the custom property

Define the externalForce() method to specify the custom particle property added to particles each time step. To track the residence time, add the timestep to the current residence time value:

```
// Cache pointers to the custom properties we wish to use
double* residenceTimeDelta =
        particlePropData->getDelta(RESIDENCE_TIME_PROPERTY.c_str());

// update the residence time for this particle by adding the time step
// to the current value
*residenceTimeDelta += timestep;

return eSuccess;
```

Step 4: Create the Library File

If you are compiling on MS Windows:

1. Start the compiler's command prompt:

   Start > All Programs > Visual C++ 9.0 Express Edition > Visual Studio Tools > Visual Studio 2008 Command Prompt

2. Browse to your working directory using the cd command.
3. Compile using the following command:

   ```
   cl /O2 /DWIN32 /LD ResidenceTime.cpp CResidenceTime.cpp
   ```

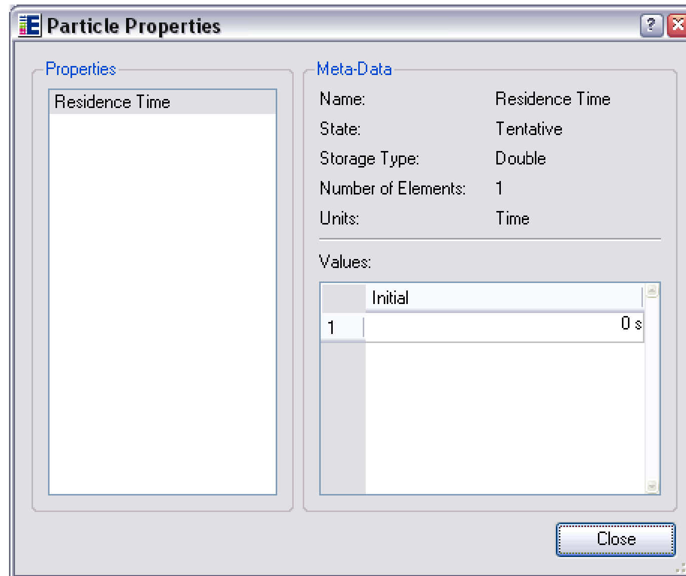
If you are compiling on Linux:

3. Open a terminal window and browse to your working directory.
4. Compile using the following command:

   ```
   gcc -O2 -shared -fPIC ResidenceTime.cpp CResidenceTime.cpp -o
   ResidenceTime.so
   ```

Using the New Custom Particle Property

1. Start EDEM then open a simulation that will use the new custom particle property. You can download an example simulation from the DEM Solutions website.
2. Load your new particle body. Select Tools > Options > File Locations > Particle Body Force then navigate to the location of your newly created dll.
3. Select *Particle Body Force* from the Interaction pulldown in the Physics section.
4. Click the **+** button then select *ResidenceTime.dll*.
5. To verify the custom particle property, select Tools > Particle. The Particle Properties window is displayed:



6. Check *Residence Time* is listed as a tentative property. The custom property will move to the finalized section when you start the simulation.

Glossary

A

API: *Application Programming Interface*. An API is an interface defining how a program requests services from libraries and/or operating systems. The API determines the vocabulary and calling conventions to use the services. It usually includes specifications for routines, data structures, object classes and protocols to communicate between the requesting software and the library.

C

Class: A class is a programming language construct used to create objects. It describes the state and behavior that the created objects all share. An object created by a class is an instance of the class, and the class that created that instance can be considered as the type of that object.

Constructor: a constructor is a special method which puts the object's members into a valid state. It is a block of statements called when an object is created, either when it is declared or dynamically constructed.

D

Data Field: A place where data is stored.

Data Type: A data type (or datatype) is a set of values and the operations on those values.

Destructor: A destructor is a method which is automatically invoked when the object is destroyed. Its main purpose is to clean up and to free the resources which were acquired by the object along its life cycle and unlink it from other objects or resources invalidating any references in the process.

F

Function: See *Method*.

H

Header File: A header file (or include file) is a file that a compiler automatically includes when processing another source file. Typically, the inclusion of header files are specified via compiler directives at the beginning (or head) of the other source file.

L

Library: A library is a collection of classes used to develop software. Libraries contain code and data that provide services to independent programs. This allows the sharing and changing of code and data in a modular fashion. Some executables

are both standalone programs and libraries, but most libraries are not executables.

M

Makefile: *Make* is a utility for automatically building executable programs and libraries from source code. Files called makefiles specify how to derive the target program from each of its dependencies.

Method: A method (or procedure, function, or subroutine) is a portion of code within a larger program which performs a specific task and is relatively independent of the remaining code.

Multithreaded: See *Thread*.

N

Namespace: A namespace is a context for identifiers. It groups related methods together.

Namespace Member: A method that belongs to a particular namespace.

O

Object: An object is a data structure consisting of data fields and methods that can manipulate those fields. Typically, when calling a method from some object, the object itself should be passed as a parameter to the method.

S

Shared Library: See *Library*.

T

Thread: A thread results from a fork of a computer program into two or more concurrently running tasks. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

U

User Defined Library (UDL): A computer program that interacts with a host application. Also known as a plugin.

Appendix A: Code Used in Examples

CCohesionModel.h

The following code is used in *Developing a Plugin Contact Model* on page 10.

```
#if !defined(ccohesionmodel_h)
#define ccohesionmodel_h

#include <string>
#include <fstream>

#include "IPluginContactModelV2_2_0.h"

using namespace std;

/**
 * This class provides an implementation of IPluginContactModelV2_2_0
 * That adds performs hertz mindlin contact force calculation
 *
 * NOTE: This version of the hertz-mindlin code is provided only
 * as an example of how to create a contact model plugin and is not
 * updated on a regular basis.
 */
class CCohesionModel : public NApiCm::IPluginContactModelV2_2_0
{
    public:

        /**
         * Constructor, does nothing
         */
        CCohesionModel();

        /**
         * Destructor, does nothing
         */
        virtual ~CCohesionModel();

        /**
         * Sets prefFileName to the empty string as this plugin has no
         * configuration
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual void getPreferenceFileName(char prefFileName[NApi::FILE_PATH_MAX_LENGTH]);

        /**
         * Returns true to indicate plugin is thread safe
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual bool isThreadSafe();

        /**
         * Returns true to indicate the plugin uses custom properties
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual bool usesCustomProperties();

        /**
         * Does nothing
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual bool setup(NApiCore::IApiManager_1_0& apiManager,
                           const char prefFile[]);
```

```
        /**
         * Does nothing
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual bool starting(NApiCore::IApiManager_1_0& apiManager);


        /**
         * Does nothing
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual void stopping(NApiCore::IApiManager_1_0& apiManager);

        /**
         *
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual NApi::ECalculateResult calculateForce(
                                                double       time,
                                                double       timestep,
                                                int          elem1Id,
                                                const char   elem1Type[],
                                                double       elem1Mass,
                                                double       elem1ShearMod,
                                                double       elem1Poisson,
                                                double       elem1ContactCurvature,
                                                double       elem1PhysicalCurvature,
                                                double       elem1PosX,
                                                double       elem1PosY,
                                                double       elem1PosZ,
                                                double       elem1ComX,
                                                double       elem1ComY,
                                                double       elem1ComZ,
                                                double       elem1VelX,
                                                double       elem1VelY,
                                                double       elem1VelZ,
                                                double       elem1AngVelX,
                                                double       elem1AngVelY,
                                                double       elem1AngVelZ,
                                                double       elem1Charge,
                                                double       elem1WorkFunction,
                                                const double elem1Orientation[9],
                                                NApiCore::ICustomPropertyDataApi_1_0*
elem1PropData,
                                                bool         elem2IsSurf,
                                                int          elem2Id,
                                                const char   elem2Type[],
                                                double       elem2Mass,
                                                double       elem2Area,
                                                double       elem2ShearMod,
                                                double       elem2Poisson,
                                                double       elem2ContactCurvature,
                                                double       elem2PhysicalCurvature,
                                                double       elem2PosX,
                                                double       elem2PosY,
                                                double       elem2PosZ,
                                                double       elem2ComX,
                                                double       elem2ComY,
                                                double       elem2ComZ,
                                                double       elem2VelX,
                                                double       elem2VelY,
                                                double       elem2VelZ,
                                                double       elem2AngVelX,
                                                double       elem2AngVelY,
                                                double       elem2AngVelZ,
                                                double       elem2Charge,
                                                double       elem2WorkFunction,
                                                const double elem2Orientation[9],
```

```
                                 NApiCore::ICustomPropertyDataApi_1_0* elem2PropData,
                                 NApiCore::ICustomPropertyDataApi_1_0* contactPropData,
                                 NApiCore::ICustomPropertyDataApi_1_0* simulationPropData,
                                       double       coeffRest,
                                       double       staticFriction,
                                       double       rollingFriction,
                                       double       contactPointX,
                                       double       contactPointY,
                                       double       contactPointZ,
                                       double       normalOverlap,
                                       double&      tangentialOverlapX,
                                       double&      tangentialOverlapY,
                                       double&      tangentialOverlapZ,
                                       double&      calculatedNormalForceX,
                                       double&      calculatedNormalForceY,
                                       double&      calculatedNormalForceZ,
                                       double&      calculatedUnsymNormalForceX,
                                       double&      calculatedUnsymNormalForceY,
                                       double&      calculatedUnsymNormalForceZ,
                                       double&      calculatedTangentialForceX,
                                       double&      calculatedTangentialForceY,
                                       double&      calculatedTangentialForceZ,
                                       double&      calculatedUnsymTangentialForceX,
                                       double&      calculatedUnsymTangentialForceY,
                                       double&      calculatedUnsymTangentialForceZ,
                                       double&      calculatedElem1AdditionalTorqueX,
                                       double&      calculatedElem1AdditionalTorqueY,
                                       double&      calculatedElem1AdditionalTorqueZ,
                                       double&      calculatedElem1UnsymAdditionalTorqueX,
                                       double&      calculatedElem1UnsymAdditionalTorqueY,
                                       double&      calculatedElem1UnsymAdditionalTorqueZ,
                                       double&      calculatedElem2AdditionalTorqueX,
                                       double&      calculatedElem2AdditionalTorqueY,
                                       double&      calculatedElem2AdditionalTorqueZ,
                                       double&      calculatedElem2UnsymAdditionalTorqueX,
                                       double&      calculatedElem2UnsymAdditionalTorqueY,
                                       double&      calculatedElem2UnsymAdditionalTorqueZ,
                                       double&      calculatedChargeMovedToElem1);

               /**
         * Returns 0
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual unsigned int getNumberOfRequiredProperties();

        /**
         * Does nothing
         *
         * Implementation of method from IPluginContactModelV2_2_0
         */
        virtual bool getDetailsForProperty(
                             unsigned int                    propertyIndex,
                             NApi::EPluginPropertyCategory   category,
                             char
name[NApi::CUSTOM_PROP_MAX_NAME_LENGTH],
                             NApi::EPluginPropertyDataTypes& dataType,
                             unsigned int&                   numberOfElements,
                             NApi::EPluginPropertyUnitTypes& unitType);

               /**
         * Name of the preferences file to load bond information
         * from
         */
        static const std::string PREFS_FILE;

               private:
                       double m_energyDensity;
};

#endif
```

CCohesionModel.cpp

The following code is used in *Developing a Plugin Contact Model* on page 10.

```
#include "CCohesionModel.h"

#include "Helpers.h"

using namespace NApi;
using namespace NApiCore;
using namespace NApiCm;

const string CCohesionModel::PREFS_FILE = "cohesion_prefs.txt";

CCohesionModel::CCohesionModel()
{
    ;
}

CCohesionModel::~CCohesionModel()
{
    ;
}

void CCohesionModel::getPreferenceFileName(char prefFileName[FILE_PATH_MAX_LENGTH])
{
    // Copy in pref file name
    strcpy(prefFileName, PREFS_FILE.c_str());
}

bool CCohesionModel::isThreadSafe()
{
    // thread safe
    return true;
}

bool CCohesionModel::usesCustomProperties()
{
    // Does not use custom properties
    return false;
}

bool CCohesionModel::setup(NApiCore::IApiManager_1_0& apiManager,
                           const char prefFile[])
{
        //Read in the preference file
    ifstream prefsFile(prefFile);

    if(!prefsFile)
    {
        return false;
    }
    else
    {
        while (prefsFile)
        {
            prefsFile >> m_energyDensity;
        }
    }
    return true;
}

bool CCohesionModel::starting(NApiCore::IApiManager_1_0& apiManager)
{
    return true;
}

void CCohesionModel::stopping(NApiCore::IApiManager_1_0& apiManager)
{
    ;
```

```
}

ECalculateResult CCohesionModel::calculateForce(double       time,
                                               double       timestep,
                                               int          elem1Id,
                                               const char   elem1Type[],
                                               double       elem1Mass,
                                               double       elem1ShearMod,
                                               double       elem1Poisson,
                                               double       elem1ContactCurvature,
                                               double       elem1PhysicalCurvature,
                                               double       elem1PosX,
                                               double       elem1PosY,
                                               double       elem1PosZ,
                                               double       elem1ComX,
                                               double       elem1ComY,
                                               double       elem1ComZ,
                                               double       elem1VelX,
                                               double       elem1VelY,
                                               double       elem1VelZ,
                                               double       elem1AngVelX,
                                               double       elem1AngVelY,
                                               double       elem1AngVelZ,
                                               double       elem1Charge,
                                               double       elem1WorkFunction,
                                               const double elem1Orientation[9],
                                               NApiCore::ICustomPropertyDataApi_1_0*
elem1PropData,
                                               bool         elem2IsSurf,
                                               int          elem2Id,
                                               const char   elem2Type[],
                                               double       elem2Mass,
                                               double       elem2Area,
                                               double       elem2ShearMod,
                                               double       elem2Poisson,
                                               double       elem2ContactCurvature,
                                               double       elem2PhysicalCurvature,
                                               double       elem2PosX,
                                               double       elem2PosY,
                                               double       elem2PosZ,
                                               double       elem2ComX,
                                               double       elem2ComY,
                                               double       elem2ComZ,
                                               double       elem2VelX,
                                               double       elem2VelY,
                                               double       elem2VelZ,
                                               double       elem2AngVelX,
                                               double       elem2AngVelY,
                                               double       elem2AngVelZ,
                                               double       elem2Charge,
                                               double       elem2WorkFunction,
                                               const double elem2Orientation[9],
                                               NApiCore::ICustomPropertyDataApi_1_0*
elem2PropData,
                                               NApiCore::ICustomPropertyDataApi_1_0*
contactPropData,
                                               NApiCore::ICustomPropertyDataApi_1_0*
simulationPropData,
                                               double       coeffRest,
                                               double       staticFriction,
                                               double       rollingFriction,
                                               double       contactPointX,
                                               double       contactPointY,
                                               double       contactPointZ,
                                               double       normalOverlap,
                                               double&      tangentialOverlapX,
                                               double&      tangentialOverlapY,
                                               double&      tangentialOverlapZ,
                                               double&      calculatedNormalForceX,
                                               double&      calculatedNormalForceY,
                                               double&      calculatedNormalForceZ,
```

```
                                double&      calculatedUnsymNormalForceX,
                                double&      calculatedUnsymNormalForceY,
                                double&      calculatedUnsymNormalForceZ,
                                double&      calculatedTangentialForceX,
                                double&      calculatedTangentialForceY,
                                double&      calculatedTangentialForceZ,
                                double&      calculatedUnsymTangentialForceX,
                                double&      calculatedUnsymTangentialForceY,
                                double&      calculatedUnsymTangentialForceZ,
                                double&      calculatedElem1AdditionalTorqueX,
                                double&      calculatedElem1AdditionalTorqueY,
                                double&      calculatedElem1AdditionalTorqueZ,
                                double&      calculatedElem1UnsymAdditionalTorqueX,
                                double&      calculatedElem1UnsymAdditionalTorqueY,
                                double&      calculatedElem1UnsymAdditionalTorqueZ,
                                double&      calculatedElem2AdditionalTorqueX,
                                double&      calculatedElem2AdditionalTorqueY,
                                double&      calculatedElem2AdditionalTorqueZ,
                                double&      calculatedElem2UnsymAdditionalTorqueX,
                                double&      calculatedElem2UnsymAdditionalTorqueY,
                                double&      calculatedElem2UnsymAdditionalTorqueZ,
                                double&      calculatedChargeMovedToElem1)
{
    // This contact model assumes that the Physical and the Contact radius
    // are the same

    // The unit vector from element 1 to the contact point
    CSimple3DPoint  contactPoint = CSimple3DPoint(contactPointX, contactPointY,
contactPointZ);
    CSimple3DVector unitCPVect   = contactPoint - CSimple3DPoint(elem1PosX, elem1PosY,
elem1PosZ);
    unitCPVect.normalise();

    // Cohesion
    // R^2 - (R - d)^2

    double nCohesiveFactor         = m_energyDensity;
    double nRadiusOfOverlapSquared = 2 * elem1PhysicalCurvature * normalOverlap;
    double nArea                   = PI * nRadiusOfOverlapSquared;
    CSimple3DVector F_cohesive     = unitCPVect * nCohesiveFactor * nArea;

    calculatedNormalForceX = F_cohesive.dx();
    calculatedNormalForceY = F_cohesive.dy();
    calculatedNormalForceZ = F_cohesive.dz();

    return eSuccess;
}


unsigned int CCohesionModel::getNumberOfRequiredProperties()
{
    return 0;
}

bool CCohesionModel::getDetailsForProperty(unsigned int                  propertyIndex,

NApi::EPluginPropertyCategory   category,
                                                                              char
name[NApi::CUSTOM_PROP_MAX_NAME_LENGTH],

NApi::EPluginPropertyDataTypes& dataType,
                                                                              unsigned int&
numberOfElements,

NApi::EPluginPropertyUnitTypes& unitType)
{
    return false;
}
```

CForceExample.h

The following code is used in *Developing a Plugin Particle Body Force* on page 20.

```
#if !defined(cforceexample_h)
#define cforceexample_h

#include "IPluginParticleBodyForceV2_1_0.h"
#include <string>
#include <fstream>

class CForceExample : public NApiPbf::IPluginParticleBodyForceV2_1_0
{
        public:
            /**
            * Name of the preferences file to load information
            */
            static const std::string PREFS_FILE;

            /**
             * Constructor, does nothing
             */
            CForceExample();

            /**
             * Destructor, does nothing
             */
            virtual ~CForceExample();

            /**
             * Retrieves the name of the config file used by the plugin.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual void getPreferenceFileName(char
prefFileName[NApi::FILE_PATH_MAX_LENGTH]);

            /**
             * Multithread
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual bool isThreadSafe();

            /**
             * If the plugin implementation wishes to register or receive custom
             * property data then this method must return true.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual bool usesCustomProperties();

            /**
             * Initialises the plugin by giving it a chance to read any config
             * files.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual bool setup(NApiCore::IApiManager_1_0& apiManager,
                               const char prefFile[]);

            /**
             * Called to indicate processing is about to begin.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual bool starting(NApiCore::IApiManager_1_0& apiManager);

            /**
```

```
             * Called to indicate processing is finished.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual void stopping(NApiCore::IApiManager_1_0& apiManager);

            /**
             * This function is called every single time step for every single
             * particle.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */

             virtual NApi::ECalculateResult externalForce(
                                double       time,
                                double       timestep,
                                int          id,
                                const char   type[],
                                double       mass,
                                double       volume,
                                double       posX,
                                double       posY,
                                double       posZ,
                                double       velX,
                                double       velY,
                                double       velZ,
                                double       angVelX,
                                double       angVelY,
                                double       angVelZ,
                                double       charge,
                                const double orientation[9],
                                NApiCore::ICustomPropertyDataApi_1_0* particlePropData,
                                NApiCore::ICustomPropertyDataApi_1_0* simulationPropData,
                                double&      calculatedForceX,
                                double&      calculatedForceY,
                                double&      calculatedForceZ,
                                double&      calculatedTorqueX,
                                double&      calculatedTorqueY,
                                double&      calculatedTorqueZ);

            /**
             * Returns the number of custom properties this plugin wishes to
             * register with the system.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual unsigned int getNumberOfRequiredProperties();

            /**
             * Retrieves details for a given property.
             *
             * Implementation from IPluginParticleBodyForceV2_1_0
             */
            virtual bool getDetailsForProperty(
                             unsigned int                    propertyIndex,
                             NApi::EPluginPropertyCategory   category,
                             char
name[NApi::CUSTOM_PROP_MAX_NAME_LENGTH],
                             NApi::EPluginPropertyDataTypes& dataType,
                             unsigned int&                   numberOfElements,
                             NApi::EPluginPropertyUnitTypes& unitType);

            private:

                              double m_forceZ; /* Force in the z-direction */

                              double m_velocity; /* Velocity */

};

#endif
```

CForceExample.cpp

The following code is used in *Developing a Plugin Particle Body Force* on page 20.

```
#include "CForceExample.h"
#include "Helpers.h"

using namespace NApiPbf;
using namespace NApi;
using namespace NApiCore;
using namespace std;

const string CForceExample::PREFS_FILE = "force_example.txt";

CForceExample::CForceExample()
{
        ;
}

CForceExample::~CForceExample()
{
        ;
}

void CForceExample::getPreferenceFileName(char prefFileName[FILE_PATH_MAX_LENGTH])
{
        // Copy in pref file name
        strcpy(prefFileName, PREFS_FILE.c_str());
}

bool CForceExample::isThreadSafe()
{
        // Thread safe
        return true;
}

bool CForceExample::usesCustomProperties()
{
        //Does not use custom properties
        return false;
}

bool CForceExample::setup(NApiCore::IApiManager_1_0& apiManager,
                          const char prefFile[])
{
        //Read in the preference file
        ifstream prefsFile(prefFile);

        if(!prefsFile)
        {
                return false;
        }
        else
        {
                while (prefsFile)
                {
                        prefsFile >> m_forceZ >> m_velocity;
                }
        }
        return true;
}

bool CForceExample::starting(NApiCore::IApiManager_1_0& apiManager)
{
        return true;
}

void CForceExample::stopping(NApiCore::IApiManager_1_0& apiManager)
{
        ;
```

```
}

ECalculateResult CForceExample::externalForce(
                                double       time,
                                double       timestep,
                                int          id,
                                const char   type[],
                                double       mass,
                                double       volume,
                                double       posX,
                                double       posY,
                                double       posZ,
                                double       velX,
                                double       velY,
                                double       velZ,
                                double       angVelX,
                                double       angVelY,
                                double       angVelZ,
                                double       charge,
                                const double orientation[9],
                                NApiCore::ICustomPropertyDataApi_1_0* particlePropData,
                                NApiCore::ICustomPropertyDataApi_1_0* simulationPropData,
                                double&      calculatedForceX,
                                double&      calculatedForceY,
                                double&      calculatedForceZ,
                                double&      calculatedTorqueX,
                                double&      calculatedTorqueY,
                                double&      calculatedTorqueZ)
{
        double height;
        double startpoz1;
        double startpoz2;

        //Translate the force in the x-axis
        height = 1.0 + (1000 * (posZ + 50e-3));
        startpoz1 = -0.03 + ((time - 1.0) * m_velocity);
        startpoz2 = 0.0 + ((time - 1.0) * m_velocity);

        if(posX >= startpoz1 && posX <= startpoz2 && posY >= -0.015 && posY <= 0.015)
        {
                //Decreases the force with height
                calculatedForceZ = (m_forceZ / height);
        }

        return eSuccess;
}

unsigned int CForceExample::getNumberOfRequiredProperties()
{
        return 0;
}

bool CForceExample::getDetailsForProperty(
                             unsigned int                     propertyIndex,
                             NApi::EPluginPropertyCategory    category,
                             char
name[NApi::CUSTOM_PROP_MAX_NAME_LENGTH],
                             NApi::EPluginPropertyDataTypes&  dataType,
                             unsigned int&                    numberOfElements,
                             NApi::EPluginPropertyUnitTypes&  unitType)
{
        return false;
}
```

Index

API, 37
calculateForce, 9, 25
Class, 37
Cohesion Contact Model, 10, 14
Compiler, 13, 23, 31, 35
Constructor, 37
Custom Properties, 32, 33
Data Field, 37
Data Type, 37
Destructor, 37
externalForce, 19, 35
Force, 20, 23, 33, 36
Function, 37
getDetailsForProperty, 9, 19, 25
getNumberOfRequiredProperties, 9, 19, 25
getPreferenceFileName, 9, 19, 25
Header File, 37
Hertz-Mindlin, 14
Integrated Development Environment, 4
isThreadSafe, 9, 19
Library, 37
Makefile, 38
Method, 38
Multithreaded, 38
Namespace, 38
Object, 38
Preferences File, 8, 18, 24
Sequence Diagram, 4
setup, 9, 19, 25
Shared Library, 38
Simulation Sequence, 5
starting, 9, 19, 25
stopping, 9, 19, 25
Thread, 38
Time, 10, 11, 20, 21, 26
User Defined External Forces, 18
usesCustomProperties, 9, 19, 25