Altair MotionView 2019 Tutorials

MV-1060: Introduction to MDL

altairhyperworks.com

MV-1060: Introduction to MDL

In this tutorial, you will learn how to:

- Create a model using the Model Definition Language (MDL).
- Run a dynamic simulation of this model for a time of 2 seconds and 500 steps.
- Plot the rotation of the pendulum about the global X-axis and view the animation.

MDL stands for Model Definition Language. A MotionView model is an object that holds the information in the form of this language which is required to describe a mechanical system. The complete information about the model is stored in the MDL format. MDL is an **ASCII programmable** language.

Some benefits of MDL include:

- Opening and editing in any text editor
- Assisting with model debugging
- Using conditional statements "if" for custom modeling requirements
- Building modular and reusable models
- Parameterizing the models
- Use modeling entities which are not available through GUI (for example, CommandSets)

Section 1: Entities in MDL

A modeling entity is saved to MDL in the form of MDL statements. All MDL statements begin with an asterisk (*).

There are two types of entities:

- General Entities
- Definition Based Entities

General Entities

- Have one statement to define the entity. They may have one or more statements to set their properties.
- Some examples include points, bodies, joints, etc.
- Each general entity has certain properties consistent with its type. For example, a point has the properties x-coordinate, y-coordinate, z-coordinate, label, state, and varname (variable name).

Definition Based Entities

- Are defined through a block statement, called definition, and its instance is created in a model by an instantiation statement.
- The block generally begins with a `*Define()` statement and end with a `*EndDefine()` statement.
- The entity (or block) is comprised of a series of other MDL entities or members.
- These entities are reusable. Once defined, the same entity-definition may be instantiated several times within the same model or different model files.

Some of the commonly used user-defined entities are outlined in the table below:

Entity	Description
System	A system entity defines a collection of modeling entities. These definitions may be used repeatedly within the same model or different MDL model files. A model can be organized into different systems. Examples of system entities include SLA suspension system, wiper blade system, and power-train system. Systems can be hierarchical in nature (for example, a system can be a child of another system).
Assembly	An assembly is similar to a system entity, except that the definition resides in a separate file than the model file.
Analysis	An analysis is a collection of entities (bodies, joints, etc.) describing a particular analysis task or event applied to a model. For example, a static ride analysis is one of the analysis that can be applied to a model. An analysis can only be instantiated under Model (the top level root system). A system can be a child of an analysis, however the reverse is not true.
Dataset	A dataset is a collection of user-defined variables of type integer, real, string, Boolean, or filename. These variables can be referred or parameterized to other entity properties. Datasets are displayed in a tabular form, thereby offering a single window to modify a model. Generally, design variables are collectively defined in the form of a dataset. A dataset can be instantiated within a system or an analysis.
Template	A template is a utility that uses the Templex program in HyperWorks. It can be used to create user-defined calculations and codes embedded into the model. The output of such code can be written out to solver deck or execute another program. Another use is to implement solver statements and commands not supported by MDL and to generate text reports.

Note The system, assembly, and analysis are together referred to as container entities (or simply containers).

Section 2: Properties of Entities

- Each entity has variable, label, and other properties associated with it.
- Each entity should have a unique variable name.
- Following is the recommended convention for variable names which allows the user to identify the modeling entity during debugging. You are not restricted to this nomenclature, however you are encouraged to adopt it.

This list of entities and their properties is not comprehensive. For a complete list, refer to the *MDL Language Reference* on-line help.

General Entities	Naming Convention	Properties
Point	p_	x, y, z, label, state, varname
Body	b_	mass, IXX, IYY, IZZ, IXY, IYZ, IXZ, cg, cm, im, lprf, label, state, varname
RevJoint	j_	b1, b2, i, j, id
Vector	v_	x, y, z, label, state, varname
Marker	m_	body, flt, x-axis, y-axis, z-axis, origin
ActionReactionForce	frc_	b1, b2, fx, fy, fz, id, tx, ty, tz

General entities, their naming conventions, and properties

Definition Based Entities	Naming Convention	Properties
System	sys_	Label, varname, state
Analysis	ana_	Label, varname, state
Dataset	ds_	Label, varname, state
Template	tmplt_	Label, varname, state

User-defined entities, their naming conventions, and properties

To access entity properties; use the entity varname, followed by a dot separator, followed by the property. Below are some examples:

Entity Varname	Varname Represents
`b_knuckle`	A body representing the knuckle in the mechanical system.
`p_knuckle_cg`	A point representing the center of mass point for the knuckle body.

Entity Property Name	Property Accessed
`b_knuckle.cm`	The center of mass marker of the knuckle body, `b_knuckle`.
`b_knuckle.cm.id`	The ID of the center of mass marker of the knuckle body, `b_knuckle`.
`p_knuckle_cg.x`	The x coordinate of `p_knuckle_cg`.

Section 3: Global Variables

MotionView comes with Global Variables, by default, which are available for use anywhere in the model. These variables are case sensitive.

The table below lists some commonly used keywords and what they represent:

Keyword	Refers to
B_Ground	Ground body
P_Global_Origin	Global Origin
V_Global_X, V_Global_Y, V_Global_Z	Vectors along the global XYZ axes
Global_Frame	Global reference marker
MODEL	Reference to the top level system of the model

Common keywords in MotionView

Section 4: MDL Statement Classification

Topology statements

These are statements that define an entity and establish topological relation between one entity and the other. For example, `*Body(b_body, "Body", p_cg)`. In this example, the `*Body` statement defines a body having its CG at point `p_cg`. Through this statement the body (`b_body`) is topologically connected to point `p_cg`.

Property or Set Statements

These statements assign properties to the entities created by topological entities. For example, `*SetBody()` is a property statement that assign mass and inertia properties to a body defined using `*Body()`. Since most of the property statements begin with "`*Set`", they are generally referred as Set statements.

Definition and Data

Building upon the concept of a definition block, these terminologies are used specifically with regard to container entities such as Systems, Assembly, and Analysis.

The block of statements when contained within a `*Define()` block are termed as a Definition. The statements within the block may include:

1. Topology statements that define entities.
2. Set statements that assign properties. These Set statements within a definition block are called "Default Sets", as they are considered as default values for the entities in the definition.

Any statements or block that resides outside the context of `*Define()` block are termed as Data. These include:

1. Set statements within a `*BeginContext()` block that relate to entities within a system, assembly, or analysis definition.
2. Some of the *Begin statements, such as `*BeginAssemblySelection` and `*BeginAnalysis`.

Section 5: MDL Model File Overview

- MDL model file is an ASCII file; it can be edited using any text editor.
- All statements in a model are contained within a `*BeginMDL()` - `*EndMDL()` block.
- The syntax of the MDL statement is an asterisk (*) followed by a valid statement with its arguments defined.
- Statements without a leading asterisk (*) are considered comments. In this tutorial, comment statements are preceded by // to improve readability. The comments are not read in by the MotionView graphical user interface and are removed if the model MDL is saved back or saved to a different file.

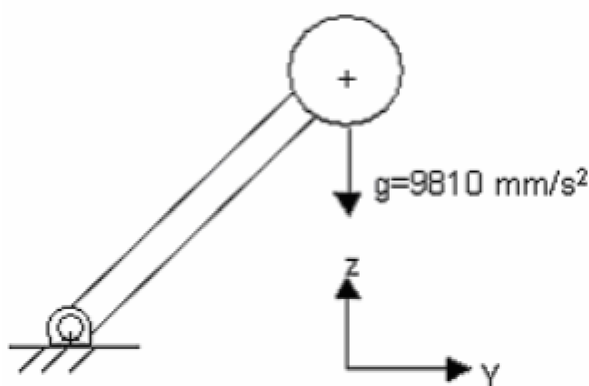
MDL accepts statements in any order, with a few exceptions.

To help you learn this language, the code in the tutorial examples will follow this structure:

```
//comments about the MDL file

*BeginMDL(argument list)

//Topology section

*Point…

*Body…

*System(…)

// definitions sub-section

*DefineSystem(..)…

..

.*EndDefine()

//Property of entities directly in *BeginMDL()//Property section for
entities within Systems and analysis

*BeginContext()

..

..

*EndContext()

.

*EndMDL
```

Exercise: Build a Pendulum Model using MDL Statements

The figure below shows the schematic diagram of a pendulum. The pendulum is connected to the ground through a revolute joint at the global origin. The pendulum falls freely under gravity, which acts in the negative global-Z direction. Geometry and inertia properties are shown in the figure. The center of mass of the pendulum is located at (0, 10, 10).



Schematic representation of the pendulum

The following MDL statements are used in this exercise:

- `*BeginMdl()`
- `*EndMdl()`
- `*Point()`
- `*Body()`
- `*Graphic() - cylinder`
- `*Graphic() - sphere`
- `*RevJoint()`
- `*Output() - output on entities`
- `*SetPoint()`
- `*SetBody()`

Step 1: Create an MDL model file.

1. In a text editor, create the following comment statements describing the purpose of the MDL model file:

```
//Pendulum falling under gravity

//date
```

2. Create a `*BeginMdl() - *EndMdl()` block to signify the beginning and end of the MDL model file. Create all other MDL model file statements between these block statements:

   The syntax for the `*BeginMdl()` statement is:

   ```
   *BeginMdl(model_name, "model_label")
   ```

   where

`model_name`	The variable name of the model.
`model_label`	The descriptive label of the model.

   For this model, use:

   ```
   *BeginMdl(pendulum, "Pendulum Model")
   ```

   ```
   *EndMdl()
   ```

   It is strongly recommended that you look for the syntax of the corresponding statements by invoking the online Help and typing the statement in the Index. In MDL statements, only the keywords are case sensitive.

Step 2: Create the entity declarations required for the problem.

1. Create a point where the pendulum pivot would be placed using a `*Point()` statement. The syntax is:

   ```
   *Point(point_name, "point_label", [point_num])
   ```

   where:

`point_name`	The variable name of the point.
`point_label`	The descriptive label of the point.
`point_num`	An optional integer argument assigned to the point as its entity number.

   For this problem, you will need `point_name` and `point_label`.

   ```
   //Points
   *Point(p_pendu_pivot, "Pivot Point")
   ```

2. Using the same `*Point` statement create another point which would be pendulum center of mass:

   ```
   *Point(p_pendu_cm, "Pendulum CM")
   ```

3. Use the `*Body()` statement to define the ball's body. The syntax is:

```
*Body(body_name, "body_label", [cm_origin], [im_origin], [lprf_origin],
[body_num])
```

where:

`body_name`	The variable name of the body.
`body_label`	The descriptive label of the body appearing in the graphical display of the body.
`cm_origin`	An optional argument for the center of mass point of the body.
`im_origin`	An optional argument for the origin point of the inertia marker of the body.
`lprf_origin`	An optional argument for the origin point of the local part reference frame of the body.
`body_num`	An optional integer argument assigned to the body as its entity number.

Square brackets, `[ ]`, in the description of any statement syntax means that an argument is optional.

This problem requires `body_name`, `body_label`, and `cm_origin`.

```
//Bodies
*Body(b_link, "Ball", p_pendu_cm)
```

4. Define the graphics for the body for visualization. To attach graphics to the body, use the `*Graphic()` statement for spheres and cylinder to display the link and the sphere.

Statement syntax for sphere graphics:

```
*Graphic(gr_name, "gr_label", SPHERE, body, origin, radius)
```

where:

`gr_name`	The variable name of the graphic.
`gr_label`	The descriptive label of the graphic.
`SPHERE`	This argument indicates that the graphic is a sphere.
`body`	The body associated with the graphic.
`origin`	The location of center point of the sphere.
`radius`	The radius of the sphere.

For this exercise, use all of the arguments. The statement is:

```
//Graphics for sphere
```

```
*Graphic(gr_sphere, "pendulum sphere graphic", SPHERE, b_link,
p_pendu_cm, 1)
```

Statement syntax for cylinder graphics:

```
*Graphic(gr_name, "gr_label", CYLINDER, body, point_1, POINT|VECTOR,
orient_entity, radius, [CAPBOTH|CAPBEGIN|CAPEND])
```

where

`gr_name`	The variable name of the graphic.
`gr_label`	The descriptive label of the graphic.
`CYLINDER`	This argument indicates that the graphic is a cylinder.
`body`	The body associated with the graphic.
`Point1`	The location of one end of the cylinder.
`POINT|VECTOR`	Keyword to indicate the type of entity used to orient the cylinder. If POINT is used, the following argument should resolve to a point, otherwise it should resolve to a vector.
`orient_entity`	The variable name of the entity for orienting the cylinder.
`radius`	The radius of the cylinder.
`[CAPBOTH| CAPBEGIN| CAPEND]`	An optional argument that identifies if either or both cylinder ends should be capped (closed).

For this exercise, use all of the arguments. The statement is:

```
//Graphics for cylinder
```

```
*Graphic(gr_link, "pendulum link graphic", CYLINDER, b_link
p_pendu_pivot, POINT, p_pendu_cm, 0.5, CAPBOTH )
```

5. Create a revolute joint at the pivot point. The syntax is:

```
*RevJoint(joint_name, "joint_label", body_1,body_2, origin,
POINT|VECTOR, point|vector, [ALLOW_COMPLIANCE])
```

where:

`joint_name`	The variable name of the joint.
`joint label`	The descriptive label of the revolute joint.
`body 1`	The first body constrained by the revolute joint.
`body 2`	The second body constrained by the revolute joint.

`origin`	The locations of revolute joint.
`POINT|VECTOR`	Keyword to suggest the method of orientation for the joint using a point or vector.
`point|vector`	A point or vector that defines the rotational axis of the revolute joint.
`[ALLOW COMPLIANCE]`	An optional argument that indicates the joint can be made compliant (a joint that is compliant is treated like a bushing and can be toggled between compliant and non-compliant).

For this problem, you will use the following statement:

```
//Revolute Joint

*RevJoint(j_joint, "New Joint", B_Ground, b_link, p_pendu_pivot, VECTOR, V_Global_X)
```

6. Create an entity output statement. The syntax for `*Output - output on entities` is:

```
*Output(out_name, "out_label", DISP|VEL|ACCL|FORCE, entity_type, ent_name, [ref_marker], [I_MARKER|J_MARKER|BOTH_MARKERS])
```

where:

`out_name`	The variable name of the output.
`out_label`	The descriptive label of the output.
`DISP|VEL|ACCL|FORCE`	An argument that indicates whether the output type is displacement, velocity, acceleration, or force.
`entity_type`	Keyword to indicate the type of entity on which the output is being requested. Valid values are: BODY|JOINT|BEAM|BUSHING|FORCE|SPRINGDAMPER
`ent_name`	The entity on which output is requested.
`ref_marker`	An optional argument for the reference marker in which the output is requested.
`I_MARKER|J_MARKER|BOTH_MARKERS`	Keyword to indicate the capture of output on the I marker, J Marker or both markers. The default is both markers.

In order to obtain the displacement versus time output of the falling ball, you will use the `*Output()` statement as follows.

```
//Output

*Output(o_pendu, "Disp Output", DISP, BODY, b_link)
```

7. Set property values for the entities you created in your MDL model file. This is done in the property data section of the MDL model file. For this problem, use the `*SetSystem()`, `*SetPoint()`, and `*SetBody()` statements.

```
//Property data section

*SetPoint(p_pendu_pivot, 0, 5, 5)

*SetPoint(p_pendu_cm, 0, 10, 10)

*SetBody(b_link, 1, 1000, 1000, 1000, 0, 0, 0)
```

8. Save the model as `pendulum.mdl`.

   Your MDL model file will look like the file below (it summarizes the key sections of the MDL model file for this exercise):

```
//Pendulum Model

//05/31/XX

*BeginMDL(pendulum, "Pendulum Model")


//Topology information

//declaration of entities


//Points

*Point(p_pendu_pivot, "Pivot Point")


*Point( p_pendu_cm, "Pendulum CM")


//Bodies

*Body(b_link, "Ball", p_pendu_cm)


//Graphics

*Graphic(gr_sphere, "pendulum sphere graphic", SPHERE, b_link,
p_pendu_cm, 1)

*Graphic(gr_link, "pendulum link graphic", CYLINDER, b_link,
p_pendu_pivot, p_pendu_cm, 0.5, CAPBOTH)


//Revolute Joint

*RevJoint(j_joint, "New Joint", B_Ground, b_link, p_pendu_pivot, VECTOR,
V_Global_X)


//Output

*Output(o_pendu, "Disp Output", DISP, BODY, b_link)

//End Topology

// Property Information

*SetPoint(p_pendu_pivot, 0, 5, 5)
```

```
*SetPoint(p_pendu_cm, 0, 10, 10)
*SetBody( b_link, 1, 1000, 1000, 1000, 0, 0, 0)
*EndMDL()
```

Step 3: Load and run the MDL model file.

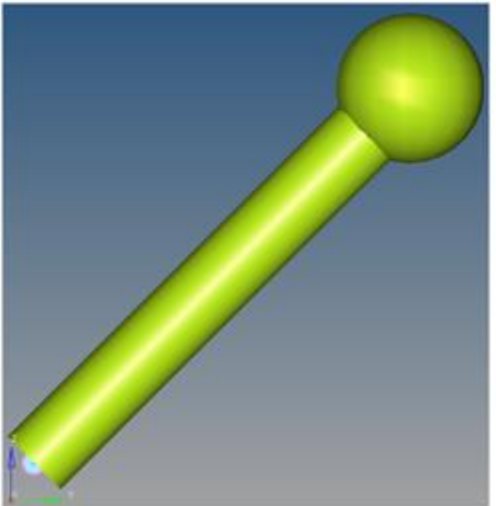
1. Launch ***MotionView***.
2. Click the ***Open Model*** icon, , on the **Standard** toolbar.

   OR

   – From the **File** menu, select ***Open > Model***.

4. From the **Open Model** dialog, locate and select the file `pendulum.mdl`.
5. Click ***Open***.
6. Observe and review the model in the graphics area.
7. From the **Standard View** toolbar, click the ***YZ Rear Plane View*** button.

   The model is seen as shown in the image below:



8. Use the **Project Browser** to view the model entities and verify their properties.
9. Go to the ***Tools*** menu and click on ***Check Model*** to check for any modeling errors.

10. Perform the following steps to run MotionSolve:
    - Go to the ***Run*** panel and select ***Transient*** as the **Simulation Type:** option.
    - Set the **End time** as `2` seconds.
    - Click on the ***Simulation Settings*** button.

    The **Simulation Settings** dialog is displayed.
    - Click on the ***Transient*** tab and review the integrator parameters.
    - Click ***Close*** to close the dialog.
    - From the ***Main*** tab, use the **Save and run current model** file browser, and enter `pendulum` for the xml.
    - Click ***Run***.
    - Upon completion of the run, close the solver window and clear the message log.

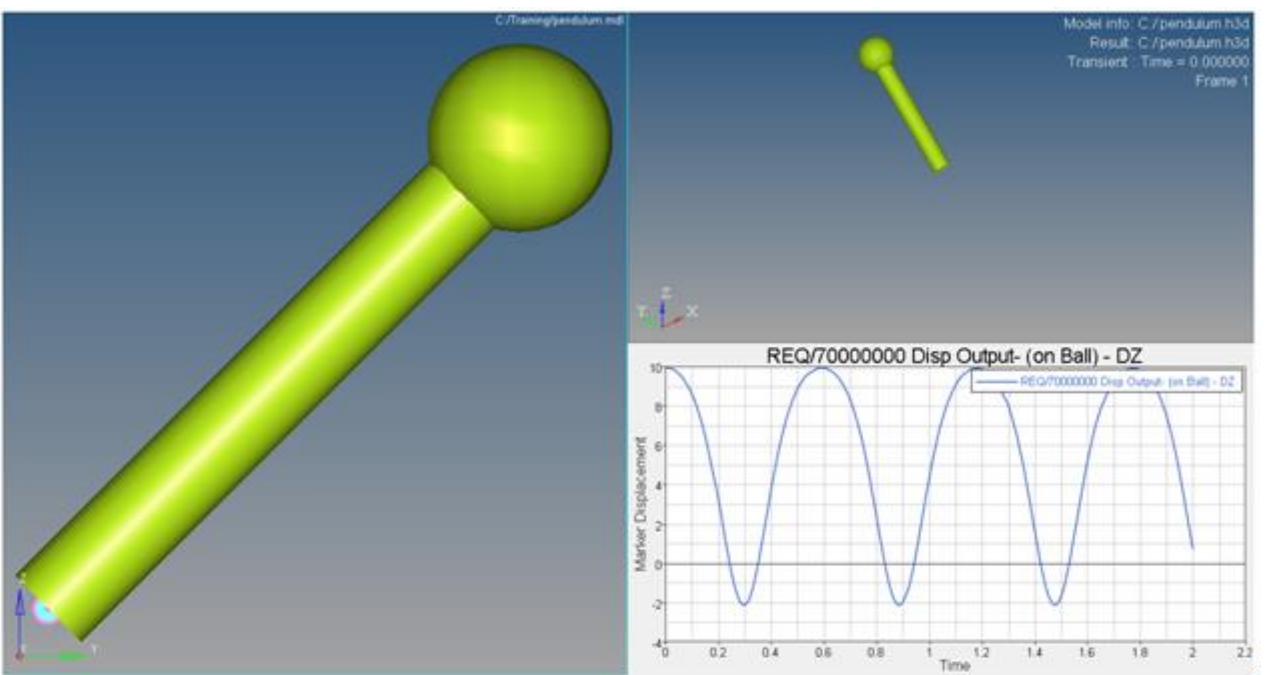
Step 4: Animate and plot the results.

1. Click ***Animate***.
2. The animation file `pendulum.h3d` will be loaded in the adjacent window. Click on that window to activate it.
3. Click the ***Start/Pause Animation*** icon on the **Animation** toolbar to start the animation, and click the ***Start/Pause Animation*** icon again to pause the animation.
4. Right-click on the ***Fit Model/Fit All Frames*** icon icon on the **Standard Views** toolbar to fit the visualization in all frames of the animation.
5. Click on the MotionView window to make it active.
6. From the **Run** panel, click ***Plot***.
7. The plot window will be added, with the `pendulum.abf` loaded.
8. Select **Y Type** as `Marker Displacement`, **Y Request** as `REQ/70000000 Disp Output – (on Ball)`, and **Y component** as `DZ`.
9. Click ***Apply***.

   The plot for the displacement of the pendulum in the Z direction is shown.

10. Click the ***Start/Pause Animation*** icon, ▶, to review the plot and animation together. Click ⏸, to pause the animation.

    Your session page should look similar to the image below:



11. Close the session using the **File** menu (***File > Exit***).