**LbcbPlugin Data Organization**

1. Introduction. This document is a guide to the organization of the LbcbPlugin step data. The guide shows how this data can be accessed by the customizable plugins which are written for specific tests. The step data is organized with MATLAB classes.
2. Anatomy of a Plugin Function. The LbcbPlugin provides a customization feature called plugin functions that allow the researcher to specify the control protocol of a test. The application supports two types of plugin functions; control point transformation functions and step correction functions. Examples of these functions can be found in the UserFunctions folder.
   1. Control Point Transformation Functions. These functions are used to convert step data between LBCB coordinate space and UI-SimCor model coordinate space. The functions have two input arguments and one output argument. The two input arguments are the ‘me’ parameter and the input displacement and force data expressed in the input coordinate space. The output argument is the same data expressed in the output coordinate space.
   2. Step Correction Functions. These functions are used during the execution of a step or sub-step as indicated in Figure 1 Step Correction Flowchart . The functions accept the ‘me’ reference and a reference to the step data (4) as inputs. The functions are executed at key points of the flow as shown in Table 1 Derived DOF Correction Functions. The step parameter passed into these functions has contents that are defined by the execution state of the flow diagram. The step parameter for both adjustment functions contains the next command to be sent to the LBCBs. The step parameter for the calculate function contains the just executed command plus the response from the LBCBs:

Table 1 Derived DOF Correction Functions

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Description | Parameters | Executed At |
| Preliminary Adjustment | Adjust the initial command for a step before sending it to the OM | ‘me’ and ’step’ as input | Prelim DD Adjustment |
| Response Calculations | This is where all of the calculations are done on the response data for a step. The data from the response will be used for decisions concerning the next step | ‘me’ and ’step’ as input | DD (Level) Calculations |
| Needs Correction | This returns true if a correction step needs to be generated | ‘me’ as input  A true/false flag as output. | Needs DD (Level) Correction? |
| Adjust Target | Adjusts the command in a correction step | ‘me’ and ’step’ as input | Adjust DD (Level) |

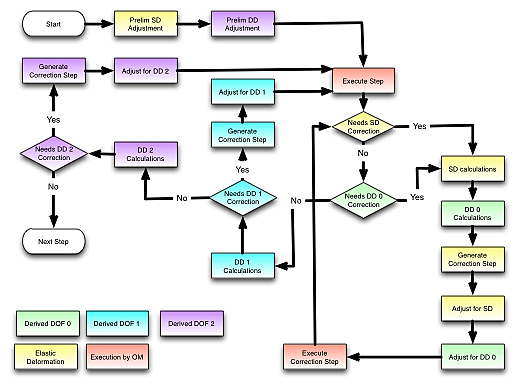


Figure 1 Step Correction Flowchart

* 1. The ‘me’ Parameter. The ‘me’ parameter is a reference to the DerivedDof object found in the States/Step folder. The ‘me’ parameter can be used to access the variables of the application. These variables are used to pass data between the functions. For example, a calculation function could set some calculations based on the response of a step that the ‘needs correction’ function can access to determine if another correction step is necessary. Only the floating point data type is supported by these variables. There are three types of variables; Configured, Archived, and Data. Configured variables can be set using the “Edit->Config Variables” menu while the application is running. This allows the behavior of the function to be modified during a test. The Archived variables have their values for each step archived in the CorrectionData archive. The data from these variables can also be used in plots. The Data variables are never used.
  2. Accessing Variables. The application variables are managed by ‘get’, ‘put’, and ‘exists’ methods of the ‘me’ parameter. Each type of variables has its own set of methods. Configured variables are accessed by the functions getCfg (‘Name’), putCfg (‘Name’, ’Value’), and existsCfg (‘Name’). The archived variables are accessed using a set of functions with ‘Cfg’ replaced by ‘Arch’.
  3. DerivedDof Class Properties. The DerivedDof class has the following properties:

Table 2 DerivedDof Properties

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Configuration Access | Handle to a ConfigDaoProvider (4.9) object which allows access to the application’s configuration data | cdp |
| Derived DOF Control Level | The current level of Derived DOF correction | level |
| Target History | Handle to a CellFifo (3.10) object that allows access to past targets. The size of the buffer is set to be 20 in the SimSharedData constructor | targetHist |
| Substep History | Handle to a CellFifo (3.10) object that allows access to past substeps | substepHist |
| Execute Step History | Handle to a CellFifo (3.10) object that allows access to past-executed steps. **Note:**  The next step is not added to this buffer until it is sent to the OM | executeHist |

1. Step Data Class Hierarchy. All of these classes are contained in the SimData folder.
   1. StepData Class. The step data contains a complete set of everything that occurred during the step. The following data is included:

Table 3 StepData Properties

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| LBCB Data | Cell array if one or two LbcbControlPoint (4.2) objects which contain commands and responses from the OM. Also contains the setup deformation calculations | lbcbCps |
| Model Data | Cell array of ModelControlPoint (3.6) objects which contain commands and response in the UI-SimCor coordinate space | modelCps |
| Step Number | Handle to the StepNumber (3.7) object contain all of the numbers for the step. | stepNum |
| Calculated Data | Names and values of all of the archive variables set by the plugins stored in a CorrectionData (3.8) object | cData |
| Configuration Data | Handle to a ConfigDaoProvider which contains the current configuration of the application as well as a set of convenience functions | cdp |
| Flags | Miscellaneous flags which characterize the step | containsModelCps |

* 1. LbcbControlPoint Class. This class contains the response and command for one LBCB as well as data for the control sensor associated with the LBCB.

Table 4 LbcbControlPoint Properties

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Response Data | Handle to an LbcbReading (3.3) object | response |
| Command Data | Handle to a Target (3.5) object | command |
| Control Sensor Data | Numeric array of control sensor readings | externalSensors |

* 1. LbcbReading Class. This class contains the responses of an LBCB. It has two virtual properties which return the appropriate response regardless of whether the application is configured to calculate setup deformation or not.

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Displacements | Numeric array containing the position of the LBCB in LBCB Cartesian space. This is a dependent property that returns either lbcb or ed data depending on whether ed calculate is set. | disp |
| Forces | Numeric array containing the force readings of the LBCB in LBCB Cartesian space. This is also a dependent property. | force |
| LBCB Data | Handle of DofData (3.4) containing the LVDT & load cell readings of the LBCB in LBCB Cartesian space | lbcb |
| Setup Deformation Data | Handle of DofData (3.4) containing the position of the LBCB in LBCB Cartesian space calculated from the control sensors as well as the Cartesian forces | ed |

* 1. DofData Class. This class contains the 12 degrees of freedom in Cartesian space of displacements and forces.

Table 5 DofData Properties

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Displacements | Numeric array of displacements; 3 translations, 3 rotations. | disp |
| Forces | Numeric array of 3 forces and 3 moments. | force |
| Labels | Cell array of DOF labels | labels |

* 1. Target Class. Class containing the displacement and force commands.

Table 6 Target Properties

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Displacements | Numeric array of displacements; 3 translations, 3 rotations. | disp |
| Forces | Numeric array of 3 forces and 3 moments. | force |
| Control DOFs for Displacements | Binary array indicating which displacement DOFs are being commanded | dispDofs |
| Control DOFs for Forces | Binary array indicating which force DOFs are being commanded | forceDofs |
| Labels | Cell array of DOF labels | labels |

* 1. ModelControlPoint Class. Contains displacement and force commands and responses in the coordinate space of UI-SimCor.

Table 7 Model Control Point

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Response Data | Handle to an Target (3.5) object | response |
| Command Data | Handle to a Target (3.5) object | command |
| UI-SimCor Address | The model control point address referenced by UI-SimCor | address |

* 1. StepNumber Class. Class containing the Step, sub-step, and correction step number of the step.

Table 8 StepNumber Properties

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Step number | Step number of the step which comes from either the input file or UI-SimCor | step |
| Sub-Step Number | Number of the sub-step. This is created when a step is split up by DOF increments | subStep |
| Correction Step Number | Number created by the correction plugins. Setup deformations increase this number by single digits. Derived DOF corrections increase this number by increments of 100 times what every control level the correction happens in. | CorrectionStep |
| Flags | Miscellaneous flags which characterize the step | isFirstStep, isInitialPosition, isLastSubStep |

* 1. CorrectionData Class. Class containing the labels and values of all of the archive variables.

|  |  |  |
| --- | --- | --- |
| Name | Description | Property Labels |
| Variable Names | Cell array of variable names | labels |
| Variable Values | Numeric array of variable values | values |

* 1. ConfigDaoProvider Class. Class containing the application configuration data (in a property called ‘cfg’). It is found in the Config folder. It also contains a number of convenience methods:

Table 9 ConfigDaoProvider Methods

|  |  |
| --- | --- |
| Method Name | Description |
| getAddress() | Get the primary UI-SimCor address of the test |
| numLbcbs() | Returns the number of LBCBs configured |
| numModelCps() | Returns the number of control points passed between UI-SimCor and the application |
| getAddresses() | Returns an array of control point addresses. |
| getExtSensors() | Returns arrays of names, sensitivities, and LBCB labels for all of the control sensors. The LBCB labels array can be one of ‘LBCB1’, LBCB2’, or ‘BOTH’ |
| getNumExtSensors() | Returns the number of control sensors |
| getFilteredExtSensors(isLbcb1) | Returns the same arrays as getExtSensors() for a particular LBCB |
| useEd() | Returns true if setup deformation is being corrected. |
| useDd(level) | Returns true if the specified level of Derived DOF is being corrected. |
| doStepSplitting() | Returns true if steps are being subdivided into sub-steps |
| forceAccept() | Returns true if only sub-steps and correction steps are auto-accepted. |
| getSubstepInc(isLbcb1) | Returns a numeric array of the sub-step increment for each displacement DOF. |

* 1. CellFifo Class. Class used to manage target and command histories. It is found in the SimData folder. It has several methods:

Table 10 CellFifo Methods

|  |  |
| --- | --- |
| Method Name | Description |
| push() | Add a stepdata object. The oldest object is removed if the FIFO is at max length |
| pop() | Remove the newest stepdata object. The oldest FIFO slot becomes empty |
| get(index) | Get the stepdata object at the index. Index 1 is the newest. The object at the max index is the oldest. |
| toString() | Prints out the step number for each object in the fifo |

1. Examples.
   1. Accessing the LBCB Response.
      1. Accessing Dx, and Ry for LBCB 1. The first example accesses the displacement array for the LBCB 1 response directly. The second example uses the StepData (3.1) method respData() to get the displacement and forces for all of the LBCBs in numeric arrays.

[dx ry] = step.lbcbCps{1}.response.disp(1 4);

[ disp force] = step.lbcbCps{1}.respData();

[dx ry] = disp([1 4]);

* + 1. Getting Fz from LBCB 2. Two more examples using the two methods described in 4.1.1. Notice the cell array index is 2.

fz = step.lbcbCps{2}.response.force(3);

[ disp force] = step.lbcbCps{2}.respData();

fz = force(3);

* 1. Accessing the LBCB Command.
     1. Accessing Dx and Ry commands for LBCB 1. The first example accesses the displacement array for the LBCB 1 command directly. The second example uses the StepData (3.1) method cmdData () to get the displacement and forces for all of the LBCBs in numeric arrays. This function returns not only the displacement and force values but also two binary vectors whose elements are set to true if the DOF is of the command. For example, for a command that sets Dx, Fz, and My, the displacement control vector will be [ 1 0 0 0 0 0 ] and the force control vector will be [ 0 0 1 0 1 0].

[dx ry] = step.lbcbCps{1}.command.disp(1 4);

[disp dDofs force fDofs] = step.lbcbCps{1}.cmdData();

[dx ry] = disp([1 4]);

* + 1. Setting Dx, Fz, and My commands for LBCB 1. Use the setDispDof() and setForceDof() methods so that the control vectors are set correctly.

step.lbcbCps{1}.command.setDispDof(1,dx);

step.lbcbCps{1}.command.setForceDof([3 5], fz, my);

* 1. Using Variables:
     1. Here is an example of how the setup deformation code accesses the configuration variables for the correction factor. The two methods used are existCfg() which returns whether the configuration variable is set or not and getCfg() which returns the value.

if me.existsCfg('EdCorrectionFactor')

cf = me.getCfg('EdCorrectionFactor');

else

cf = 1;

end

* + 1. Here is an example of a calculate function using the response from a step to calculate various force values. Notice that the code uses configuration variables (‘MomentShearRatio’ and ‘MyCorrectionFactor’) to guide the calculation. The resulting archived variables can be used for the other step correction functions as well as for plotting and archiving.

msr = me.getCfg('MomentShearRatio');

fx = step.lbcbCps{1}.response.force(1);

mz = step.lbcbCps{1}.response.force(6);

mzshear = fx \* 25.7 + mz;

mmy = step.lbcbCps{1}.response.force(5);

mm2s = mmy / fx;

myBot = mmy - 144 \* fx;

fz = step.lbcbCps{1}.response.force(3);

mx = step.lbcbCps{1}.response.force(4);

x\_eccentricity = mx / fz;

shearcenter = mz/ fx;

cmy = (step.lbcbCps{1}.response.force(1))\* msr;

correction = cmy - mmy;

if me.existsCfg('MyCorrectionFactor')

cf = me.getCfg('MyCorrectionFactor');

else

cf = 1;

end

pmy = mmy + correction \* cf;

me.putArch('ProposedMy',pmy);

me.putArch('MeasuredMy',mmy);

me.putArch('MyBot',myBot);

me.putArch('MeasuredMoment2Shear',mm2s);

me.putArch('MzAtShearCenter',mzshear);

me.putArch('ShearCenter',shearcenter);

me.putArch('XEccentricity',x\_eccentricity);

* + 1. Here is an example of an adjust target function using archive and configuration variables to set the corrected commands:

my = me.getArch('ProposedMy');

step.lbcbCps{1}.command.setForceDof(5,my);

sfz = true;

if me.existsCfg('setAxialForce')

spfz = me.getCfg('setAxialForce');

sfz = spfz == 1;

end

if sfz

fz = me.getCfg('AxialForce');

step.lbcbCps{1}.command.setForceDof(3,fz);

else

me.log.debug(dbstack,'Not setting Axial Force in DD0 command');

end

* 1. Using ‘log’ For Echo Printing. All of the classes defined in the LbcbPlugin code have a Logger class property called ‘log’. The Logger allows you to structure the print statements you need to include in your code so that you can filter them and also see where they come from. The Logger class has 4 different methods; debug, info, warning, and error. All of these methods have the same parameters; a call to dbstack, and the string to be displayed. The functions have a filtering priority with debug being the lowest and error being the highest. You choose which function to use by how important the information is that you want to print.
     1. Assembling A debug Message. The following code prints out the results of a preliminary adjustment in one long string:

str = sprintf('Prev Position: %s\n',curStep.lbcbCps{lbcb}.response.toString());

str = sprintf('%sPrev Cmd: %s\n',str,curStep.lbcbCps{lbcb}.command.toString());

str = sprintf('%sNew Target: %s\n',str,nextStep.lbcbCps{lbcb}.command.toString());

nextStep.lbcbCps{lbcb}.command.disp = newOMcommand;

str = sprintf('%sCorrectionFactor: %f\n',str,cf);

str = sprintf('%sNew Cmd: %s',str,nextStep.lbcbCps{lbcb}.command.toString());

me.log.debug(dbstack,str);

* + 1. Printing Out A Step. All of the classes used to store the step data come with a toString() method that saves on the having to assemble messages such as the previous example. This method prints out the all of the properties of the class. Here is an example on the method being used to print out the just processed response:

me.log.debug(dbstack,sprintf('Current Response: %s', me.dat.curStepData.toString()));

* + 1. Printing Out Step History. In the DerivedDof class you have access to the target, substep, and executed step history. Here are some example debug statements accessing that history from DerivedDof functions.

me.log.debug(dbstack, sprintf('Target History: \n%s',me.targetHist.toString()));

me.log.debug(dbstack, sprintf('Substep History: \n%s',me.substepHist.toString()));

me.log.debug(dbstack, sprintf('Executed Step History: \n%s',me.executeHist.toString()));

me.log.debug(dbstack,sprintf('2nd Prev = %s',me.executeHist.get(2).toString()));