Driving a Time-based Simulation from MATLAB for Custom Calculations and Control

Wes Sunderman
Roger Dugan
June 2010

The OpenDSS allows other programs to 'drive' the simulations and perform custom calculations and control via the program's COM interface. This documentation is provided to give the reader with a basic understanding for how to drive the OpenDSS using the familiar Matlab program.

Note that the OpenDSS program is provided in two forms:

1. OpenDSS.EXE: A stand-alone executable that does not have a COM interface. This is for running the program through text scripts, and for developing and testing scripts.
2. OpenDSSEngine.DLL: An in-process COM server designed to provide easy access to program features via the COM interface. This module must be *registered* with the Windows operating system.

These two forms are functionally equivalent for distribution system analysis. The DLL version lacks some of the features available through the menus of the EXE version.

More documentation on the COM interface is available in the 'doc' folder that is provided on the Sourceforge.Net website. A direct link to the folder is:

http://electricdss.svn.sourceforge.net/viewvc/electricdss/Doc/

Another document at this site relevant to writing custom control algorithms and interfacing with the OpenDSS through the CtrlQueue interface is

'OpenDSS CtrlQueue Interface.doc'.

This document will outline the steps for running a time-based simulation that performs custom calculations for running a control algorithm. The examples here are for MATLAB. They can be implemented in other programming languages that support connecting to COM interfaces on the Windows platform. Similar algorithms have been developed using VBA in Microsoft Excel and Access and such languages as Python, C#, and Delphi.

Step 1: Get a connection to the OpenDSS COM interface and start the OpenDSS

First, define some variables and call the DSSStartup function with a directory path name. The DSSStartup function will make this directory the home directory and then start up the OpenDSS.

```
    %Start up the DSS
    global DSSStartOK;
    global DSSObj;
    global DSSText;

    [DSSStartOK, DSSObj, DSSText] = DSSStartup(myDir);
```

The Matlab function DSSStartUp (below) starts the OpenDSS and provides a handle to the DSS via the Obj variable, which subsequently assigned to the DSSObj variable when the function returns. The function definition for the DSSStartup is shown here:

```
function [Start,Obj,Text] = DSSStartup(mydir)
    % Function for starting up the DSS
    % make sure we are in the proper directory
    cd(mydir);
    %
    %instantiate the DSS Object
    Obj = actxserver('OpenDSSengine.DSS');
    %
    %Start the DSS.  Without this statement, DSS will not run (gives
    %max_circuit=0 error).  Only needs to be executed the first time
w/in a
    %Matlab session
    % Define the text interface
    Start = Obj.Start(0);
    Text = Obj.Text;
end
```

Note that we're instantiating the OpenDSSengine.DSS COM server via the Matlab actxvserver function, and then starting the OpenDSS by passing in the value '0' to the Start method. Then we assign the local variable Text, which is returned in the function argument list to the DSSText variable, to the Text Interface of the OpenDSS for passing in text commands.

Step 2: Setting up the Time-based Simulation

Now that the OpenDSS COM server is running and ready for some circuit definitions or other initial steps, we need to do some 'setup' work before getting into the simulation loop itself.

We have an OpenDSS script that includes all of the objects/classes that we need to define for the circuit on which we're interested in performing simulations. This is usually a file named '**master.dss'** that contains such things as a source definition, transformer

definitions, line definitions, capacitor definitions, loadshape definitions and load definitions.

An approach that we took in setting up the **master.dss** file was to not include any solve statements as we wanted to let the MATLAB code control any solutions as needed. This is not absolutely necessary, but it did allow us more flexibility. The **master.dss** file does contain a **calcvoltagebases** statement to set the base voltages. This causes a solution without any Load object or other shunt elements.

To the code:

```
% compile the base circuit which does NOT have a solve in it
DSSCircuit = DSSObj.ActiveCircuit;
result = dss_Command('compile master.dss');
```

First we set the DSSCircuit variable to the ActiveCircuit interface of the DSSObj. At this point, ActiveCircuit is empty because there is no circuit defined. The DSSCircuit variable gives us a handle into the Circuit interface of the OpenDSS COM server for convenience.

Next, the text statement to compile the master.dss file is passed into the DSSText.Command interface using the dss_Command function written in the Matlab code. Notice that the result code from the command is retrieved. Generally speaking, if the result code from dss_Command is anything except 1, we want to exit the MATLAB script. We use the following general code to provide the text of the result to the user and then exit the script:

```
  % if an error returned from the DSS - return text from the DSS %
  %and exit this function with a failure return value
if(result~= 1)
    disp(['Error:' result])
    k = 0;
    return
end
```

Some commands send a sizeable string back from the DSSText.Result interface, even when the text command is successfully parsed and executed. For example, you can retrieve voltages and currents, etc. through this interface if they are the result of the command. Check the Error interface to distinguish whether an actual error occurred.

We perform a “snapshot” solve and then assign a variable that will allow us easy access to all of the EnergyMeter objects in the circuit:

```
%solve the circuit in snap shot mode
result = dss_Command('solve mode=snap');
DSSEnergyMeters = DSSCircuit.Meters;
```

For a yearly simulation, we set the mode to ‘yearly’ via the text interface, and then, in this example we set the number of solutions per invocation of the Solve command and the interval (or step-size) between solutions using directly using numerical values in the Solution interface. We are telling the OpenDSS that we want only 1 solution for each time that it performs a Solve so that we can interact with the solution before advancing to the next time step. Note that the stepsize is specified in seconds when using the Solution interface directly.:

```
result = dss_Command('set mode = yearly');
DSSCircuit.Solution.Number = 1;
DSSCircuit.Solution.Stepsize = 3600;
```

If we were using the Text interface to specify the stepsize, we could have defined it in minutes or hours as well using text strings. The following are equivalent;

```
result = dss_Command('set stepsize=3600');
result = dss_Command('set stepsize=3600s');
result = dss_Command('set stepsize=60m');
result = dss_Command('set stepsize=1.0h');
```

For our purposes, we want to get a variety of reports for each time step from the energymeters in the circuit. The following code sets this up:

```
%set the case name
result = dss_Command('set casename=pv_var_testing9');

%set trace control to true
result = dss_Command('set tracecontrol = yes');

%we want an overload report
result = dss_Command('set overloadreport=true');

%we want a voltage exception report
result = dss_Command('set voltexceptionreport=true');

%we want verbose demand interval data
result = dss_Command('set DIVerbose=true');

%demand is true
result = dss_Command('set demand=true');
```

The last step before getting into the external solution and control loop, is to set the hour for the first simulation to zero. We can do that by setting the ‘dblHour’ property in the Solutions interface through the following statement:

```
DSSCircuit.Solution.dblHour = 0.0;
```

The dblHour property is expecting a double-precision floating-point number.

Step 3: Time-based Simulation Loop

For the time-based simulation that we're performing we first want to get a solution of the circuit. Since the simulation is in **Yearly** mode, the loads will follow the yearly **Loadshape** objects that were defined when the **master.dss** file was loaded along with definitions of the **Load** objects and all other circuit elements.

In our particular loop, we want to run for a certain number of steps. At each step we want to solve the circuit, perform some calculations that are specific to our problem, update some loads or other objects in the circuit definition based on a control algorithm, and repeat the process. The Matlab code for this is shown below:

```
while(present_step <= num_pts)
    DSSCircuit.Solution.Solve;

    t = sample_dss_values(present_step);
    t = do_custom_control_loop();
    if (t == 0)
        disp('Error running the pv control loop')
        k = 0;
        return
    end

   %increment the present step
    present_step = present_step + 1;
end
result = dss_Command('CloseDI');
```

In more detail, at the top of the loop we first perform a Solve. Recall that we set the mode to 'Yearly', so the loads will follow their loadshapes. Had we been in Snapshot mode the loads would not follow their respective loadshapes – they would have been at the defined kW output level at each solve.

Next, we call a function to sample some values from the OpenDSS solution that are pertinent to our control loop. Based on those values we go into our custom control loop that, in this example, updates reactive power output of some of the loads.

The custom control loop function does some solutions of its own to iterate on the reactive power output, to set it to a correct value based on the conditions of the circuit. The solutions that the custom control loop performs are "NoControl" solutions. These perform solutions of the circuit, but do not allow any the other OpenDSS control objects to change the state of their controlled object

We increment the internal (i.e., MATLAB) step counter variable, present_step. Since we are in Yearly mode, the OpenDSS solution time variable is automatically incremented and the energy meters and monitors sampled each time Solve is executed. Since demand interval metering was turned on, the demand interval files have to be closed by the CloseDI command.