Best of the Discussion Forum – Part II

On MaxControlIterations …

From Roger Dugan:

I see many scripts come by my inbox where users have set MaxIterations to 100 or 1000 thinking that will help with convergence problems. Likewise, I see MaxControlIterations set to very high numbers. This is seldom helpful.

The default MaxIterations is 15 and the default MaxControlIterations is 10. This is nearly always sufficient for reasonably-modeled distribution circuits.

The fixed point iteration solution technique essentially uses the Y matrix as the Jacobian. We don't update the Y matrix between iterations, or time steps, unless there has been a major change such as a regulator tap change or a capacitor switching. As long as the Y matrix is generally pointing in the right direction for the solution at which all currents at a node will sum to zero, the iterative process will usually get there eventually. It will take more iterations than a Newton method that updates Y at every iteration, but it will generally not take hundreds of iterations. The worst I have seen is 81 and that was for a circuit modeling error. The IEEE 8500-node Test Case takes 16 iterations on the first control iteration, but fewer on subsequent control iterations (there are 5 total control iterations -- see **Show Summary** after the solution). So in the script, we set MaxIterations=20.

It mostly has to do with how close the first guess at the voltages is to the final solution. The fixed-point iteration requires the first guess to be closer than other methods. The worst case is generally the first snapshot power flow. Thereafter, for a sequential-time power flow solution, the result for one time step is a really good starting point for the next time step. Thus, most solutions in a yearly or daily simulation take only 2 iterations -- one to get there and one to verify convergence.

So the method is not as good as other methods in terms of number of iterations for the initial solution, but it is not terrible either. So you should never have to set MaxIterations to a high value. In fact, if you are getting convergence only after 100 iterations, there is probably something wrong with your model such as a Load or Generator defined with the wrong voltage rating. Note that you can easily fix that if you've got the voltage base defined for each bus by using the **SetLoadandGenkV** command. This corrects the kV defined for Loads and Generators to match the kV base of the bus to which they are connected. Note that this mismatch could also happen for an incorrect definition of a Transformer object or Vsource object.

If OpenDSS reports that your solution did not converge, first try executing 'Solve' again. As with most things in OpenDSS, the program remains in the last state it was in when it stopped doing something. Hitting the Solve button again will continue the iterations from where it left off. If it still doesn't converge after two more Solves, there is likely something happening that will prevent convergence no matter how many iterations.

Users were recently having difficulties with convergence after opening Line objects. Part of this was due to the Load models not working well at near-zero voltages and part was due to tiny perturbations in the zero sequence, or ground, path. The model frequently loses connection to a strong ground source when a line is open, making it more difficult to figure out where the neutral should be. Build 7.6.3.26 corrected most of this but I wouldn't guarantee it for all cases. But doing more iterations will generally not help.

On **MaxControlIterations**, the program first achieves a converged power flow solution then queries all the control devices to see if they need to adjust something. If so, the adjustment is made and a new power flow solution is executed. This iteration process continues until all control actions for the present time are done.

The usual reason for exceeding MaxControlIterations is conflicting controls. One common one is where all the capacitor control time delays are set to the same value. Thus, they will all switch ON at the same time. That results in too much var support, so in the next control iteration they all switch OFF. This will continue forever if you let it. The value you set for MaxControlIterations will have no effect other than the stop the process when MaxControlIterations is reached.

Note that when you get the message that MaxControlIterations were exceeded, the power flow solution is likely OK. You can use it if you want to. It is often a reasonable solution.

The cure for the control oscillations is to **stagger the time delays**. For example, if all the CapControl objects have a delay of 60 s, set the one closer to the substation to 61 s. This will force the other one(s) to act first and will often fix the problem. In **STATIC control mode** (the default for Snapshot, Daily, and Yearly modes), OpenDSS will execute only the controls with the shortest time delay on each control iteration. In **TIME control mode** with a time step shorter than the delay time of the control, such as you would do in dutycycle mode for solar PV generation studies, this is usually not a problem because the control action waits until the solution time reaches the delay time. It is less likely that two capacitors in different parts of the circuit will arm to switch on exactly the same time step. When it comes time to switch, if the additional capacitor is no longer needed, the control action is popped off the control queue and discarded.

So the controls will generally sort themselves out when simulating in small time steps. The problem is generally when simulating in large time steps larger than the delay time in the control or in the standard Snapshot mode. In Snapshot mode, the control simulation strategy is to sort out the control actions in the order they would actually occur in real life. This is different than most other distribution system analysis tools, so it is common for OpenDSS to predict different regulator taps and capacitor switch positions. Of course, we think our method is better ;-)

There are legitimate reasons for increasing MaxControlIterations. You will see this in some of the IEEE Test Feeder scripts.

One reason is when you limit the max tap change of regulators to one tap per control iteration. This frequently gives the best simulation of an actual voltage regulator, but requires more control iterations. An example of this can be found here:

https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Distrib/IEEETestCases/123Bus/Run_IEEE123Bus.DSS

Another case would be if you are running InvControl on several PVSystems. Some of the control modes sneak up on the final answer slowly and require many passes through the control iteration loop.

From Wes Sunderman:

If you have the InvControl controlling several PVSystems, it is generally 'slow' in arriving at the final answer. One reason it was programmed this way, is to try to keep PVSystems that are electrically close to one another from interacting with each other, for instance in volt-var mode.

So we purposefully slow down the maximum reactive power change per control iteration to keep the PVSystems from interfering with each other and causing 'hunting' or oscillatory behavior.

The user-settable parameter *deltaQ*_factor (and the corresponding *deltaP*_factor for volt-watt control) can be used to either speed up the arrival at the solution on the volt-var curve (with the potential of introducing hunting or oscillatory behavior) or slow down the arrival at the solution, per control iteration.

The default for deltaQ_factor is 0.7. That means that the change in reactive power will only be 70% of the desired change (target) in reactive power per control iteration.

It is not too unusual to have maxcontroliter equal to 300 or 500 for cases involving multiple PVSystems under InvControl. Once the user is getting good solutions, then they can turn off logging of these control entries to the event log by setting EventLog=No, and this will speed things up a bit

Getting Access to CktElement data via User-Written DLLs

Question

In my user-written DLL code how can I get access to currents in neighboring devices?

Answer

In the New procedure of your DLL, which make a new instance of your user-defined model, you get a pointer to the CallBacks struct:

```
FNew:    Function(Var GenVars:TGeneratorVars; Var DynaData:TDynamicsRec; Var
CallBacks:TDSSCallBacks): Integer;  Stdcall;// Make a new instance
```

This struct is defined as follows:

```
TDSSCallBacks = Packed Record
```

```
MsgCallBack: Procedure (S : pAnsiChar; Maxlen:Cardinal);Stdcall; {Make use of DSS
        Message handling}

        {Routines for using DSS Parser.  This allows you to write models that accept
         syntax like other DSS scripts.}
GetIntValue: Procedure(var i : Integer);Stdcall; {Get next param as an integer}
GetDblValue: Procedure(var x : Double); Stdcall;  {Get next param as a double}
GetStrValue: Procedure(s : pAnsiChar; maxlen : Cardinal); Stdcall;
             {Get next param as a string <= maxlen characters  (Cardinal = 32-bit
        unsigned)}
             {caller must allocate space for s (Maxlen chars)}
LoadParser:  Procedure(S : pAnsiChar; maxlen : Cardinal); Stdcall; // Copies a string
        into a special instance of the DSS parser
NextParam:   Function (ParamName : pAnsiChar; Maxlen : Cardinal):Integer; Stdcall;
             {Advance to the next parameter and
              Get name of the param just retrieved, if one was given.
              Returns length of parameter found.  If 0, then end of string.
              This is to handle the syntax "paramname=paramvalue" commonly used in DSS
        scripts
              Copies the string to the location specified by s up to maxlen
        characters.
              Caller must allocate space (Maxlen chars)}

DoDSSCommand:             Procedure(S : pAnsiChar; Maxlen : Cardinal); StdCall;
GetActiveElementBusNames: Procedure(Name1 : pAnsiChar; Len1 : Cardinal; Name2 :
        pAnsiChar; Len2 : Cardinal); StdCall;
GetActiveElementVoltages: Procedure(Var NumVoltages : Integer; V : pComplexArray);
        StdCall;
GetActiveElementCurrents: Procedure(Var NumCurrents : Integer; Curr : pComplexArray);
        StdCall;
GetActiveElementLosses:   Procedure(Var TotalLosses, LoadLosses, NoLoadLosses :
        Complex); StdCall;
GetActiveElementPower:    Procedure(Terminal : Integer; Var TotalPower : Complex);
        StdCall;
GetActiveElementNumCust:  Procedure(Var NumCust, TotalCust : Integer); StdCall;
GetActiveElementNodeRef:  Procedure(Maxsize : Integer; NodeReferenceArray :
        pIntegerArray);  StdCall;// calling program must allocate
GetActiveElementBusRef:   Function(Terminal : Integer) : Integer;  StdCall;
GetActiveElementTerminalInfo: Procedure(Var NumTerminals, NumConds, NumPhases :
        Integer); StdCall;
GetPtrToSystemVarray:     Procedure(var V : Pointer; var iNumNodes : Integer);
        StdCall; // Returns pointer to Solution.V and size
GetActiveElementIndex:    Function() : Integer; StdCall;
IsActiveElementEnabled:   Function() : Boolean; StdCall;
IsBusCoordinateDefined:   Function(BusRef : Integer) : Boolean; StdCall;
GetBusCoordinate:         Procedure(BusRef : Integer; Var X, Y : Double); StdCall;
GetBuskVBase:             Function(BusRef : Integer) : Double; StdCall;
GetBusDistFromMeter:      Function(BusRef : Integer) : Double; StdCall;

GetDynamicsStruct:        Procedure(var pDynamicsStruct : Pointer); StdCall;  //
        Returns pointer to dynamics variables structure
GetStepSize:              Function() : Double; StdCall;  // Return just 'h' from
        dynamics record
GetTimeSec:               Function() : Double; StdCall; // returns t in sec from top
        of hour
GetTimeHr:                Function() : Double; StdCall; // returns time as a double in
        hours

GetPublicDataPtr:         Procedure(var pPublicData : Pointer; Var PublicDataBytes :
        Integer); StdCall;
GetActiveElementName:     Function(FullName : pAnsiChar; MaxNameLen : Cardinal) :
        Integer; StdCall;
```

```
GetActiveElementPtr:      Function() : Pointer; StdCall;  // Returns pointer to active
        circuit element
ControlQueuePush:         Function(Const Hour:Integer; Const Sec:Double; Const Code,
        ProxyHdl:Integer; Owner:Pointer):Integer; StdCall;
GetResultStr:             Procedure(S : pAnsiChar; Maxlen : Cardinal); StdCall;
```

From this you can access things in the OpenDSS data structures that you would not easily have access to otherwise, for example to use the simple error message mechanism, you could do something like:

```
CallBack^.MsgCallBack(pAnsichar(AnsiHelpStr), Length(HelpStr));
```

Where Callback is a local variable containing the pointer to the struct that is passed via the New procedure call. Note that strings passed between DLLs and OpenDSS are limited to ANSI strings for now. Also, note the string length argument. Delphi doesn't need this, but if you were writing your DLL in Fortran, it would push another argument, length, onto the stack.

I would use the DoDSSCommand procedure to select an element and then I can get access to that element's currents, losses, power, and voltages:

```
MyCmdString= 'Select Line.MyLine 1';

CallBack^.DoDSSCommand(pAnsiChar(MyCmdString), Length(MyCmdString));

CallBack^.GetActiveElementCurrents(NumCurrents, MyCurrents);

// where MyCurrents is a pointer to a complex array
```

You can also get access to the System V array, which is all the node voltages:

```
CallBack^.GetPtrToSystemVarray(MyVarray, NumberofNodes);

// then ...

Avoltage := MyVarray^[index]; // This is a complex number
```

There is not global current array. Currents are computed as needed by calling the device for which the currents are desired.

Keep in mind there is potential danger in tinkering with the global arrays. You can break your model if you aren't careful.

Also, let me remind you that the design philosophy of OpenDSS is that no circuit element knows anything about another circuit element. There are few exceptions (meters and control elements). If your model needs to know something like voltage or current from some other circuit element in order to function, you might want to reconsider the design.

Getting the System Jacobian Matrix

Question

I see there is a Newton solution methods. How can I get the Jacobian?

Answer

The system Y matrix is used as the Jacobian in the Newton algorithm. It is the same Y matrix as in the "normal" fixed point solution. So you can get it from either solution algorithm.
Note that you get the Y matrix in different formats depending on whether you get it from "Show Y", "Export Y", or through the SystemY property of the Circuit interface when driving the program from COM.
One thing you will have to decide is whether you want Load and Generator equivalents in the Y matrix for your algorithm. If you define Load and Generator objects you will get the Norton equivalent admittances in the Y matrix. So if that messes up your algorithm (which it frequently does), do a Solve of some sort before you define Load and Generator objects, etc., and extract the Y matrix immediately after the Solve command.

The Y matrix is a common nodal admittance matrix in all formulations that we currently have implemented. You could form a different set of equations such as an augmented Y matrix like SuperHarm uses, but we just haven't done it. In fact, the "normal" fixed point solution using Y works just fine for nearly all the things we are doing.

Early on, when the system got a little ill-conditioned we used a Newton (technically a Newton-Raphson) method to get better convergence than the fixed-point method. Now, I don't know if there is really much difference between the two after implementing some tricks to get the fixed-point to work better.

The fixed-point solution might be written:

$V_{n+1} = [Y]^{-1} * I_{comp}(V_n)$

Where I_{comp} is the *compensation* current from the non-linear loads, etc. (See the User Manual; **OpenDSSManual.pdf**)

The Newton method in OpenDSS might be written:

$V_{n+1} = V_n - [Y]^{-1} * I_{term-n}$

Where I_{term} is the array of terminal current sums for all nodes. This formulation tries to force the current to sum to zero at each node. It will converge a little better for cases that are really nonlinear and/or ill-conditioned. However, we currently don't update Y between iterations, so it really isn't much of an advantage over the fixed point and takes twice as many arithmetic operations. So we prefer the fixed point for time-sequential simulations for the obvious reason that if you are running an 8760-hour simulation it is faster.

So the Y matrix is used as the Jacobian in this formulation. As long as the derivatives point in the right direction, the solution is likely to get there eventually. It would converge in fewer iterations if we

updated the Y matrix for the nonlinear load derivatives at each iteration as my friends in Brazil (Sando Carneiro, Jr., Jose Luiz Pereira, et. al.) do, but iterations are cheap computationally. Rebuilding Y is more expensive. Besides, once you get a time series simulation going, there is frequently only 2 iterations between time steps -- one to get there and one to prove you got there -- because the solution at one time step is a real good starting point for the solution at the next.

The history is: we tried a crude fixed point in 1997 when it was in a much more primitive state than it is now and it worked for nearly all distribution systems we were simulating. We later ran into a transmission system that wouldn't converge and solved that with the Newton algorithm. But don't confuse that with the Newton-Raphson algorithm you find in most power system analysis text books. That uses a different formulation of the power flow problem. But it is still a system of non-linear equations solved by Newton's method.

Subsequently, we have evolved the power conversion models and the method of obtaining the first guess to the power flow solution so that the fixed-point can solve nearly everything we do. Getting a better first guess at the voltages was a key. That solved a lot of convergence problems.

Precision Issues with MATLAB and Python

Shortly after posting OpenDSS on line we started hearing about apparent precision problems from MATLAB users driving OpenDSS through its COM interface. The usual complaint was that OpenDSS.EXE gave a good answer but MATLAB yielded bad values for the currents and powers. We suspected that MATLAB was setting the precision of the math because the problem came up with very short lines with tiny impedances. OpenDSS uses a nodal admittance formulation and computes the node voltages from solving *I=[Y]V*. Then the voltages are used to compute the currents, which are subsequently used to compute the powers.

If a branch in the circuit has very small impedance, the voltages on both ends of the branch have nearly the same values. Then we multiply the difference in the voltages, which could be inaccurate, by very large admittances, sometimes yields very large, incorrect currents.

Recently the same complaint arose when using Python to drive OpenDSS. User Christopher Williams isolated the problem to the switches he was modeling with very small impedances. A quick search of the internet came up with the suggestion that something called the *FPCW* (floating-point control word) might be at the heart of the matter. Adam Birchfield was tasked with finding out how to set the FPCW from Python and testing to see if it would explain the differences we are seeing. His post on the site explains the problem and resolution quite well:

> Christopher, thank you for all your work to demonstrate this calculation inconsistency with using Python through the OpenDSS COM server. I was able to reproduce the inconsistency using the IEEE 13 bus network switch, when the switch was reduced to a resistance of 1e-11 ohms. I discovered that, at least on my machine with an Intel(R) Core(TM) i5-3470 CPU using Windows 7

Professional, the Floating Point Control Word (FPCW) can be controlled in a Python program, and the default precision level is 64-bit with a 53-bit mantissa. Since OpenDSS calculates with 80-bit precision on my machine, a small switch like that can produce erroneous currents on 64-bit precision when the correct currents are given for 80-bit. When I manually changed the FPCW precision level in Python to 80-bit, the results matched OpenDSS exactly.

The following link shows how to use the C function "_controlfp", which allows one to change the FPCW in C. The first argument should be 0x00010000 for 64/53 precision and 0x00000000 for 80/64 precision. The second argument should be 0x00030000 to mask the FPCW for the precision bits.

https://msdn.microsoft.com/en-us/library/e9b52ceh.aspx

The C function "_controlfp" can be accessed through the DLL "msvcrt" which I access through the module "ctypes.cdll." Here's my Python code to access the FPCW:

```
1  # Test program to change FPCW and run something on OpenDSS
2  import win32com.client
3  from ctypes import cdll
4
5  def opendss_do_something():
6      # Some basic code to run OpenDSS and get some results
7      results = "123"
8      engine = win32com.client.Dispatch("OpenDSSEngine.DSS")
9      engine.Start("0")
10     engine.text.Command = "redirect (IEEE13Test.dss)"
11     #...do some more stuff to get currents we want
12     return results
13
14 def main():
15     # run before changing FPCW
16     results_prelim = opendss_do_something()
17
18     # now change FPCW to 64-bit and run
19     cdll.msvcrt._controlfp(0x10000, 0x30000)
20     results_64 = opendss_do_something()
21
22     # now change FPCW to 80-bit and run
23     cdll.msvcrt._controlfp(0x00000, 0x30000)  # <-- this is the magic line
24     results_80 = opendss_do_something()
25
26     print results_prelim, results_64, results_80
27
28 if __name__=="__main__": main()
```

With the right model, you should see a difference between the 64-bit results and what you get from OpenDSS directly. However, the 80-bit results should match OpenDSS exactly.

In OpenDSS, all floating-point numbers are stored as *doubles*, or 64-bit values. The default math co-processor precision is used, which on the modern Intel processors is IEEE 80-bit math. So when OpenDSS.EXE is run there could be up to 11 more bits accuracy when a math operation is performed. This obviously makes a difference. Both MATLAB and Python apparently change the FPCW by default to limit the precision to 64 bits. One reason for doing this might be so that you get the same answer on different hardware platforms. Nevertheless, you can change the FPCW with either program as described above if you want the results of a simulation performed through the COM interface to exactly match the results from the standalone EXE.

This problem can also be alleviated by more judicious modeling. If you choose to model switches explicitly, choose are small but reasonable value. When you define a Line object with “SWITCH=Y” the default is a line with basically Z=0.001 +j 0.001 ohms. That should be sufficiently small for most simulations. You may have to go a little smaller on some LV circuits.

The problem of short lines was anticipated when OpenDSS was originally designed. To avoid having to specifically model switches, each terminal of each circuit element is assumed to have a switch that you can control with the OPEN and CLOSE commands.

Fuse Simulation

Question

I've run a number of simulations with various types of faults successfully. Ideally I would like to include fuse-bahavior as I'm interested in what can be observed of an LV-fault at the MV-side of the MVLV-transformer. I did do a succesful demo-run with a fuse, but that only resulted in zero current.

Ideally I would like to get a simulation result showing me for a particular LV-fault and network configuration:
1. the time it takes for the fuse to blow for the given fault
2. the cut-off current of the fuse (lower than the prospective current for short fuse times) at the moment the fuse blows
3. the other currents present in the system at the moment of cut-off

I haven't been able to find clear documentation on the fuse feature, nor gave the code a definitive answer. Normally this might be more appropriate for a time-based simulator, but OpenDSS should be able to give me the indication as listed above.

Another way of going about is by determining the short-circuit current at the LV-side and determine the fuse times and the cut-off currents using the datasheet. I did that already, but this isn't as accurate and as fast as a full-featured simulation. Also I would have to translate the LV currents to the MV-side to determine the observable effect.

Does OpenDSS have built-in features I have overlooked that might help me with this simulation?

Answer

OpenDSS performs overcurrent protection analysis by simulating the fault. If you are applying the FAULT object in Snapshot mode, the control mode is STATIC, which means it will execute all control actions in time sequence but at the same time step. So it simulated the fuse action, but it appears to have happened all at once.

To see the time variation in FUSE and other protective device actions (RECLOSER, RELAY) you would use Dynamics mode with a small time step. I loaded up the IEEE 123 test feeder, which you should have installed with OpenDSS and then added this script after an initial solve:

```
/*
   FUSE model demo for IEEE 123 test feeder supplied with OpenDSS

   Define a lateral fuse and put a Monitor at the same location
   Fuse will have the same number of phases as the monitored element (1 in this case)
   Two TCC_Curve objects for fuses are defined by default for every circuit: KLINK and
TLINK

*/

New Fuse.Fuse1 line.L2 1 Fusecurve=TLINK ratedcurrent=65  ! 65T fuse
New Monitor.Fuse1 Line.L2 1

solve  ! snapshot prior to dynamics mode to get Monitor and Fuse assimilated into
circuit

// solve a few steps (1 60-Hz cycle) in Dynamics mode just to get everything going
Solve mode=dynamics stepsize=0.0016666666 number=10

// Add a fault at the end of the lateral
New Fault.F1 phases=1 Bus1=6.3
Solve number=100   ! solve 10 cycles, still in Dynamics mode

// check out the results of the simulation
show monitor fuse1

Export monitors fuse1
Plot monitor object= fuse1 channels=(3 )  ! currents
Plot monitor object= fuse1 channels=(1)   ! voltage
Show Eventlog
```

This simulates a lateral fuse on the 1-phase lateral on phase 3 near the substation with a fault at the end of the lateral (node 6.3). You should be able to see the current in the line go up to about 6000 A and the voltage at the fuse location drop during the fault. Then when the fuse blows, the current should go to zero and the voltage bounces back up.

A FUSE element is a CONTROL element. That is, it monitors the current in Terminal 1 of Line.L2 but isn't actually in the circuit. When the FUSE determines it should blow it opens the terminal of LINE.L2, which is carrying current in the circuit.

Sometimes you have to put the MONITOR at a different location to get the right values, but it appears to work fine in the simulation.

So that is basically how you do it. If you want to have some fun put a RECLOSER and FUSE in series in a circuit with at least one fast curve (e.g. the A curve defined when the circuit is created) and a delayed curve. When you put a FAULT downstream of the FUSE, you should see the RECLOSER operate on the fast curve and then the FUSE blow after the reclose a couple seconds later. Can be a little tricky to set up, but it worked the last time I tested it.

FaultStudy Mode

Question

I have read manual and papers I found with code about fault study and dynamics mode (OpenDSS FaultStudy Mode.pdf and OpenDSS Dynamics Mode.pdf). I found that a Generator object converts into a form of Thevenin equivalent when switching to Dynamics, but in FaultStudy paper it says that Voltage objects and any Generators are converted to Norton equivalents after they has been converted to dynamic equivalent. I am confused, when switching to Dynamics are Generators converted from Norton to Thevenin equivalent and then back to Norton?
Thank you for answers in advance.

Answer

It is interesting you should be asking this at this time. Last week, I did an extensive investigation into how OpenDSS is handling the FaultStudy mode because the X/R values weren't coming out correct for one user. We fixed that by adding the **XRdp property** to the Generator model to properly set the resistance when in Dynamics mode. You should download the latest version to get that bug fix.

What I didn't realize (probably simply forgot – it was done before 2003) is that when you enter FaultStudy mode, OpenDSS switches to Dynamics mode automatically. So *Solve mode=FaultStudy* will build *Y* and the short circuit matrices (*Zsc* and *Ysc*) using X_d' for the Generator model. So you no longer need to set Mode=Dynamics and do one or two small time steps like you used to have to do. Of course, you will still do that when you model a fault by applying one or more FAULT objects.

I didn't realize this because the fault currents were coming out low, so I assumed Y was being built from the equivalent admittance for the power output as it is for other solution modes. The fault currents are computed from the pre-fault voltage and the *Ysc* at each bus computed from Y. So the trick is to get the prefault voltages as close to 1.0 per unit as possible before performing Solve Mode=Faultstudy. Then the fault currents will more closely match conventional short-circuit analysis results with generator contributions.

The user who was having the difficulty and I came up with the following script to accomplish this and I got a report today that it is working quite well.

```
...
(set source at 1.0 per unit)
```

```
Solve loadmult=0.001 genmult=0.001
Solve Mode=Faultstudy
Export Faultstudy or Show Faultstudy
...
```

On OpenDSS Solution Techniques

Question

I have a few basic questions to better understand the solution techniques behind OpenDSS and better use its capabilities.

1- OpenDSS is a Harmonics solver rather than power flow, what does that mean exactly? What's the solution techniques behind solving power flow in OpenDSS and how can it handle harmonics?
2- How does OpenDSS solve for electromechanical transient?? And if the solution is frequency domain (phasors) doesn't that assume a constant frequency for the solution?
3- If OpenDSS solves for harmonics does that make it similar to EMT even though it doesn't solve time-domain equations?
4- I see that OpenDSS is mostly being applied for Power flow and QSTS simulations, so what are some of the application of OpenDSS for Dynamic and Harmonic studies?

Answer

1. The basic solution methods as far as building the Y matrix and modeling lines, transformers, and other elements were borrowed from a series of programs developed for harmonics analysis. Ideas came from SuperHarm, V-Harm, and DFHA harmonics analysis programs as well as the METAP program from the Mid 1970's (see Talukdar's paper from about 1976). The harmonics programs are all linear, multi-frequency, multi-phase, phasor solvers that build a nodal admittance matrix at each frequency and solve it directly. They all had object-oriented programming stuctures even before OOP became popular, making for modular robust modeling. OpenDSS retains these capabilities but also adds power flow solution capability for the fundamental frequency. As I said in an IEEE paper about 3-4 years ago, it is easier to teach a harmonics solver how do a power flow problem than to make a power flow algorithm solve the harmonics problem. The harmonics solver heritage gives OpenDSS the abiility to model just about any circuit topology one might encounter in actual power systems. Since most harmonics problems occur in distribution or industrial systems with complex, unbalanced topologies, the heritage imparts these capabilities to the power flow problem -- with a few, minor tweaks to modeling the non-linear elements (PC Elements in OpenDSS terminology). The solution technique is described in one figure in the

OpenDSS user manual and that is all it is. Many users are looking for something more complicated, but it's not.

2. OpenDSS is structured more like a transient stability program than a traditional power flow program. Certain PC elements like Generators are provided with traditional, simple single-mass swing equation capability. PC Elements like Generators can employ a simple predictor-corrector algorithm to model differential equations. When in Dynamics mode, the integration routines get automatically called to compute the new values of the state variables. Yes, the network frequency is generally assumed to be constant, but rotating machines can change speeds. This is pretty much what you have on a distribution circuit where a disturbance is not likely to change the frequency of the bulk power system. So we use this capability for islanding and single-phasing analyses primarily. However, we do have a few users experimenting with changing the network frequency (Set Frequency=...) which forces a Y matrix rebuild. As we develop better microgrid capabilities, I expect this to evolve.

3. Its stucture is similar to some EMT solvers, and you could probably add the appropriate models and support algorithms. I've actually had a few grad students say they wanted to do that, but I think it turned out to be a bigger chore than they anticipated. We've got access to really good EMT solvers, so there's little incentive for us to invest in that capability. It is our opinion that we can tackled 95% of the problems we are interested in solving from the planning perspective in the phasor domain. Occasionally, we will find we have to model the switching of an inverter in detail or a ferroresonance problem in an EMT model. But we've got more issues to study with phasor models than we can get to. If we can do it acceptably with average models in the phasor domain, solution times are more than 100 times faster and we can evaluate many more options in the same amount of time. Some of the EMT simulations we run are done to prove to skeptical clients that OpenDSS got the right answer!

4. The original application of dynamics was for islanding analysis of rotating-machine DG. Can the DG change voltage or frequency fast enough to get off line before the reclose? We have also used it for storage simulations of ultracap-based storage elements that can charge and discharge quickly (uses a user-written "DynaDLL"). We also use this mode for some fault-current contribution analyses and analyses of single-phasing of DG. We expect this mode will be used extensively for our upcoming microgrid research. Harmonics mode is used for any number of harmonics analysis problems. We have some serious users studying the impact of EV chargers using the recently-added sequential

harmonics mode. The EPRI Harmonics Evaluation Module uses this mode extensively and I have seen a demo of a similar app done for CEATI. Use your imagination!