SANDIA REPORT

SAND2018-12275
Unlimited Release
Printed October 2018

Application Note: Coupled Simulation with the *Xyce*™ General External Interface

Thomas V. Russo, Russell Hooper

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

SAND2018-12275
Unlimited Release
Printed October 2018

Application Note: Coupled Simulation with the *Xyce*™ General External Interface

Thomas V. Russo
Electrical Models and Simulation

Russell Hooper
Multiphysics Applications
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1177

Abstract

The **Xyce** General External Coupling API is documented.

Contents

List of Figures

1. Introduction

Xyce is Sandia National Laboratories' SPICE-compatible high-performance analog circuit simulator, written to support the simulation needs of the laboratories' electrical designers. It has the capability to solve extremely large circuit problems on large-scale parallel computing platforms, and contains device models specifically tailored to meet Sandia's needs.

During the development of other simulation codes at Sandia, it has often been necessary to include simulation of a driving circuit to provide boundary condtions for some other physical domain such as electromagnetics or device physics. One approach often taken in these codes is to build simple circuit simulation capability into their primary simulator, but this often limits the driving circuit to very simple designs; in many cases, the circuit simulators in these codes could only simulate linear devices such as resistors, capacitors, and inductors. Since **Xyce** is a fully capable, high-performance circuit simulator with advanced device models, there is value in providing a mechanism for external simulation codes to use **Xyce** as their circuit simulator.

In earlier work[1], we developed a mechanism for coupling **Xyce** to ALEGRA[2] in order to simulate coil guns, which we called "Xygra." In this scheme, ALEGRA would simulate the electomagnetics and dynamics problem for a small time step, then pass information to **Xyce** that allowed it to simulate the behavior of the circuit over that small time step as if the entire electromagnetics problem were a single circuit device. Control passed back and forth between **Xyce** and ALEGRA until the entire desired simulation period was complete. The Xygra coupling scheme was very much tailored to the coil gun problem, in that the specific form of the information passed from ALEGRA to **Xyce** was fixed, and the device took parameters that explicitly referenced features of that problem, such as the number of windings in a coil and number of coils.

Given the success of the Xygra approach for the coil gun problem, we were asked to explore the possibility of generalizing the coupling interface so that it could be used more flexibly on a wider variety of circuit/mesh coupling problems. Furthermore, there is a growing interest in coupling frequency-domain electromagnetic simulators to **Xyce**, and performing the frequency-domain analyses that **Xyce** provides such as harmonic balance.

The experiences of the Xygra project and the desire to extend the usefulness of the coupling capability have led to the development of a new coupling scheme that is extremely general, and that is extensible to the frequency domain as well as time domain. Further, as part of the development we have created a general mechanism for communicating simulation results from **Xyce** to the coupled simulator that eliminates the need for reading and parsing **Xyce** output files, and allows the external code to treat **Xyce** solution information on equal footing with its own for analysis, output, and plotting purposes.

This report documents how this coupling scheme works, the API used to access it, and its limitations.

1.1 Target Audience

This application note is primarily intended for developers of existing simulation codes who wish to use **Xyce** in order to add circuit simulation capability to existing capabilities.

The coupling mechanism described in this report is not suitable for all coupled simulation problems. It is likely to be of most use for problems where the external problem is linear, or can be reduced to a linearized problem over short time intervals. Highly nonlinear problems with behaviors that are much faster than the behaviors of the coupled circuit may require an alternate coupling scheme not addressed here.

Xyce is an application code written in C++, and this document is aimed at developers who wish to couple other external codes written in C++ to **Xyce**. Coupling to codes written in other languages may have additional complicating details that are not explored in this document.

2. **Xyce** overview

In this chapter we give a very brief overview of the mathematical formulation used in **Xyce** in order to establish the context for the coupling mechanism. For more detail, the user is referred to the **Xyce** mathematical formulation document[3].

2.1 DAE Formulation Overview

At its core, **Xyce** solves the Differential-Algebraic Equation (DAE)

$$\vec{F}(\vec{x}) + \frac{d\vec{Q}(\vec{x})}{dt} - \vec{B}(t) = 0$$

for the unknowns $\vec{x}$. The specific forms of $\vec{F}$, $\vec{Q}$, and $\vec{B}$ are established using the formalism of modified nodal analysis.[4]. When needed, we will refer to these three vectors collectively as “the DAE vectors.”

In a general circuit, devices are connected together at *nodes*, and in the simplest cases the DAE represents the requirement that the currents flowing into or out of a node must sum to zero (Kirchhoff’s Current Law, an expression of charge conservation). The solution vector $\vec{x}$ is made up mostly of the electrical potentials (voltages) at the nodes.

Some devices (such as voltage sources and inductors) may require additional unknowns (branch currents) to be added to the solution vector, and additional equations (branch equations) to be introduced into the DAE system. Esoteric devices can even have solution variables and equations unrelated to the electronic problem (e.g., chemical reaction dynamics), so long as they can be written in the correct DAE form.

2.2 Device Model overview

Xyce is structured so that each device present in the circuit has associated with it a code object responsible for loading the elements of $\vec{F}$, $\vec{Q}$, and $\vec{B}$ associated with the nodes between which it is connected. Generally speaking, most elements of $\vec{F}$ are currents, and most elements of $\vec{Q}$ are charges or fluxes (i.e., terms that must be differentiated with respect to time in order to obtain currents). Elements of $\vec{B}$ are explicitly time-dependent and solution-independent forcing terms such as those from independent sources.

2.2.1 Netlisting basics

A circuit is input to **Xyce** through a text file known as a "netlist." Complete netlist syntax descriptions are provided in the **Xyce** Reference Guide [5], but for the purposes of this document we will provide a brief introduction to establish a context for our examples for the benefit of users who may not already be familiar with SPICE simulation.

Each line of the netlist represents either a discrete device in the circuit or a directive controlling simulation parameters. Directive lines always begin with a period, while device lines always begin with a letter. Netlists are case insensitive.

The first line of any netlist is always taken to be a title comment and is ignored.

Any line that begins with an asterisk or a space is taken to be a comment and is ignored. A common netlisting error is to attempt to indent lines for readability, which results in the lines being ignored. A semicolon on a line marks an in-line comment; all characters after the semicolon up to the end of line are ignored.

The first character of a device line determines its device type (e.g., resistor, capacitor, diode, etc.), and is followed by a unique device name (the characters following the first, up to but not excluding the first space on the line), then additional information to determine the connectivity of the device and its controlling parameters.

If the first character of a line is a plus sign, the line is taken as a continuation of the previous non-comment line.

Numerical quantities may have multipliers applied by following the number with a string representing the multiplier. The recognized multipliers are:

- "f" (1×10^{-15})
- "p" (1×10^{-12})
- "n" (1×10^{-9})
- "u" (1×10^{-6})
- "m" (1×10^{-3})
- "k" (1×10^{3})
- "meg" (1×10^{6})

- “g” (1×10^9)
- “t” (1×10^{12})

Unrecognized characters following a numerical quantity or the multiplier are ignored. It is common in SPICE netlists to take advantage of this feature to place unit names after the quantities knowing that they should be ignored (e.g., “1pf” for “1 picofarad” or “1Ohm”), but one must be careful not to use unit names that may be mistaken for multipliers (e.g., “1Farad”, which would be interpreted as 1×10^{-15}). Note also that since netlists are case insensitive, it is necessary to have the three-character text “meg” to denote millions. It is a common error to use “M” when “meg” is intended, which results in a significant error because of the case insensitivity.

2.2.2 Circuit nodes, internal nodes, and branch currents

Each device is connected to some number of circuit nodes, as enumerated in the circuit netlist given to **Xyce**. For example:

```
R1 1 2 2K
```

defines a resistor “R1” connected between nodes 1 (the positive node) and node 2 (the negative node), with a resistance of 2000 ohms.

Connecting nodes that are specified on a netlist instance line like this are known as “external” nodes for the device — and in a properly connected circuit, each node should have at least two devices connected to it. Devices can also have internal variables which may serve as nodes in the devices’ equivalent internal circuit, but may also be additional variables with additional associated equations such as branch currents and branch equations.

The number of external variables is fixed as the number of connecting nodes specified on the netlist line, but the number and meaning of internal variables are defined by the device itself. The device set-up code has a set of accessor functions used to query each device for how many internal variables it needs; this is used to determine the size and layout of the solution and other vectors. Additional queries are performed to determine the pattern of interdependency between variables (known as the “Jacobian stamp”), so that appropriate elements of the sparse Jacobian matrices can be allocated.

Once the circuit topology is determined, it is fixed for the duration of a simulation. The device is provided information so that it can find the locations of the elements of the solution and DAE vectors corresponding to its variables, and the locations of the Jacobian matrix elements that it needs to fill.

2.2.3 Sign convention

By convention, as a device loads its contributions to the various vectors, it adds a positive contribution to the $\vec{F}$ vector element associated with a node when it is drawing current (or for $\vec{Q}$, charge) from that node, and a negative contribution when sinking current into that node. As shown by the sign of $\vec{B}$ in the general DAE, the convention for that vector is the opposite: current drawn from a node adds a negative contribution to $\vec{B}$.

2.2.4 Device model examples

In order to establish more context for the general external coupling mechanism, this section shows how several basic linear electrical devices must load the DAE vectors.

Resistor

The current-voltage relationship of a simple linear resistor is Ohm's law

$$V = IR$$

or

$$I = GV$$

with R being the resistance, and G being the conductance (inverse of resistance). No differentiation with respect to time is needed, nor are there any explicitly time-dependent terms, so this device would contribute only to the $\vec{F}$ vector.

The netlist line

```
R1 1 2 2K
```

defines a $2,000\Omega$ resistor connected between node 1 (its positive node) and node 2 (its negative node). By convention, the resistor current is positive when the positive node is at higher potential than the negative node, meaning that current is drawn from the positive node and sunk into the negative node.

Therefore, this device would need to retrieve the voltages V_1 and V_2 (the potentials at nodes 1 and 2) from the solution vector, compute the voltage drop across the device, and thence the resistor current:

$$I_{res} = (V_1 - V_2) * (\frac{1}{2000}) .$$

This resistor current would then be added to the element of $\vec{F}$ corresponding to node 1 and subtracted from the element of $\vec{F}$ corresponding to node 2. Considering only the contributions of this device and observing the appropriate sign convention, we have

$$\begin{aligned} F_1 &= \frac{(V_1 - V_2)}{2000} \\ F_2 &= -\frac{(V_1 - V_2)}{2000} \end{aligned}$$

Capacitor

A simple linear capacitor with constant capacitance holds a charge

$$Q = CV$$

with C being the capacitance, and therefore the current through it is

$$I = \frac{dQ}{dt} \ .$$

The current through this device contains only a term dependent on a time derivative, and has no explicitly time-dependent terms, so the device model loads only the $\vec{Q}$ vector.

The netlist line

```
C1 1 2 1p
```

defines a capacitor of $1pF$ between nodes 1 and 2.

To load the DAE vectors, this device would compute the charge on the device as $Q = C * (V_1 - V_2)$. It would then add this charge to the element of $\vec{Q}$ corresponding to node 1, and subtract it from the element corresponding to node 2. Again, considering only the contributions of this one device,

$$\begin{aligned} Q_1 &= (V_1 - V_2) * 1 \times 10^{-12} \\ Q_2 &= -(V_1 - V_2) * 1 \times 10^{-12} \ . \end{aligned}$$

Inductor

A linear inductor is an example of a device that must have an additional internal variable and associated equation. The voltage-current relationship for an inductor is

$$V = \frac{d\phi}{dt}$$

with ϕ being the flux $L * I$, L being the inductance, and I being the current through the device.

For the inductor represented by the netlist line

```
L1 1 2 1u
```

representing a $1\mu H$ inductor between nodes 1 and 2, this may be written in the appropriate DAE form as

$$\frac{dLI}{dt} - (V_1 - V_2) = 0 \; .$$

The current out of node 1 and into node 2 is I. Unlike devices such as the capacitor or resistor, the device cannot compute the inductor current directly knowing only the voltages of the terminals, and therefore the inductor current must be added as an unknown, known as the inductor's "branch current", and must add an equation to the DAE system so that this unknown may be solved for. We will call this unknown I_{branch}.

As a result, the inductor device must declare that it has an additional internal unknown; **Xyce** will set up all of the vectors and matrices needed to solve the circuit problem to include this additional variable.

During device evaluation, the inductor must obtain the value of I_{branch} from the solution vector corresponding to the branch variable requested in addition to the nodal voltages V_1 and V_2. It then adds I_{branch} to the element of $\vec{F}$ corresponding to node 1 and subtracts it from the element corresponding to node 2. But in addition to loading the elements for the two nodes, it must also set the elements of $\vec{F}$ and $\vec{Q}$ corresponding to the extra variable (branch current) — this extra equation is known as the "branch equation". The branch equation is

$$\frac{dLI_{branch}}{dt} - (V_1 - V_2) = 0 \; ,$$

and so the device must load the part that is not differentiated with respect to time, $-(V_1 - V_2)$, into the element of $\vec{F}$ corresponding to the branch current, and LI_{branch} into the corresponding element of $\vec{Q}$ (because this term must be differentiated with respect to time).

As with the resistor and capacitor, considering only this device's contributions to the DAE vectors, we have:

$$\begin{aligned} F_1 &= I_{branch} \\ F_2 &= -I_{branch} \\ F_{branch} &= -(V_1 - V_2) \\ Q_{branch} &= I_{branch} * 1 \times 10^{-6} \; . \end{aligned}$$

Serial RLC network

Consider a two-terminal device implementing a serial RLC network such as the one shown in figure 2.1. The two terminals, labled “N1” and “N2” in the figure, are the device’s external nodes. There is an internal node “NI” between the inductor and capacitor. It is also necessary to include a branch current variable to evaluate the inductor, labeled “Branch” in the figure. Because the current through the resistor and inductor must be the same, it is unnecessary to include a third internal variable for the voltage at the node between the resistor and inductor — one can use the $V = IR$ form of Ohm’s law rather than the $I = GV$ form to construct the equation set for this macromodel.

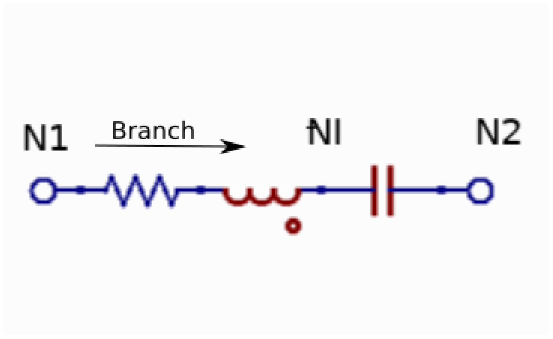

Figure 2.1. Series RLC subcircuit as two-terminal device.

The current out of node N1 and into node NI is simply the branch current. The potential drop across the resistor and inductor together is $I_{branch}R + \frac{dLI_{branch}}{dt}$, which must be equal to $V_{N1} - V_{NI}$; taken together, these terms form the branch equation for the device,

$$I_{branch}R + \frac{dLI_{branch}}{dt} - (V_{N1} - V_{NI}) = 0 \ .$$

The current out of node NI and into node N2 is simply $\frac{d}{dt}(C * (V_{NI} - V_{N2}))$.

The F and Q contributions for this device are therefore:

$$\begin{aligned}
F_{N1} &= I_{branch} \\
F_{NI} &= -I_{branch} \\
F_{branch} &= I_{branch} * R - (V_{N1} - V_{NI}) \\
F_{N2} &= 0.0 \\
Q_{N1} &= 0.0 \\
Q_{branch} &= L * I_{branch} \\
Q_{NI} &= C * (V_{NI} - V_{N2}) \\
Q_{N2} &= -C * (V_{NI} - V_{N2}) \ .
\end{aligned}$$

Here we have explicitly shown that this device contributes nothing to the N2 element of $\vec{F}$ or the N1 element of $\vec{Q}$.

Closing remarks on examples

The contributions shown above for each device must be added into the global $\vec{F}$ and $\vec{Q}$ vectors representing the entire circuit problem, and while we have not shown it, their derivatives with respect to the three solution variables must also be computed and added into the global DAE derivative matrices, $\frac{d\vec{F}}{dX}$ and $\frac{d\vec{Q}}{dX}$.

These simple linear devices demonstrate how device models in **Xyce** construct their contributions to the $\vec{F}$ and $\vec{Q}$ vectors, how a device model may require additional internal variables, and how additional equations may be added to the DAE system.

In order to couple **Xyce** to a simulator that does not simply implement standard circuit elements such as these, one must map the interfaces between the domains appropriately. For example, **Xyce** will be providing the voltages at the nodes as if they were point contacts, and is expecting that the DAE vectors be loaded with currents and charges as if they were feeding into a point contact. Thus, for coupling with a mesh-based simulator, **Xyce**'s voltages would need to be applied as boundary conditions on a set of interface elements such as a side set, and current densities computed by the mesh simulator at these interfaces would have to be integrated across the set to become nodal currents that would be loaded into the DAE vectors.

2.3 Numerical solution overview

Xyce solves the DAE via a nested set of solvers. Figure 2.2 illustrates the relationship between the solvers for a transient analysis[3]. The outermost solver varies for other analysis types. For example, in a DC steady state calculation the nonlinear solver is the outermost of these loops.

For the purposes of understanding how to use the coupling interface, the most important of these solves to understand is the nonlinear solve. The sequence of operations performed by the nonlinear solver step is illustrated schematically in figure 2.3.

Xyce's nonlinear solver implements Newton's method[3]. At each iteration of the nonlinear solver, all devices are called upon to load their contributions to the DAE vectors and matrices as computed from the current guess, $\vec{x}_n$. The DAE vectors and matrices are assembled along with terms resulting from the time integration method to create a nonlinear system

$$\vec{f}(\vec{x}, \frac{dx}{dt}, t) = 0 \ .$$

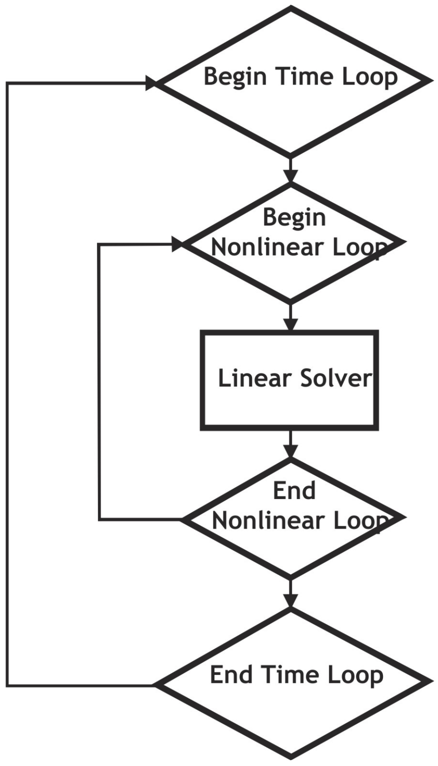

Figure 2.2. **Xyce** solver structure.

Evaluating this function at the current guess and applying Newton's method leads to a linear system $J_n \Delta\vec{x}_{n+1} = -f_n$ to be solved for the updated guess, where f_n is f evaluated at the guess $\vec{x}_n$, and J_n is its Jacobian. The new solution $\vec{x}_{n+1}$ is obtained as $\vec{x}_{n+1} = \vec{x}_n + \Delta\vec{x}_{n+1}$. The process is repeated until the convergence criteria are met.

Xyce's time integration loop also bears some discussion. **Xyce** uses adaptive time-stepping with error control. Loosely speaking, **Xyce** starts from a previously accepted solution, tentatively advances the simulation time by the current time step and predicts a first guess at the solution, which is then used as the initial guess for the nonlinear solver. If the nonlinear solver converges and the local truncation error computed for the time integration method is within tolerances, the timestep is accepted and simulation proceeds. If the solution at the tentative time is not accepted, the time step is reduced and a new solution attempted. This requires that the devices' models be written in such a manner that there is no assumption that they will be called only for monotonically increasing times — from the device model's point of view it may seem that time jumps backward if one of these tentative steps is rejected.

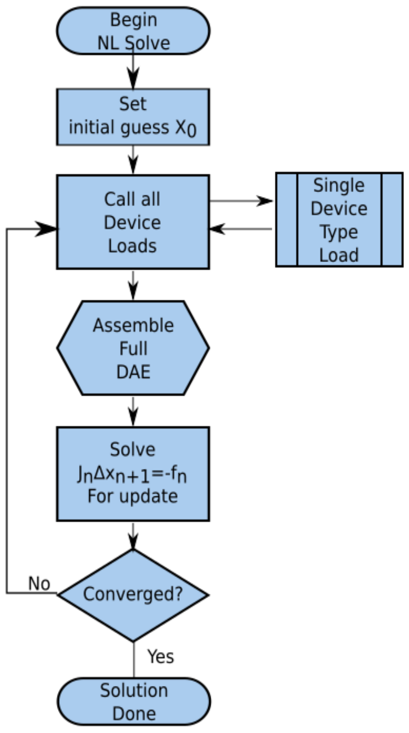

Figure 2.3. **Xyce** Nonlinear solver process.

2.4 Summary

The preceding sections provide the context needed to use the “General External” coupling mechanism. The coupling between **Xyce** and an external code will be handled as a black-box device, and the linking code between them will be responsible for providing **Xyce** with the information it needs to construct its DAE.

3. General External Coupling Interface

3.1 Overview

As shown in figure 2.3, at each step of the solution of the circuit problem, **Xyce** calls upon each device in the circuit to populate portions of the DAE vectors and their derivatives with respect to solution variables.

The approach we have taken to generalize the original approach to **Xyce**/ALEGRA coupling is to define a new device type (the “General External” device), whose device load function simply calls methods of a user-defined interface object (the “vector compute interface”). An external code defines such an interface object for each simulation domain required, and associates this object with a specific instance of the general external device in the **Xyce** circuit. Each pass that **Xyce** takes through its nonlinear solve triggers a call back to the methods of the interface objects for each general external device present in the netlist.

In broad strokes, the process of coupling is for the user to construct a circuit netlist that includes both the real circuit elements that **Xyce** will simulate and special general external devices which serve as black-boxes that actually invoke the vector compute interfaces provided by the external code. The external code then instantiates an object of type Xyce::Circuit::GenCouplingSimulator, which serves as a control object for **Xyce**. The external code calls methods of this object to read in the netlist, query for the existence of coupling devices in it, and to set up those coupling devices. As part of the set up of coupling devices, the external code calls methods of the GenCouplingSimulator object to associate a vector compute interface object with each coupling device. Once all set-up is complete, the external code calls methods of the GenCouplingSimulator object to begin a **Xyce** simulation.

This chapter will document how to use the general external device in a circuit netlist, and how to construct the vector compute interface class to allow an external code to couple to **Xyce** by acting as one or more compact models in a **Xyce** circuit simulation. We will conclude the chapter with several simple examples.

Detailed documentation of the API used to invoke **Xyce** as a coupled simulator and the methods that must be implemented for the interface class will be found in the next chapter.

3.2 General External Device

In order to couple a circuit simulated by **Xyce** to a specialized domain simulation run by another code, we must establish in the **Xyce** input netlist how the circuit connects to the external simulation's input ports. This is done using a special interface device, the "general external" device.

As noted earlier, devices in netlists are defined by lines whose first character is a letter, and the type of device is determined by that letter (R for resistor, D for diode, etc.); this syntax is dictated by SPICE compatibility requirements. Because most of the letters of the alphabet are already reserved for normal SPICE devices, this makes it difficult to extend the input syntax to new, esoteric device types. To deal with this limitation, **Xyce** has an extension mechanism where if the first character of the line is "Y," the characters immediately following the "Y" up to the next space are taken as the device type, and the next space-delimited word on the line is taken as the unique name; this is used by the **Xyce** team to add devices that do not fit into the categories of devices normally simulated in SPICE. The "general external" device is such a device, and is specified by the string "YGENEXT" at the beginning of a device line:

```
YGENEXT mydevice1 1 2
```

This netlist line defines a two-terminal general external device whose complete name is "YGENEXT!MYDEVICE1" (the concatenation of the device type and device name). It is connected betweens nodes 1 and 2.

General external devices can have any number (up to a maximum of 1000) of external connecting nodes, but due to a limitation in the **Xyce** parser an additional netlist quirk is required for **Xyce** to recognize the line if it has more than two nodes. This quirk is that the normally-optional model name must be given, even though the general external device does not take any model parameters:

```
YGENEXT myotherdevice 1 2 3 4 dummymodel
.model dummymodel genext
```

As a simple example, a netlist fragment representing the schematic in figure 3.1 that connects a DC voltage source to a two-terminal external device through a 1K Ohm resistor would be:

```
* Example netlist
V1 1 0 5V
R1 1 2 1000
YGENEXT mydevice1 2 0
```

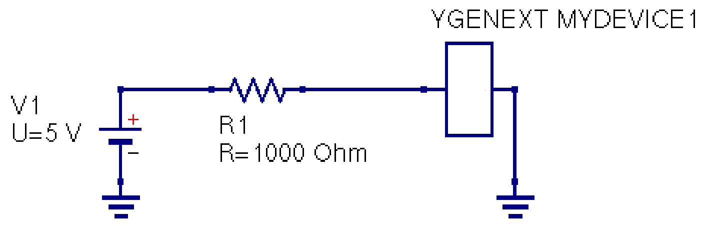

Figure 3.1. Example circuit with external device.

This fragment is not enough to run a simulation as it lacks any analysis or output directives, but it shows how the circuit layout is translated into netlist lines, and how the coupling device appears simply as another device in the layout.

Once defined in the netlist in this way, **Xyce** will be able to associate an interface object with the device, and call the interface class' methods when device evaluations are required. What remains is to define the interface object and associate it with this device before running the circuit.

3.2.1 YGENEXT lead currents

Each "YGENEXT" device has some number of external nodes (as defined in the netlist). Such a device interfaces with the rest of the circuit by drawing current from or sinking current into these connecting nodes. The device itself computes these currents from solution variables, but sometimes users wish to view this information as part of simulation results.

As with many other of **Xyce**'s internal device types, it is possible to print the values of these currents on **Xyce** `.PRINT` lines using the "lead current" notation. The syntax is the letter "I" followed by a number representing which external node of the device to query, followed by the name of the device in parentheses. The numbers refer to the position of the terminal in question, not the node name.

Thus, a three terminal YGENEXT device connected to nodes A, B, and C may be run and the currents flowing into the device (positive) or out of the device (negative) at each of its terminals may be printed using the netlist snippet below.

```
YGENEXT mydevice1 A B C dummymodel
.model dummymodel genext
.print tran I1(YGENEXT!mydevice1) I2(YGENEXT!mydevice1) I3(YGENEXT!mydevice1)
```

Positive values of lead currents denote current flowing *into* the device through the given terminal, and negative values denote current flowing *out of* the device at that terminal.

Lead currents of these forms are also legal output fields for the external output capability and may be used as strings in the `requestedOutputs` method, described below(4.3.1).

3.3 Interface Class

All communication between an external simulation code and **Xyce** during computation of **Xyce**'s circuit solution is handled through an interface class.

In order to couple an external code to **Xyce**, that application code must define a class that derives from the `Xyce::Device::VectorComputeInterface` abstract interface class, and must reimplement one or more of the methods of that interface. This is done by including the header file "N_DEV_VectorComputeInterface.h" in the application code and defining a new class that inherits from this abstract interface class. This header file is installed in the "include" directory that is produced when one installs **Xyce** from source code using the "make install" target.

The methods of the interface class must provide to **Xyce** the contributions to the DAE vectors and their derivative matrices related to the device emulated by the coupled simulator.

The interface object is then associated with a named "YGENEXT" device in the netlist using the GenCouplingSimulator class' `setVectorLoader` method.

There are only three functions defined in the interface that are called by **Xyce**. One is for time domain evaluation, and the other two for frequency domain. The application code may, of course, always define additional methods in the derived class for its own use.

3.3.1 Time domain coupling — computeXyceVectors

The only method of the interface that absolutely must be implemented is the `computeXyceVectors` method; `computeXyceVectors` is declared as a pure virtual function, and one cannot instantiate an object of any class inheriting from this interface without implementing it. The documentation of this function is provided in the API reference in section 4.2.1.

This method is the one called to load the devices' vectors and matrices during all time domain analyses, in small-signal AC analysis, and in noise analysis. This function is passed the current simulation time and a Standard Template Library (STL) double-precision vector containing all of the solution vector values for variables associated with the device. This vector first contains values of external nodes in the order that they appear on the netlist line, followed by values of internal variables. The function is also given references to STL double-precision vectors into which the application should store contributions to $\vec{F}$, $\vec{Q}$, $\vec{B}$, and references to two-dimensional vectors into which it should store the derivatives of $\vec{F}$ and $\vec{Q}$ with respect to solution variables. It is the responsibility of this function to set the sizes of these vectors and matrices appropriately, then fill them in and return "true."

If `computeXyceVectors` sets the size of one or more vectors to zero, then it is taken to mean that the device contributes nothing to that vector; it is not necessary to return a vector full of zeros if the device isn't contributing anything.

The vectors and matrices passed to `computeXyceVectors` are not the global DAE vectors and matrices, they are temporary arrays used for the device to load only its own contributions. Summing of these contributions into the global vectors and matrices is handled by higher level code internal to **Xyce**.

Once an interface object is associated with a general external device via a `setVectorLoader` call, the general external device will call this `computeXyceVectors` method every time it is called upon to provide DAE vector contributions, and it will simply copy the data it receives from the method call into **Xyce**'s internal data structures.

As noted in the previous chapter, **Xyce** calls upon every device to load the DAE vectors on every Newton step of every nonlinear solve, and the general external device will call `computeXyceVectors` every time it is asked to load the DAE vectors. Thus, it is essential to implement the function in as efficient a manner as possible — one would not want `computeXyceVectors` to trigger a full mesh solve of a PDE problem on every call, for example. This does mean that this coupling method is appropriate mostly for external problems that are either already linear, can easily be linearized over short time scales, or for which a simplified surrogate for the full mesh solve can be derived that is valid over short time scales.

3.3.2 Frequency domain coupling — computeXyceFDVectors and haveFDLoads

If performing Harmonic Balance simulation in **Xyce** and coupling **Xyce** to a frequency domain simulator, one must implement the `computeXyceFDVectors` method as well. The function arguments will be the current frequency value and the solution vector elements for the current solution guess at this frequency, as well as references to STL containers for $\vec{F}$ and $\vec{B}$, and the derivative of $\vec{F}$ with respect to the solution variables . The function should fill in these vectors from the results of its simulation at this frequency. The $\vec{Q}$ vector is not used in frequency domain simulation.

If doing a frequency domain simulation in the external code and there is no way to implement the time-domain `computeXyceVectors` method in a meaningful way, then one should implement `computeXyceVectors` so it simply returns false. Returning false will assure that **Xyce** will exit with a meaningful error message if any attempt is made to perform a time-domain simulation in **Xyce** while coupled.

Finally, if implementing `computeXyceFDVectors` for frequency domain coupling, one must also reimplement the method `haveFDLoads` in the interface class to return true. The default implementation returns false. **Xyce** calls this function to determine whether or not to use frequency domain loads from the device. If frequency domain loads are not available (i.e., if `haveFDLoads` returns false) and harmonic balance analysis is requested, **Xyce** will instead use the time-domain load functions at a series of time sample points, then Fourier transform the results to obtain the frequency domain contributions.

Frequency domain limitations

As of **Xyce** release 6.10, no provision has been made in **Xyce**'s harmonic balance analysis code to allow nonlinear frequency domain models. As a result of this design decision, the external code's `computeXyceFDVectors` method will only be called once for each frequency, and only the contents of the $\frac{d\vec{F}}{dX}$ matrix and the $\vec{B}$ vector it returns are really used — when $\vec{F}$ is needed, it is obtained by multiplying the derivative matrix by the solution vector. All that is necessary is that `computeXyceFDVectors` resize the $\vec{F}$ vector correctly — the values are never used.

This does mean that if the frequency domain problem in the external code is truly nonlinear, then this coupling method is not yet suitable to couple **Xyce** to the external simulator.

3.3.3 Examples

Figure 5.1 demonstrates an implementation of a simple linear resistor using the vector compute interface. It demonstrates correct sizing and loading of the vectors and matrices needed. An object of this class could be instantiated and associated with a two-terminal “YGENEXT” device in a circuit and would behave precisely as the internal resistor device of Xyce. Note also that this class has a `setResistance` method that is not part of the coupling API. This method is never called by **Xyce** itself, but may be used to change the behavior of the class from outside of **Xyce**, as one might need to do after processing instance parameters for an associated “YGENEXT” device.

Figures 5.2 and 5.3 demonstrate the implementation of a linear capacitor for both time- and frequency-domain simulation. Once associated with a “YGENEXT” device, any analysis other than harmonic balance will invoke its `computeXyceVectors` method, and harmonic balance will invoke its `computeXyceFDVectors` method.

Finally, figures 5.4, 5.5, 5.6, and 5.7 demonstrate the implementation of a two-terminal serial RLC network as illustrated by the circuit diagram in figure 2.1. This example demonstrates a device with two external variables (N1 and N2), use of two internal variables (the internal node NI between the inductor and capacitor, and the branch current flowing through the resistor and capacitor), a sparse Jacobian stamp, and the use of additional class members not required by the minimum abstract interface. In this simple example we have simply created a public data member for the Jacobian stamp that is initialized in the constructor (figure 5.4), which can later be accessed directly in order to be passed to the `setJacStamp` method of the general coupling simulator class. It also sets up static variables to use as indexes into the vectors and matrices for clarity (figure 5.7). The actual `computeXyceVectors` function is split between figures 5.5 and 5.6. This example may be found with detailed comments in the file src/test/GenExtTestHarnesses/testGenCoup.C of the **Xyce** source distribution. The application code must use the `setNumInternalVars` call to allocate two internal variables to the device, and `setJacStamp` to use the sparse jacobian stamp prior to calling the `setVectorLoader` method to associate the interface object with a specific “YGENEXT” device.

None of these simple implementations perform any sort of error checking on the sizes of the solution vectors passed to them; the implementations simply assume they have always been passed the correct number of solution variables for a two-terminal device. Neither have we yet demonstrated how to allow retrieval of device parameters from a netlist, which would be necessary for obtaining appropriate values to pass into their constructors. This topic will be discussed in section 4.1.3.

3.4 External Output Class

The general external simulator provides a `getSolution` method that allows the external code to obtain the converged solution values for the variables associated with specific general external devices in the circuit. This is often sufficient, but it can be useful for external simulators to have access to more information about the circuit than can be provided in that manner.

Xyce netlists may contain “.PRINT” statements to cause **Xyce** to emit solution information to a file. It is always possible for an external code to read these files if needed, but beginning with **Xyce** release 6.9, a new facility was added so that external codes could have all the information available to “.PRINT” lines delivered directly to function calls of their own design. This is known as the “external outputter” capability.

As with the general external device, external outputters work by having the external code provide an object that implements an interface class — the methods of these objects are called by the **Xyce** output system instead of writing the data to files, and the parameters of the output are determined by calls to query methods of the object instead of by parsing a `.PRINT` line in the netlist.

Access to the external outputter mechanism requires including the **Xyce** header file “N_IO_ExtOutInterface.h” in the application code. This provides the declaration of the `Xyce::IO::ExternalOutputInterface` abstract interface class. As with the vector compute interface, the user must define a new class that inherits from the abstract interface and implement its methods.

The application code will then call the `addOutputInterface` method of the GenCouplingSimulator class to add the interface object to a list of outputters kept by Xyce. The external outputter is then used just as if it had been created by a `.PRINT` line in the netlist — but instead of outputting to a file or to the screen, methods of the object are called with the data that would be output. The application code can then take any action necessary to handle this data, such as storing it in a database, plotting it, or performing analysis on it.

The methods of the `Xyce::IO::ExternalOutputInterface` class replace various components of a `.PRINT` line of **Xyce**. They include query methods, reporting methods, and output methods.

Query methods of this class are called by **Xyce** to determine what analysis type (DC, AC, Transient, etc.) should invoke the outputter, the list of requested output items, and a unique name for the outputter (generally used only for error reporting). **Xyce** will call these methods in order to set up its output stage at the beginning of a run.

Reporting methods are called after initial processing, such as to report parse status of the requested output values, and to report the actual names of output values that will be returned (which may differ from the list originally requested either because **Xyce** has augmented the list to include a TIME or FREQUENCY column, because wild cards have been expanded, or because the names have been canonicalized in some manner).

Output methods of the class are called each time **Xyce** would normally output data for `.PRINT` lines. These methods are passed vectors of doubles or complex numbers (as appropriate to the analysis type) for the requested output quantities.

3.4.1 Examples

A minimal example of an external output interface object is presented in figure 3.2. This interface object requests only the wildcard output `V(*)` (all known voltage nodes) for transient runs, and outputs all data directly to the standard output stream. Its reporting method `reportParseStatus` does no error checking, but merely reports to the standard error stream the parse status of all requested strings. Its output method `outputFieldNames` simply outputs the field names as a header; the wild card requested will be expanded by **Xyce** and the actual list of all voltage nodes to be printed will be returned to the application using this method. Its output method `outputReal` outputs each set of data on a single line.

The output interface of figure 3.2 would produce to the standard output stream a data file very similar to the one that would be created if the netlist had contained the line `.print tran V(*)`.

Though this simple example shows a class that takes no arguments in its constructor, and implements only the minimum subset of the API, one could always create a class that is more configurable and which contains additional methods to control its operation from the external code — **Xyce** only calls methods of this class that are part of the published API, and the internal details of implementation of this class are entirely up to the user, so long as it derives from the interface and implements all required methods.

A more extensive example that includes a configurable, but still basic, external output interface class is provided in the file `testGenCoup.C` in the directory `src/test/GenExtTestHarnesses` of the **Xyce** source tree.

```
#include <N_IO_ExtOutInterface.h>
class ioExampleInterface
  : public Xyce::IO::ExternalOutputInterface
{
public:
 ioExampleInterface() {};
 Xyce::IO::OutputType::OutputType getOutputType()
  {
    return Xyce::IO::OutputType::TRAN;
  };
  void requestedOutputs(std::vector<std::string> &outputVars)
  {
    outputVars.resize(1);
    outputVars[0] = "V(*)";
  };
  void reportParseStatus(std::vector<bool> & statusVec)
  {
    for (int i=0;i<statusVec.size();i++)
    {
      std::cout << "Requested output " << i << " is "
           <<  ( (statusVec[i])? " parseable" : " unparseable")
           << std::endl;
    }
  };
  void outputFieldNames(std::vector<std::string> & outputNames)
  {
    for(int i=0; i<outputNames.size(); i++)
    {
       std::cout << outputNames[i] << "   ";
    }
    std::cout << std::endl;
  };
  void outputReal(std::vector<double> & outputData)
  {
    for (int i=0;i<outputData.size();i++)
    {
       std::cout << outputData[i] << "   ";
    }
    std::cout << std::endl;
  };
  void finishOutput()
  {
    std::cout << "End of Xyce(TM) Simulation" << std::endl;
  };
};
```

Figure 3.2. Trivial output interface

3.5 Simulator class

3.5.1 Setup and invocation

Use of the general coupling mechanism requires that the application code instantiate an object of type `Xyce::Circuit::GenCouplingSimulator`, which serves as the application's control object for **Xyce** (see section 4.1.1). An example of this instantiation is shown in figure 4.1.

After instantiation of the `GenCouplingSimulator` object, the application code calls its methods to read a netlist, set up any general external devices that appear in the netlist, register output interface objects, and finalize initialization.

Reading of the netlist and initialization of internal data structures is accomplished by calling the `intializeEarly` method (section 4.1.2) This method takes arguments similar to the standard "main" function of C or C++ programs, so the caller must construct a C-style array of strings representing a command line; the possible command line options are the same as those that would be used for invoking a standard **Xyce** binary, and are documented in the Xyce Reference Guide [5]. An example of this call is shown in figure 4.2. That example demonstrates calling `initializeEarly` as if **Xyce** were invoked with no arguments other than a netlist name.

After the call to `initializeEarly`, the application code can query the GenCouplingSimulator object to determine if any general external ("YGENEXT") devices are present in the netlist. This is done via the `getDeviceNames` method(4.1.3), an example of which is shown in figure 4.3. If present, these devices should be set up using the `setNumInternalVars` (4.1.3), `setJacStamp`(4.1.3), and `setVectorLoader`(4.1.3) methods.

If making use of the external outputter capability, it is also necessary to instantiate any external output interface objects and pass them to the `addOutputInterface`(4.1.3) call in between the `initializeEarly` and `initializeLate` calls.

Once all general external devices and external outputs have been properly set up and `initializeLate`(4.1.2) is called, **Xyce** is ready to begin the coupled simulation. There are two basic mechanisms for running the simulation.

In the first, using `runSimulation`(4.1.4), **Xyce** is instructed to run the entire simulation as specified in the netlist, making necessary calls back to the vector compute interfaces along the way. In this case, control is not returned to the external code except through the interface objects until the simulation is complete. To run in this manner, **Xyce** is completely in control of time stepping (for transient simulation), and it is necessary to write the vector compute interface in such a manner that it can be called for any time value between 0 and the final requested time in the netlist.

The second mechanism is for the external code (which may have its own time stepping algorithm) to call **Xyce** via the `simulateUntil`(4.1.4) call. **Xyce** will simulate only until the requested time value, then return control to the calling program. In this way, the caller can simulate the larger problem at specific time intervals, then call upon Xyce to advance its simulation time only for that short interval.

3.5.2 Compiling and linking with the Xyce library

When **Xyce** is built and installed, it will have installed a **Xyce** binary and all of its libraries and include files into the bin, lib, and include subdirectories (respectively) of the directory given to the `--prefix` option of `configure` when it was run. We will refer to this top-level directory as `$prefix`, and assume that **Xyce** has been built according to the guidance in the Xyce Building Guide[6].

To build and link an application that uses the general coupling interface of this report, one requires not only an installed version of **Xyce** and its libraries, but also the installation of Trilinos that was used to build **Xyce**. We will, for the purposes of this section, refer to the top level installation directory of Trilinos as `$archdir`, and will assume that Trilinos has also been built according to the guidance in the Xyce Building Guide.

Xyce depends on many libraries, including not only its own as installed in `$prefix/lib`, but also individual package libraries of Trilinos and third party libraries provided by the operating system, all of which must be linked into any application code that uses **Xyce** libraries. Most of these dependencies have been captured well in the ".la" files that libtool created and installed while building and installing **Xyce**, and so the simplest path to linking a code with **Xyce** is to use libtool and leverage this information.

Compiling with libtool wrappers

Libtool may be invoked both in compile mode and link mode, and this may be the most straightforward way to build a simple application code against **Xyce**. The command in figure 3.3 will compile a test program called "mySimCode.C" using the same Trilinos and Xyce headers that were installed for building **Xyce** itself. This example assumes no additional "-I" directories are needed to find header files directly referenced in the ".C" file, and assumes the C++ compiler's name on the system is "c++".

```
libtool --tag=CXX --mode=compile c++  \
  -I$prefix/include -O3 -I$archdir/include/ \
  -c mySimCode.C -o mySimCode.o
```

Figure 3.3. Compiling a test code with libtool

Figure 3.4 shows an example of how libtool may be invoked to link this example program against the same libraries against which **Xyce** itself was linked, making use of the ".la" files that were installed by **Xyce**'s "make install" operation. This example is appropriate for an open source build of **Xyce**, which includes only the three libraries "libxyce.la", "libADMS.la", and "libNeuronModels.la". Additional libraries are required for a full internal Sandia build of **Xyce**, and are easily identified simply by looking at the contents of the $prefix/lib directory. Because we are using the ".la" files that libtool produced during the build of **Xyce**, all of the additional libraries and linker options (notably any "-L" flags) needed are automatically included by this command line because those dependencies are already encoded in the libxyce.la file.

```
libtool --tag=CXX --mode=link c++  \
-L$prefix/lib -o mySimCode  mySimCode.lo \
$prefix/lib/libxyce.la  $prefix/lib/libADMS.la \
$prefix/lib/libNeuronModels.la
```

Figure 3.4. Linking a test code with libtool

Compiling without libtool wrappers

If building an application code that uses its own build system, and which cannot make use of libtool, one must invoke the C++ compiler and linker directly, and specify all dependent libraries explicitly. We cannot provide specific guidance here for all such build systems. However, the complete list of libraries and linker flags that must be linked in may be found in the "dependency_libs" variable in the "libxyce.la" file. Calling these for the sake of example "$deplibs" (as if the entire contents of the "dependency_libs" variable setting in the ".la" file were copied into an environment variable), one may compile the application code with the command line in figure 3.5, and link it with the command line in figure 3.6.

```
c++  \
 -I$prefix/include -O3 -I$archdir/include/ \
 -c mySimCode.C -o mySimCode.o
```

Figure 3.5. Compiling a test code without libtool

```
c++  \
-L$prefix/lib -o mySimCode  mySimCode.lo \
-lxyce  -lADMS \
-lNeuronModels $deplibs
```

Figure 3.6. Linking a test code without libtool

3.6 Limitations

The design of the general coupling interface is flexible and general, but not applicable to all manner of circuit coupling that might be envisioned.

Xyce calls the `computeXyceVectors` method of the interface classes for every Newton iteration of every time step of a simulation. If coupling to a mesh-based solver, it would be highly inefficient to perform a full mesh solve every time `computeXyceVectors` is called. For such solvers, it is generally the case that **Xyce** is providing boundary conditions on interface surfaces of the mesh, and this coupling mechanism would be best suited to such problems that are linear in their response to these boundary conditions, or which can be linearized over small time scales.

Xyce performs its own adaptive time stepping to solve the circuit problem in transient. Even when using the `simulateUntil` method to limit how much simulation time **Xyce** is permitted to advance, **Xyce** may take many small timesteps under its own control, making multiple calls back to the external code through the Vector Compute Interface objects. The application code must therefore pay attention to the value of time passed to it through the `computeXyceVectors` call. **Xyce** may also reject some of its own timesteps based on its computation of local truncation error; when it does so it will attempt a new solution at an earlier time than the one it rejected. The application code must therefore be prepared to recompute quantities when the current simulation time passed to it through `computeXyceVectors` has jumped backward.

If carefully written, it would be possible to couple a **Xyce** circuit to a full mesh solver that recomputes the entire mesh solution on every call to `computeXyceVectors`, and the results would be physically correct and consistent. This would be highly inefficient, however, and such simulation needs would likely be better served with a different coupling mechanism.

Coupling of **Xyce** to frequency domain simulation codes is limited to harmonic balance analysis at this time; **Xyce**'s AC analysis does not yet support direct frequency domain loads, and still depends only on the time-domain load functions. Furthermore, while **Xyce**'s harmonic balance analysis can work with **Xyce**'s own nonlinear circuit devices, it does so by evaluating those devices in time domain at a series of time sample points and transforming the vectors they produce to obtain the frequency domain response. As of this writing, the harmonic balance analysis is able to work with devices that can provide direct frequency domain loading, but it does so by assuming these devices are linear functions of the solution variables in the frequency domain. As a consequence, **Xyce** only calls the `computeXyceFDVectors` method of the interface class once per frequency, and only uses the $\frac{d\vec{F}}{dX}$ matrix and $\vec{B}$ vectors that it computes. It is not yet possible to couple to a truly nonlinear frequency domain model.

Xyce's robust and adaptive time integrator makes it attractive for more general purpose DAE time integration, e.g., by augmenting the internal variables and equations. However, it should be noted that it is not currently possible to specify non-zero initial conditions for the extra variables. It may be possible to work around this limitation using a variable transformation, and this limitation may be addressed in future versions of **Xyce**[1].

While it is not possible for external codes to give their internal nodes initial conditions, it would be possible to impose initial conditions on these nodes through the **Xyce** netlist using the `.IC` directive. Each internal node is known to **Xyce** by a name that consists of the YGENEXT device's name followed by the text "_internalnode_" followed by the index of the internal node (with zero being the first internal node). So, for example, if the netlist defines a device

```
YGENEXT RLC1 1 2
```

and this device is associated with an RLC vector compute interface as in figure 5.4 with two internal variables, one could set the initial condition on the first internal variable (the node between the inductor and capacitor) to 1.0 with:

```
.IC V(YGENEXT!RLC1_internalnode_0)=1.0
```

Note that unless the `UIC` or `NOOP` keyword is used on the `.TRAN` line, **Xyce** will attempt to solve for the DC steady state (operating point) using these initial conditions. If the initial condition is one that is inconsistent with a steady state there will be a simulation failure. `UIC` or `NOOP` causes the transient simulation to skip the operating point computation and proceed directly to transient simulation using the initial condition given (all zeros except where specified otherwise by `.IC` lines.

[1]The limitation that devices may not simply impose their own initial conditions is a consequence of the way that **Xyce** solves the nonlinear system for a solution update. It is possible to write device models so that they can do so, but this requires a mechanism known as "voltage limiting" as described in the **Xyce** mathematical formulation document [3]. The general external device code does not yet support this mechanism, and would have to be extended to include it in order to address this limitation. Use of this mechanism would then also require additional code to be implemented by the coupled application.

4. API Reference

This chapter contains detailed documentation for the simulator interface API and for each class and each method that can or must be implemented by an external simulator in order to make use of the general coupling mechanism.

4.1 General Coupling Simulator Class

Coupling to **Xyce** via the technique described in this paper requires that the external simulator instantiate an object of the type Xyce::Circuit::GenCouplingSimulator, which provides all the methods required to set up and run circuit simulations.

In order to instantiate this object, one must include the files "Xyce_config.h" and "N_CIR_GenCouplingSimulator.h" from the **Xyce** installation, and link the application code with **Xyce** libraries. The header files and libraries required are installed in the "include" and "lib" directories by the "make install" target when building Xyce from source.

This section contains detailed documentation of the methods of the GenCouplingSimulator class.

4.1.1 Constructor

```
Xyce::Circuit::GenCouplingSimulator(Xyce::Parallel::Machine comm=MPI_COMM_NULL)
```

The single optional argument to this constructor is the parallel communicator that should be used when running under MPI. If not specified, the **Xyce** code will run in serial. An example of instantiation of this object with default behavior is shown in figure 4.1.

```
#include <Xyce_config.h>
#include <N_CIR_GenCouplingSimulator.h>

  [...]
  // This creates a variable ``xyce'' of the
  // coupling simulator class.
  Xyce::Circuit::GenCouplingSimulator xyce;
  [...]
```

Figure 4.1. Instantiating Simulation Object

4.1.2 Initialization methods

After construction, the GenCouplingSimulator object must be initialized before a circuit simulation can proceed. The initialization is divided into early and late stages, between which one may run the query and setup methods to modify the behavior of general external devices present in the circuit netlist.

initializeEarly

```
Xyce::Circuit::Simulator::RunStatus initializeEarly(int argc, char **argv)
```

The initializeEarly method's two arguments mimic the function of the same arguments in a normal C or C++ main function: they are interpreted as representing the command line that invoked **Xyce**. `argc` is the number of strings present in the array of strings, `argv`.

The string `argv[0]` is taken to be the name of the program, and no use is made of it. Subsequent elements of the `argv` array are command line options as documented in chapter 3 of the **Xyce** Reference Guide[5]. The final argument string in this array should be the name of the netlist to be processed.

The `initializeEarly` method sets options as specified in `argv` and reads the given netlist. It will instantiate the devices present in the netlist, allocate all of the solvers and packages needed, and then stop processing, returning control to the caller. This allows external codes to then set internal properties of general external devices, such as assigning a vector compute object to them or setting their numbers of internal solution variables. Only after the external code has completed those setup calls are the final initialization steps performed by calling `initializeLate`.

The method returns a value from the Xyce::Circuit::Simulator::RunStatus enum, which has the values “Xyce::Circuit::Simulator::ERROR”, “Xyce::Circuit::Simulator::SUCCESS”, or “Xyce::Circuit::Simulator::DONE”.

“ERROR” signifies failure of the initialization, and the actual error condition will have been printed to **Xyce**’s standard error stream. Further calls to the GenCouplingSimulator object’s methods should not be made, as **Xyce** has effectively terminated with a fatal error when this value is returned.

“DONE” signifies that all processing is complete. This return value is used when the command line arguments include an argument that prevents **Xyce** from proceeding to full simulation, such as “-syntax”, “-count”, “-v”, “-norun” and so forth. If `initializeEarly` returns this value, **Xyce** has effectively exited successfully and further calls such as `initializeLate` should not be performed.

“SUCCESS” signifies that the initialization was successful, and the GenCouplingSimulator object is ready for futher calls.

initalizeLate

```
Xyce::Circuit::Simulator::RunStatus  initializeLate()
```

`initializeLate` is called after all query and setup methods are completed. It finalizes the analysis of topology, sets up the internal vector and matrix storage, initializes the output manager, and makes the object ready for simulation to take place. It returns the same run status values as `initializeEarly`, which have the same meaning.

An example of calling the initialize methods is found in figure 4.2

```
#include <Xyce_config.h>
#include <N_CIR_GenCouplingSimulator.h>


Xyce::Circuit::Simulator::RunStatus run_status;

Xyce::Circuit::GenCouplingSimulator xyce;


char * argv[2];
char progName[]="xyce";
char netlist[]="mynetlist.cir";

argv[0]=progName;
argv[1]=netlist;

// perform early initialization
run_status = xyce.initializeEarly(2,argv);
if (run_status==Xyce::Circuit::Simulator::ERROR) exit(1);
if (run_status==Xyce::Circuit::Simulator::DONE) exit(0);


// do more setup
[...]

// and finish initialization
run_status = xyce.initializeLate();
if (run_status==Xyce::Circuit::Simulator::ERROR) exit(1);
if (run_status==Xyce::Circuit::Simulator::DONE) exit(0);

[...]
```

Figure 4.2. Initializing Simulation Object

4.1.3 Query and setup methods

In between calling `initializeEarly` and `initializeLate`, one can call methods of the GenCouplingSimulator class that can influence the behavior of devices.

Recall that the devices used to couple external simulators are all of the “General External

Device” type, and so they will all appear in netlists with the word “YGENEXT” at the beginning of lines. The GenCouplingSimulator object has methods that allow the external code to query whether any such devices exist in the netlist and what their full names are, and then to manipulate these individually.

getDeviceNames

```
bool getDeviceNames(const std::string & modelGroupName,
                    std::vector<std::string> & deviceNames)
```

`getDeviceNames` takes a string containing a “model group” name, and returns a list of names for all devices in the netlist of that type. It is a general purpose method that can be given any valid model group name (“M” for MOSFETs, “Q” for BJTs, etc.) but for the purposes of this application note, we will only need to call it with “YGENEXT” as the model group name.

The method will return true if any devices of the given type are located, and their names will be in the returned STL string vector deviceNames. The method will return false if no such devices are found.

Devices of the “YGENEXT” type will all have names returned by this function that are the concatenation of the string “YGENEXT” and the next token on the netlist line, separated by an exclamation point (“!”). For the netlist line

```
YGENEXT MYDEVICE1 2 0
```

the array returned by `getDeviceNames("YGENEXT",[...])` would contain “YGENEXT!MYDEVICE1”.

Names returned by `getDeviceNames` may be used as the device name in all of the query and setup functions below.

```
#include <Xyce_config.h>
#include <N_CIR_GenCouplingSimulator.h>

[...]
// xyce object already declared as in 4.2
run_status = xyce.initializeEarly(argc,argv);

bool bsuccess;
std::vector<std::string> deviceNames;
bsuccess = xyce.getDeviceNames("YGENEXT", deviceNames);

if (bsuccess)
{
  // there are some, do something with them
  [...]
}
else
{
  // otherwise, there is nothing in this netlist
  // to connect to... handle appropriately,
  // maybe that means to abort here.
  [...]
}

run_status = xyce.initializeLate();
[...]
```

Figure 4.3. Querying device by type

Parameter access methods

Most device types in **Xyce** allow the user to specify parameters that control their function from the netlist line. The general external device is no exception. General external devices can have any number of double-precision, integer, boolean, or string parameters specified on their instance lines in the netlist, and the external simulator that is implementing the internals of these devices can query **Xyce** to obtain their values.

An example netlist snippet containing a “YGENEXT” device with parameters appears in figure 4.4. Double precision parameters are passed in the “DPARAMS” “vector composite” parameter, integers in “IPARAMS”, strings in “SPARAMS” and booleans in “BPARAMS”.

Each composite contains a “NAME” and a “VALUE” variable, and each of these variables contains a comma-separated list of elements as values. Each desired parameter must be given a name in the NAME variable and a value in the VALUE variable in the corresponding position.

The names and values of parameters given in the netlist are given no significance in **Xyce** — they are made available to the calling application code for its use via the `getDParams`, `getSParams`, `getIParams`, and `getBParams` methods.

The **Xyce** netlist parser puts strict limits on what can be used as string values in the “SPARAMS” composite. These strings cannot contain any sort of parentheses, brackets, or braces, and cannot contain commas or spaces. There is (as of version 6.9) currently no capability in **Xyce** to use quotation marks to escape the contents of the strings, as both single and double quotation marks are reserved for other purposes. This may change in a future version of **Xyce**.

```
*
* [...]

YGenExt rlc1 1a 0
+           DPARAMS={NAME=R,L,C VALUE=1K,1m,1p}
+           IPARAMS={NAME=num1,num2,num3 VALUE=1,2,3}
+           SPARAMS={NAME=string1,string2 VALUE=value1,value2}
+           BPARAMS={NAME=bool1,bool2 VALUE=true,false}

* [...] other elements follow [...]
```

Figure 4.4. Specifying Parameters to YGENEXT

```
#include <Xyce_config.h>
#include <N_CIR_GenCouplingSimulator.h>

[...]
// xyce object already declared as in 4.2
run_status = xyce.initializeEarly(argc,argv);

bool bsuccess;
std::vector<std::string> deviceNames;
bsuccess = xyce.getDeviceNames("YGENEXT", deviceNames);

if (bsuccess)
{
  // there are some, do something with them
  [...]
  // Get any netlist parameters specified for the devices
  for ( int i=0; i<deviceNames.size(); i++)
  {
    std::vector<std::string> doubleParamNames;
    std::vector<double> doubleParamValues;
    bsuccess = getDParams(deviceNames[i],
                          doubleParamNames,
                          doubleParamValues);
    // if bsuccess false, this should be fatal
    // otherwise, process names and values as needed

    std::vector<std::string> intParamNames;
    std::vector<int> intParamValues;
    bsuccess = getIParams(deviceNames[i],
                          intParamNames,
                          intParamValues);
    // if bsuccess false, this should be fatal
    // otherwise, process names and values as needed
```

Figure 4.5. Querying device parameters

```
        std::vector<std::string> boolParamNames;
        std::vector<bool> boolParamValues;
        bsuccess = getBParams(deviceNames[i],
                              boolParamNames,
                              boolParamValues);
        // if bsuccess false, this should be fatal
        // otherwise, process names and values as needed

        std::vector<std::string> stringParamNames;
        std::vector<std::string> stringParamValues;
        bsuccess = getSParams(deviceNames[i],
                              stringParamNames,
                              stringParamValues);
        // if bsuccess false, this should be fatal
        // otherwise, process names and values as needed

      }
      [...]
    }
    else
    {
      // otherwise, there is nothing in this netlist
      // to connect to... handle appropriately,
      // maybe that means to abort here.
      [...]
    }

    run_status = xyce.initializeLate();
    [...]
```

Figure 4.6. Querying device parameters (continued)

getDParams

```
  bool getDParams(const std::string & deviceName,
                  std::vector<std::string> &pNames,
                  std::vector<double> & pValues);
```

Given a device name (as returned by getDeviceNames), populate the `pNames` and `pValues` vectors with the names and values of parameters given in the DPARAMS composite.

As an example, if this function were called while processing the netlist of figure 4.4 as in figure 4.5, the names vector would be filled with the strings “R”, “L”, and “C”, and the values vector would have 1000, 0.001, and 1e-12.

The function returns “true” unless the device named cannot be found. The vectors will be empty if the device is found but no double-type parameters have been found on the netlist line.

getIParams

```
bool getIParams(const std::string & deviceName,
                std::vector<std::string> &pNames,
                std::vector<int> & pValues);
```

Given a device name (as returned by getDeviceNames), populate the `pNames` and `pValues` vectors with the names and values of parameters given in the IPARAMS composite.

As an example, if this function were called while processing the netlist of figure 4.4 as in figure 4.5, the names vector would be filled with the strings “num1”, “num2”, and “num3”, and the values vector would have 1, 2, and 3.

The function returns “true” unless the device named cannot be found. The vectors will be empty if the device is found but no int-type parameters have been found on the netlist line.

getBParams

```
bool getBParams(const std::string & deviceName,
                std::vector<std::string> &pNames,
                std::vector<bool> & pValues);
```

Given a device name (as returned by getDeviceNames), populate the `pNames` and `pValues` vectors with the names and values of parameters given in the BPARAMS composite.

As an example, if this function were called while processing the netlist of figure 4.4 as in figure 4.5, the names vector would be filled with the strings “bool1, “bool2”, and the values vector would have true and false.

The function returns “true” unless the device named cannot be found. The vectors will be empty if the device is found but no boolean-type parameters have been found on the netlist line.

getSParams

```
bool getSParams(const std::string & deviceName,
                    std::vector<std::string> &pNames,
                    std::vector<std::string> & pValues);
```

Given a device name (as returned by getDeviceNames), populate the `pNames` and `pValues` vectors with the names and values of parameters given in the SPARAMS composite.

As an example, if this function were called while processing the netlist of figure 4.4 as in figure 4.5, the names vector would be filled with the strings “string1” and “string2”, and the values vector would have “value1” and “value2”.

The function returns “true” unless the device named cannot be found. The vectors will be empty if the device is found but no string-type parameters have been found on the netlist line.

setVectorLoader

```
bool setVectorLoader(const std::string & deviceName,
                     Xyce::Device::VectorComputeInterface * vciPtr);
```

Every “YGENEXT” device in the circuit must have a vector loader associated with it. Application codes define classes that inherit from the Xyce::Device::VectorComputeInterface, and then implement the methods of that class for **Xyce** to call as needed.

The `setVectorLoader` function associates a pointer to a VectorComputeInterface object (defined and instantiated by the external application) with the “YGENEXT” device assocated with the name `deviceName` (as returned by `getDeviceNames`). The methods of this object will be called every time that **Xyce** needs the device to compute its DAE vector and matrix contributions.

Xyce stores the pointer provided in the device object associated with the named device, and calls the class’ methods through that pointer. It is essential that the vector compute interface object not be deleted until **Xyce** processing is complete.

This method should only be called in between early and late initialization. Attempting to change the vector loader associated with a device during the course of a run may lead to undefined and unpredictable behaviors.

This function always returns true unless the named device cannot be found. A false return value indicates an error condition that must be handled by the application code.

An example of using this method is shown in context in figure 4.7. Note that in this example we have taken care to declare the vector compute object so that it remains in scope and is not deleted while **Xyce** is active.

```
#include <Xyce_config.h>
#include <N_CIR_GenCouplingSimulator.h>

  [...]
  // define the resistorVectorCompute class as shown
  // in figure 5.1
  [...]
  // instantiate simulator object as in figure 4.2
  [...]

run_status = xyce.initializeEarly(argc,argv);

bool bsuccess;
std::vector<std::string> deviceNames;
bsuccess = xyce.getDeviceNames("YGENEXT", deviceNames);

// Intentionally declared outside of if block
// so it will not be deleted while pointer is stored!
resistorVectorCompute rvc(1.0);

if (bsuccess)
{
  // there are YGENEXTs, do something with them
  // For the purposes of this example, just connect first device
  // to a resistor object as shown in figure 5.1,
  rvc.setResistance(1000.0);
  bsuccess = xyce.setVectorLoader(deviceNames[0],&rvc);
  [...]
  // perform other set-up
  [...]
}
else
{
  // otherwise, there are no YGENEXTs in this netlist
  [...]
}

run_status = xyce.initializeLate();
[...]
// perform simulations
[...]
```

Figure 4.7. Setting a vector loader

setNumInternalVars

```
bool setNumInternalVars(const std::string & deviceName,const int numInt);
```

By default, general external ("YGENEXT") devices have no internal variables, and only as many external variables as have been specified as nodes on the netlist line.

If the formulation of the external device model requires additional solution variables that **Xyce** needs to solve for, **Xyce** must allocate additional space in the vectors and matrices it uses during solution, and the device must load additional elements of the DAE vectors and matrices. The application code must therefore specify to **Xyce** how many extra variables should be associated with each device.

The `setNumInternalVars` function allows an external code to indicate that a named device has a specified number of additional internal variables. The `deviceName` argument is the name of the device as returned by `getDeviceNames`.

If this function is never called for a device, then the device will be taken to have no internal variables.

Since the number of internal variables must be known by the time of late initialization (at which time various vectors are allocated and circuit topology finalized), if `setNumInternalVars` is called at all for any given device, it *MUST* be called in between the `initializeEarly` and `initializeLate` calls.

getNumVars

```
int getNumVars(const std::string & deviceName);
```

The `getNumVars` returns the total number of solution variables (both external and internal) associated with the named device.

getNumExtVars

```
int getNumExtVars(const std::string & deviceName);
```

The `getNumExtVars` returns the total number of external nodes (i.e., those listed explicitly on the netlist line) associated with the named device.

getSolution

```
bool getSolution(const std::string & deviceName, std::vector<double> & sV);
```

Causes **Xyce** to fill the `sV` vector with the values of all solution variables associated with the named device. If passed an empty vector for `sV`, the method resizes the vector to have the length equal to the total of external and internal variables for the device, but otherwise assumes the vector is already the correct length (the length that would be returned by `getNumVars`). External variables are the first in the vector, followed by internal variables.

This method is primarily used after simulateUntil has returned, to get the most recent converged solution after completion of a partial transient run.

setJacStamp

```
bool setJacStamp(const std::string & deviceName,
                 std::vector< std::vector<int> > &jS);
```

Xyce uses a sparse matrix representation to store the Jacobian matrices of the problem. Each device is responsible for providing an accessor method that returns a data structure known as a “Jacobian Stamp” to notify the topology packge of which non-zero elements of the Jacobian the device needs to load.

By default, the general external device informs the topology package that it has a dense Jacobian stamp — every equation of the model depends on all variables of the model. If this is not true and this behavior is not overriden, the device will be responsible for loading the full Jacobian even if many elements are actually zero.

Vector compute interfaces actually return a full, square Jacobian in a separate data structure, and need to load zeros in the appropriate locations if needed, so this default will always work. However, for large problems where the pattern of non-zero elements in the Jacobian is very sparse, it can be inefficient for **Xyce** to allocate Jacobian elements that will always be zero anyway.

The `setJacStamp` method exists to allow the developer to override the default and provide the correct pattern of non-zero elements. Its first argument is the name (as returned by `getDeviceNames`) of the device for which the Jacobian stamp is being defined, and the second argument is the Jacobian stamp to use.

The Jacobian stamp is an array (STL vector of STL vectors) of integers identifying the nonzero elements of the matrix. The first index of the array is the row of the Jacobian, and the row has size equal to the number of non-zero elements of the matrix for that row. The values of the elements of the row vector are the indices of the variables in the row for which there are non-zero entries in the matrix. The Jacobian stamp need not be square.

For example, consider a device with three variables, whose Jacobian matrix happens to have the pattern of entries:

$$\begin{bmatrix} xx & 0 & xz \\ yx & yy & 0 \\ 0 & zy & zz \end{bmatrix}.$$

such that the device will always have zeros where indicated. This device could benefit from specifying the Jacobian stamp instead of relying on the default implementation.

Using the normal C++ 0-based indexing scheme, The first row (row 0) depends only on variables 0 and 2, the second (row 1) only on 0 and 1, and the third (row 2) only on 1 and 2. The Jacobian stamp would therefore be set as:

```
std::vector< std::vector<int> > jacStamp

jacStamp[0].resize(2)
jacStamp[0][0] = 0;
jacStamp[0][1] = 2;

jacStamp[1].resize(2)
jacStamp[1][0] = 0;
jacStamp[1][1] = 1;

jacStamp[2].resize(2)
jacStamp[2][0] = 1;
jacStamp[2][1] = 2;
```

If a non-default Jacobian stamp is needed for a device, `setJacStamp` must be called prior to calling `setVectorLoader` for that device. If the jacobian stamp is not set when `setVectorLoader` is called, then `setVectorLoader` will create a dense jacobian stamp for the device itself.

addOutputInterface

```
bool addOutputInterface(Xyce::IO::ExternalOutputInterface * extIntPtr);
```

Add an external output interface object pointed to by `extIntPtr` to the list of external output objects.

There can be multiple outputters defined simultaneously. Each `.PRINT` line in a netlist creates an outputter that formats data and writes it to a file in a manner dictated by options on the print line. Similarly, the output manager will create a new outputter for each interface object added by `addOutputInterface`, add it to the list of known outputters, and associate the interface object with that outputter. The methods of the interface object determine how the data is processed.

Each analysis type within **Xyce** causes all outputters appropriate to that analysis to become active. The type of analysis that applies to an external output interface is determined by calling the `getOutputType` method of the interface object.

An example of using this method to make use of a user-defined output interface is found in figure 4.8. An example of defining a suitable output interface class is found above in figure 3.2.

Xyce stores the pointer provided for the duration of the run, and calls the class' methods through that pointer. It is essential that the output interface object not be deleted until **Xyce** processing is complete. Note that the example in figure 4.8 takes care to declare the object so that it is not possible for it to go out of scope and be deleted while running **Xyce** simulations.

```
#include <Xyce_config.h>
#include <N_CIR_GenCouplingSimulator.h>
#include <N_IO_ExtOutInterface.h>

  // define a new derived outputter class
  class ioExampleInterface
  : public Xyce::IO::ExternalOutputInterface
  {
  // see figure 3.2
  [...]
  };
  // instantiate simulator object as in figure 4.2
  [...]

run_status = xyce.initializeEarly(argc,argv);

// Instantiate the outputter
ioExampleInterface * myIOInterface;
myIOInterface = new ioExampleInterface();

// and tell Xyce to use it
xyce.addOutputInterface(myIOInterface);

// perform additional set-up
[...]

run_status = xyce.initializeLate();
[...]
// run simulations
[...]
```

Figure 4.8. Adding an external outputter

4.1.4 Execution functions

runSimulation

```
Xyce::Circuit::Simulator::RunStatus runSimulation();
```

Causes **Xyce** to take control and run the entire simulation specified in the netlist to completion. All general external devices will call back to the external simulator via their vector compute interfaces, but otherwise the entire simulation is controlled by **Xyce**'s time stepping (or other analysis) algorithm.

The method returns a value from the Xyce::Circuit::Simulator::RunStatus enum, which has the values “Xyce::Circuit::Simulator::ERROR”, “Xyce::Circuit::Simulator::SUCCESS”, or “Xyce::Circuit::Simulator::DONE” as described in the `initializeEarly` section.

This function must be called *AFTER* the call to `initializeLate`.

simulateUntil

```
bool simulateUntil(double requestedUntilTime,
                   double& completedUntilTime);
```

`simulateUntil` causes **Xyce** to perform a limited transient simulation not to exceed the simulation time specified in `requestedUntilTime`. Upon return, `completedUntilTime` will contain the actual time that **Xyce** reached, which will be less than or equal to `requestedUntilTime` — either because the netlist specified a final time earlier than `requestedUntilTime`, or because there was a fatal convergence error.

Each call to `simulateUntil` after the first resumes the current simulation from where the last call left off.

The method returns true if the simulation completed successfully, either by reaching `requestedUntilTime` or the final time specified in the netlist, whichever is earlier. It returns false if the run was unsuccessful.

If `simulateUntil` returns true and `completedUntilTime` is less than `requestedUntilTime`, **Xyce** has completed its work and further calls to `simulateUntil` will do nothing.

This function must be called *AFTER* the call to `initializeLate`.

finalize

```
Xyce::Circuit::SimulatorRunStatus finalize();
```

This method causes **Xyce** to close all output files after a run is complete and emit timing information. It should be called after the **Xyce** simulation is complete, either by returning from `runSimulation` or by reaching the final netlist time after repeated `simulateUntil` calls.

4.2 Vector Compute Interface Class

Each “YGENEXT” device in the circuit must have associated with it an object that derives from the `Xyce::Device::VectorComputeInterface` class, as defined in the file “N_DEV_VectorComputeInterface.h”. This object has two primary methods, at least one of which must be implemented by the user. **Xyce** will call these methods every time it loads the DAE vectors and matrices for the circuit, and the information they return will be used to load the DAE elements for the associated device.

The `Xyce::Device::VectorComputeInterface` class is an abstract interface class, meaning it cannot be instantiated directly. The user must define a new class that inherits from this interface, and then implement the required methods. A trivial example of such a derived class implementing a simple linear resistor is shown in figure 5.1.

Once allocated and initialized, a pointer to the interface object is associated with a “YGENEXT” device using the `setVectorLoader` method of the `GenCouplingSimulator` class described in section 4.1.

4.2.1 computeXyceVectors

```
virtual bool computeXyceVectors(std::vector<double> & solutionVars,
                                double time,
                                std::vector<double>&F,
                                std::vector<double>&Q,
                                std::vector<double>&B,
                                std::vector<std::vector<double> > &dFdx,
                                std::vector<std::vector<double> > &dQdx) = 0;
```

This pure virtual function must be implemented by every class derived from the interface. It is called for all time domain analyses, and also for all frequency domain analyses if the class does not also implement frequency domain loads.

A vector of doubles is passed as `solutionVars`, and contains all of the solution variables associated with the General External device to which this interface is tied (by a `setVectorLoader` call). The first entries of the `solutionVars` array are the external nodes in the order listed on the netlist “YGENEXT” line, and subsequent entries are the internal variables, if any were allocated by a `setNumInternalVars` call.

The single double precision value `time` is the current time value being solved for by **Xyce**. For DC and AC simulations this value will always be zero. In transient simulation, it will be the value of the time point currently being attempted.

Recall that for each time step **Xyce** is performing a Newton solve to find the solution, so this method may be called more than once with the same `time` value. Further, during transient simulation **Xyce** uses an adaptive timestepping algorithm, where the time step is adjusted to maintain an acceptable local truncation error; it is therefore often the case that **Xyce** will attempt a time step, conclude that the local truncation error is too large, and then reject that step. When this happens, it will then attempt to take a smaller time step — the value of `time` will then be earlier than the previous call. Therefore, it cannot be assumed that `time` will be monotonically increasing, and it is the responsibility of the coupled code to take this value of time into account when evaluating the DAE contributions.

In Harmonic Balance (HB) simulation, **Xyce** will do time-domain loads at specific time values for all devices that do not provide frequency domain loads, and will then perform Fourier transforms to obtain the HB DAEs. Devices that do provide frequency domain loads will be loaded separately and their contributions added to the Fourier transformed time-domain vectors.

The remaining arguments of `computeXyceVectors` are STL vectors into which the object should load this device's contributions to $\vec{F}$, $\vec{Q}$, and $\vec{B}$ and their derivatives with respect to the slution variables. The length of these vectors is set by the method itself, and should be either length zero (to signify that this device loads nothing into that vector) or the same length as the `solutionVars` vector. Similarly, the matrix size must either be zero (to signify no load) or square with each dimension being the same length as `solutionVars`. When the "YGENEXT" device loads matrix elements from the arrays returned by a computeXyceVectors call, it only accesses the elements of these STL matrices that are flagged as non-zeros by the device's Jacobian stamp. As noted in section 4.1.3, the default Jacobian stamp is fully dense (and so all elements of the STL matrices are copied out and loaded into the full system Jacobians), but may be overridden by the application to take advantage of known sparsity patterns by using a call to `setJacStamp`.

When loading the derivatives into the `dFdX` and `dQdX` matrices, the row index is the equation index, and the column index is the variable index — thus, `dFdX[row][col]` will be the derivative of `F[row]` with respect to variable `col`.

`computeXyceVectors` must return true unless there is a fatal error condition, in which case it should return false.

The example of figure 5.1 demonstrates how a simple linear resistor would implement `computeXyceVectors`.

4.2.2 computeXyceFDVectors

```
virtual bool computeXyceFDVectors(
  std::vector<std::complex<double> > & solutionVars,
  double frequency,
  std::vector<std::complex<double> >&F,
  std::vector<std::complex<double> >&B,
  std::vector<std::vector<std::complex<double> > > &dFdx);
```

The `computeXyceFDVectors` method is called during Harmonic Balance (HB) analysis to perform frequency-domain loads instead of time-domain loads.

For most devices **Xyce**'s HB analysis uses normal time-domain loads, calling their load functions at designated time points to create a full period of information at the desired sampling interval, and then performing Fourier transforms to obtain the needed HB DAE vectors. For devices that have no internal time-dependent state, this works well and simplifies implementation of devices.

Some devices such as the transmission line devices maintain internal state during transient runs that assumes that the simulation is integrating forward in time and makes this method of HB loading invalid. Such devices have to load their frequency domain vectors directly.

Similarly, if coupling to a frequency domain code with no transient analysis, it would be necessary that the general external device load the frequency domain DAE vectors directly, since transient information isn't available from the external simulator.

`computeXyceFDVectors` is used by the general external device to perform frequency domain loads exactly in the same manner as `computeXyceVectors` is used in transient — it is called for each required frequency with a vector of solution variables at that frequency, and must return the $\vec{F}$ and $\vec{B}$ vectors and the derivative of $\vec{F}$ with respect to the solution variables.

`computeXyceFDVectors` is only called by the general external device if `haveFDLoads` returns true. Otherwise the HB loader uses the normal transient loads and Fourier transforms.

`computeXyceFDVectors` must return true unless there is a fatal error condition, in which case it should return false.

Because it has a default implementation that simply returns "true" and does nothing else, `computeXyceFDVectors` need not be reimplemented by user code unless it is necessary to load frequency domain vectors directly (as would be the case for coupling to frequency domain codes). If reimplemented to do loads, `haveFDLoads` must also be reimplemented to return true. An example of a vector loader class that implements frequency domain loads is shown in figures 5.2 and 5.3.

Note: If coupling to a purely frequency domain simulator for which it makes no sense to run **Xyce** in transient, it is still necessary to reimplement the pure-virtual time-domain `computeXyceVectors` function. Since it would be an error to run such a coupled simulation in transient, the `computeXyceVectors` function should simply return false — this will cause any attempt to perform **Xyce** using anything but HB simulation to exit with an error.

4.2.3 haveFDLoads

```
virtual bool haveFDLoads();
```

Xyce's HB loader will use time-domain loads and perform Fourier transforms to obtain the harmonic balance DAEs unless a device implements frequency domain loads directly.

`haveFDLoads` is used by the general external device and the HB loader to determine whether a general external device supports frequency domain loads. If this function returns false (as its default implementation does), then no calls are made to `computeXyceFDVectors`, and instead only time domain loads (calling `computeXyceVectors`) are performed.

If `computeXyceFDVectors` is reimplemented to perform frequency domain loads, `haveFDLoads` should be reimplemented to return true.

4.3 External Output Interface Class

External applications may create output interfaces to cause **Xyce** to transmit solution results in the same manner as those resulting from the presence of `.PRINT` lines in the netlist. This is done by creating objects that derive from the `Xyce::IO::ExternalOutputInterface` class as defined in the file "N_IO_ExtOutInterface.h". Once allocated and initialized, a pointer to these objects must be registered with **Xyce** via the `addOutputInterface` method of the `GenCouplingSimulator` class (see section 4.1).

The methods of this class are broken down into three sets: query, reporting, and output. Query methods are called by **Xyce** during setup of its output manager, and should be implemented to inform **Xyce** of the nature of the output interface, such as the analysis with which it should be associated and the list of output quantities that should be delivered to it. Reporting methods are used by **Xyce** after initial processing to report parsing status and to inform the external application of the actual names of output quantities it will be receiving. Output methods are called repeatedly through a run to deliver the values of the requested quantities.

4.3.1 Query Methods

getOutputType

```
virtual OutputType::OutputType getOutputType();
```

This method is used to determine which analysis should cause the outputter to be invoked. It is similar to the second field of a `.PRINT` as described in the **Xyce** Reference Guide [5]. The outputter associated with the interface object will not be invoked unless the current analysis type matches the value returned by this method.

The allowed return values are those of the OutputType enum in the file "N_IO_OutputTypes.h" that is automatically included by the "N_IO_ExtOutInterface.h" header. The external outputter has not yet been coded to support all of the output types listed in this enum. The supported return values are given below, along with the analysis that applies.

- `Xyce::IO::OutputType::TRAN` – `.TRAN` analysis
- `Xyce::IO::OutputType::DC` – `.DC` analysis
- `Xyce::IO::OutputType::AC` – `.AC` analysis primary output
- `Xyce::IO::OutputType::AC_IC` – `.AC` analysis, initial condition output (usually only needed for debugging of Xyce AC analysis).
- `Xyce::IO::OutputType::HB_FD` – `.HB` analysis, frequency domain output
- `Xyce::IO::OutputType::HB_TD` – `.HB` analysis, time domain output
- `Xyce::IO::OutputType::HB_IC` – `.HB` analysis, initial condition output (time domain)
- `Xyce::IO::OutputType::HB_STARTUP` – `.HB` analysis, startup transient output (time domain)

The default implementation of this method returns `Xyce::IO::OutputType::TRAN`.

getName

```
virtual std::string getName();
```

Returns a string that is used by **Xyce** in error reports concerning set up of the outputter. The default implementation returns the name "ExternalOutput". One need only reimplement this method if creating multiple interfaces, so that any error messages generated by **Xyce** can be easily associated with the code that caused them. These errors would typically concern **Xyce** being unable to parse the requested output fields it gets from the object's `requestedOutputs` query.

requestedOutputs

```
virtual void requestedOutputs(std::vector<std::string> & outputVars);
```

This method is called once by **Xyce** while it is constructing an outputter to determine what output quantities it should deliver to the external application. It corresponds to the output list on a `.PRINT` line.

The application should fill the vector with the names of requested output quantities, one quantity per element of the vector. Any name that is legal on a `.PRINT` line may be used, including the wildcards `V(*)` or `I(*)` and expressions delimited by curly braces ("{" and "}").

If an output interface is intended to be used in transient or frequency domain simulation (as opposed to DC steady state), it is advisable to include the independent variable "TIME" or "FREQUENCY" in the list of requested outputs. If this is omitted, **Xyce** will prepend the appropriate variable to the list. No independent variable is prepended for DC simulation, or if the appropriate variable already appears anywhere in the requested list.

4.3.2 Reporting methods

As **Xyce** sets up an outputter, it will call these methods to report its progress back to the external application.

reportParseStatus

```
virtual void reportParseStatus(std::vector<bool> & statusVec);
```

Xyce will call this method after processing the list of strings returned by `requestedOutputs`. The vector of booleans will be of the same length as the vector obtained from calling `requestedOutputs`, and an element will be "true" if **Xyce** was able to parse the string in that vector position, and "false" if it was not parseable. The application should then be responsible for reporting an error to the user or developer.

outputFieldNames

```
virtual void outputFieldNames(std::vector<std::string> & outputNames) = 0;
```

This pure virtual method must be implemented. There is no default implementation.

After the outputter is set up, **Xyce** will call this function to inform the external application of the actual names of output quantities it will be receiving from **Xyce**, in the order that they will be delivered. This list may not be identical to the list provided by `requestedOutputs`, because **Xyce** will have canonicalized the name (generally by upcasing the string), will have ignored any unparseable fields, will have expanded any wild cards present in the requested list, and may have prepended an independent variable. It is the responsibility of the application code to map the strings returned in this function onto those requested.

Xyce will return canonicalized names in the same order as the non-canonical names were provided by `requestedOutputs`, but when expanding wild cards the ordering of fields will be platform dependent — node and current names are stored in an unordered map, and the actual order in which names are emitted when a wild card is expanded depends on the implementation of the STL unordered map container on the current platform. It is essential therefore that the application not make assumptions about the ordering of expanded wild card output lists and instead use the vector provided by outputFieldNames to get the definitive list of output field names for values that will be provided by **Xyce** during a run.

This method is called by **Xyce** once per run, as if it were a request to output a header line for an output file. It is up to the application to use this information as needed.

4.3.3 Output methods

Once analysis actually begins, **Xyce** sends data to be output to each active outputter at designated intervals. For DC sweep simulations, the outputters are called for each step of the sweep after the solution is converged. For transient simulations, by default the solution is output for every time step computed by **Xyce**, but this can be overridden to output at specified intervals with `.OPTIONS OUTPUT` directives. For AC analysis, the solution is output for each computed frequency. For harmonic balance simulation the entire solution in frequency domain is output to the HB_FD outputters, and the Fourier transform of the frequency-domain solution is output to the HB_TD outputters.

If the netlist contains `.STEP` sweeps, all analyses are repeated for each step of the sweep, as is all output. The outputters are notified when a new step of a sweep is started so they may take special action if needed.

The external output interface class contains methods that will be called by the outputters as the run proceeds to accomplish the actions that would simply result in file I/O for its internal outputter types. These methods should be implemented by the external application to complete the delivery of this data.

outputReal

```
virtual void outputReal(std::vector<double> & outputData);
```

This method is called each time that time-domain data should be output (this includes transient and DC analysis, but also time-domain (HB_TD), initial condition (HB_IC), or startup (HB_STARTUP) output from harmonic balance analysis and initial condition output for AC analysis (AC_IC)). These analyses and auxilliary output types produce only real quantities.

The `outputData` vector will be of the same length as the vector filled by the `outputFieldNames` vector, and the data in each element is the current value of the output quantity named in the names vector.

outputComplex

```
virtual void outputComplex(std::vector<std::complex<double> > & outputData);
```

This method is called each time that frequency-domain data should be output. These are the AC and HB_FD output types.

The `outputData` vector will be of the same length as the vector filled by the `outputFieldNames` vector, and the data in each element is the current value of the output quantity named in the names vector.

finishOutput

```
virtual void finishOutput();
```

This method is called by **Xyce** at the conclusion of a run as if it were a request to output a file footer. It also informs the external application that no further data will be sent through the interface for the current run.

The method has a default implementation that does nothing.

newStepOutput

```
virtual void newStepOutput(int stepNumber, int maxStep);
```

When a netlist contains a `.STEP` directive, **Xyce** will perform repeated runs of any analysis statement in the netlist with the variable or parameter in the `.STEP` line varying each time. This method is called each time a `.STEP` loop transitions to the next value. The total number of steps that will be taken is passed in `maxStep`, and the current step number passed as `stepNumber`. (Note: The first step is step 0.)

The default implementation does nothing.

finishedStepping

```
virtual void finishedStepping();
```

This method is called when all steps of all `.STEP` directives in a netlist have been completed.

5. Examples

This chapter collects various examples that have been referred to previously along with a basic description for an approach to coupling a production-level finite element analysis code. The first section includes the minimal implementation of a linear resistor, a linear capacitor including frequency domain loads, and the two-terminal, serial RLC network. The second section describes the steps and some details for how a production level finite-element (FEM) code can be treated as a lumped element within the Xyce circuit using the General External Device API.

5.1 Basic Examples

In addition to the simple implementations shown here, a fully functional demonstration code that contains several implementations of vector compute interfaces (including several that require one or more internal variables and/or sparse Jacobian stamps), a functional external output interface, and a complete implementation of all code needed to invoke **Xyce** and run netlists that contain certain YGENEXT devices may be found in the **Xyce** source code tree in the `src/test/GenExtTestHarnesses` directory in the file `testGenCoup.C`. This program may be built inside a **Xyce** build directory after configuring and building **Xyce** itself by navigating into the `src/test/GenExtTestHarnesses` subdirectory of the build directory (NOT the source directory) and running the command `make testGenCoup`. This same program is invoked by the test cases found in the “GenCoupAPI” directory of the **Xyce** test suite. The test cases in this directory run DC, AC, Transient, and Harmonic balance simulations.

```
#include <Xyce_config.h>
#include <N_DEV_VectorComputeInterface.h>
class resistorVectorCompute
   : public Xyce::Device::VectorComputeInterface
{
public:
  resistorVectorCompute(double R) : R_(R) {};

  // Not part of the API:
  void setResistance(double R) {R_ = R;};

  virtual bool computeXyceVectors(std::vector<double> & sV,
                  double t,
                  std::vector<double> & F,
                  std::vector<double> & Q,
                  std::vector<double> & B,
                  std::vector<std::vector<double> > & dFdX,
                  std::vector<std::vector<double> > & dQdX)
  {
    F.resize(2);
    Q.clear();
    B.clear();
    dQdX.clear();
    dFdX.resize(2);
    double voltageDrop=sV[0]-sV[1];
    double conductance = 1.0/R_;
    double current = voltageDrop*conductance;
    F[0] = current;
    F[1] = -current;
    dFdX[0].resize(2);
    dFdX[0][0] = conductance;
    dFdX[0][1] = -conductance;
    dFdX[1].resize(2);
    dFdX[1][0] = -conductance;
    dFdX[1][1] = conductance;
    return true;
   }
private:
   double R_;
};
```

Figure 5.1. Simple linear resistor implementation

```
#include <Xyce_config.h>
#include <N_DEV_VectorComputeInterface.h>
class capacitorVectorCompute
    : public Xyce::Device::VectorComputeInterface
{
public:
  capacitorVectorCompute(double C) : C_(C) {};

  virtual bool computeXyceVectors(std::vector<double> & sV,
                      double t,
                      std::vector<double> & F,
                      std::vector<double> & Q,
                      std::vector<double> & B,
                      std::vector<std::vector<double> > & dFdX,
                      std::vector<std::vector<double> > & dQdX)
  {
    Q.resize(2);
    F.clear();
    B.clear();
    dFdX.clear();
    dQdX.resize(2);
    for (int i=0; i<2; i++)
      dQdX[i].resize(2);
    double voltageDrop = (sV[0]-sV[1]);
    Q[0] = voltageDrop*C_;
    Q[1] = -Q[0];

    dQdX[0][0] = C_;
    dQdX[0][1] = -C_;
    dQdX[1][0] = -C_;
    dQdX[1][1] = C_;
    return true;
    }
```

Figure 5.2. Simple linear capacitor implementation (part 1)

```
virtual bool computeXyceFDVectors(
        std::vector<std::complex<double> > &sV,
        double frequency,
        std::vector<std::complex<double> > &F,
        std::vector<std::complex<double> > &B,
        std::vector<std::vector<std::complex<double> > > &dFdX)
  {
    B.clear();
    F.resize(2);
    dFdX.resize(numVars);
    double pi = 3.14159265358979;

    for ( int i=0; i<2; i++)
      dFdX[i].resize(2);

    dFdX[0][0] = std::complex<double> ( 0, frequency*2*pi*C_);
    dFdX[0][1] = -dFdX[0][0];
    dFdX[1][0] = -dFdX[0][0];
    dFdX[1][1] = dFdX[0][0];

    F[0] = dFdX[0][0]*sV[0]+dFdX[0][1]*sV[1];
    F[1] = dFdX[1][0]*sV[0]+dFdX[1][1]*sV[1];

    return true;
  };

  virtual bool haveFDLoads() { return true; };

private:
  double C_;
};
```

Figure 5.3. Simple linear capacitor implementation (continued)

```
class rlcVectorCompute
  : public Xyce::Device::VectorComputeInterface
{
public:
  rlcVectorCompute(double R, double L, double C)
    : R_(R),
      L_(L),
      C_(C)
  {
    jacStamp.resize(4);
    jacStamp[node1].resize(1);
    jacStamp[node1][0] = branch;
    jacStamp[node2].resize(2);
    jacStamp[node2][0]=node2;
    jacStamp[node2][1]=nodeInt;
    jacStamp[nodeInt].resize(3);
    jacStamp[nodeInt][0]=node2;
    jacStamp[nodeInt][1]=nodeInt;
    jacStamp[nodeInt][2]=branch;
    jacStamp[branch].resize(3);
    jacStamp[branch][0]=node1;
    jacStamp[branch][1]=nodeInt;
    jacStamp[branch][2]=branch;
  };
```

Figure 5.4. Serial RLC implementation (constructor)

```
    virtual bool computeXyceVectors(std::vector<double> & sV,
                         double t,
                         std::vector<double> & F,
                         std::vector<double> & Q,
                         std::vector<double> & B,
                         std::vector<std::vector<double> > & dFdX,
                         std::vector<std::vector<double> > & dQdX)
    {
      int numVars=sV.size();

      F.resize(numVars);
      Q.resize(numVars);
      B.clear();
      dFdX.resize(numVars);
      dQdX.resize(numVars);
      for (int i=0; i<numVars; i++)
      {
        dFdX[i].resize(numVars);
        dQdX[i].resize(numVars);
      }
      F[node1]   = sV[branch];
      F[node2]   = 0.0;
      F[nodeInt] = -sV[branch];
      F[branch] = sV[branch]*R_ - (sV[node1]-sV[nodeInt]);

      Q[node1] = 0.0;
      Q[node2] = -C_*(sV[nodeInt]-sV[node2]);
      Q[nodeInt] = C_*(sV[nodeInt]-sV[node2]);
      Q[branch] = L_*sV[branch];
```

Figure 5.5. Serial RLC implementation (computation)

```
dFdX[node1][node1] = 0.0;
dFdX[node1][node2] = 0.0;
dFdX[node1][nodeInt] = 0.0;
dFdX[node1][branch] = 1.0;

dFdX[node2][node1] = 0.0;
dFdX[node2][node2] = 0.0;
dFdX[node2][nodeInt] = 0.0;
dFdX[node2][branch] = 0.0;

dFdX[nodeInt][node1] = 0.0;
dFdX[nodeInt][node2] = 0.0;
dFdX[nodeInt][nodeInt] = 0.0;
dFdX[nodeInt][branch] = 1.0;

dFdX[branch][node1]   = -1.0;
dFdX[branch][node2]   =  0.0;
dFdX[branch][nodeInt] =  1.0;
dFdX[branch][branch]  =  R_;

dQdX[node1][node1]   = 0.0;
dQdX[node1][node2]   = 0.0;
dQdX[node1][nodeInt] = 0.0;
dQdX[node1][branch]  = 0.0;

dQdX[node2][node1]   =  0.0;
dQdX[node2][node2]   =  C_;
dQdX[node2][nodeInt] = -C_;
dQdX[node2][branch]  =  0.0;

dQdX[nodeInt][node1]   =  0.0;
dQdX[nodeInt][node2]   = -C_
dQdX[nodeInt][nodeInt] =  C_;
dQdX[nodeInt][branch]  =  0.0;

dQdX[branch][node1]   = 0.0;
dQdX[branch][node2]   = 0.0;
dQdX[branch][nodeInt] = 0.0;
dQdX[branch][branch]  = L_;

return true;
};
```

Figure 5.6. Serial RLC implementation (computation, continued)

```
    public:
      std::vector< std::vector<int> > jacStamp;
    private:
      double R_;
      double L_;
      double C_;
      static const int node1=0;
      static const int node2=1;
      static const int nodeInt=2;
      static const int branch=3;
    };
```

Figure 5.7. Serial RLC implementation (class variables)

5.2 Finite-Element Code Examples

This section describes an approach for coupling a FEM analysis code to a circuit using the `computeXyceVectors` callback API described in section 4.2.1. There are two key steps consisting of 1) incorporating the values in `solutionVars` provided by **Xyce** and 2) providing the corresponding device current response and associated derivtives (Jacobian contributions) based on the FEM analysis back to **Xyce**. The approach described here is currently employed by the Aleph and ALEGRA physics codes with more details provided in [7].

It is important to recognize that representing a high-fidelity mesh-based device as a lumped circuit element requires both an expansion of lumped circuit node information from **Xyce** to the FEM analysis and a complementary coarsening (or lumping) of mesh-based information from the FEM analysis back to **Xyce**. The former is achieved by associating a netlist circuit node with an equipotential conductor surface in the mesh-based description, i.e. all mesh nodes belonging to the conductor surface have the same voltage at any given time. The voltage values for each equipotential conductor are provided by the `solutionVars` data in the `computeXyceVectors` callback method. The reverse step of coarsening mesh-based information into a single value of charge or current at a circuit node is done by the analysis code performing an appropriate surface integral consistent with the physical model and the finite-element method. The result of these surface integrations are passed back to **Xyce** in the $\vec{F}$, $\vec{Q}$, and $\vec{B}$ vectors (along with Jacobian contributions to $\frac{d\vec{F}}{dX}$ and $\frac{d\vec{Q}}{dX}$).

Incorporating **Xyce** Data

Aleph and ALEGRA both model their internal potential field using a quasistatic assumption with governing equation

$$\nabla \cdot (\epsilon \nabla \phi) = s$$

where ϕ is the scalar potential or voltage, ϵ is the material permittivity, and s accounts for either dielectric material in ALEGRA or charged particle density in Aleph. The electric field is computed as the gradient of the scalar potential field as $\vec{E} = -\nabla \phi$.

Recognizing that the governing electrostatic equation is linear, it can be decomposed into $n+1$ subproblems suitable to coupling to the electric circuit. For each conductor C_i, the following homogeneous problem can be solved

$$\nabla \cdot \epsilon \nabla \phi_i = 0 \quad \texttt{with} \quad \phi_i = \delta_{ij} \quad \texttt{on conductor} \quad C_j \; .$$

for all i, j = 1 to n with n being the total number of conductors in the mesh-based problem. The remaining equation for ϕ_{n+1} captures the effects of the source term and takes the form

$$\nabla \cdot \epsilon \nabla \phi_{n+1} = s \quad \texttt{with} \quad \phi_{n+1} = 0 \quad \texttt{on all conductors} \quad C_j \; .$$

These potentials are then superposed to compute the potential field solution using

$$\phi = \sum_j^n V_j \phi_j + \phi_{n+1}$$

where V_j are the unknown voltages on each conductor C_j and are supplied by **Xyce** in the `solutionVars` vector in the `computeXyceVectors` call back function.

Computing Application Reponse DAE Contributions

Once the scalar potential field has been computed, the responses supplied by the FEM application to **Xyce** can be computed as the value of charge Q_i on each conductor C_i. This takes the form of a surface integral over each conductor surface,

$$Q_i = \int_{C_i} \left(\vec{E} \cdot \vec{n} \right) da \; .$$

Using the superposed form of the scalar potential solution, the surface charge can be expressed as

$$Q_i = \sum_j K_{ij} V_j + S_i$$

where

$$K_{ij} = -\oint_{C_i} (\epsilon \nabla \phi_j \cdot \vec{n}) \, da$$

and

$$S_i = -\oint_{C_i} (\epsilon \nabla \phi_{n+1} \cdot \vec{n}) \, da$$

with $\vec{n}$ being an outward pointing unit vector normal to the conductor surface. These values of Q_i can be supplied to **Xyce** in the $\vec{Q}$ response vector of the `computeXyceVectors` call back function. Alternatively, currents can be supplied in the $\vec{F}$ response vector after differentiating the expression for Q_i with respect to time to obtain the current entering the i^{th} conductor.

More details for how the surface integrations can be performed by leveraging the native FEM methodology along with desireable properties of K_{ij} and S_i and how to achieve them can be found in [7],[8],[9],[10].

References

[1] Derek C. Lamppa, Christopher J. Garasi, Allen C. Robinson, Thomas V. Russo, David N. Shirley, and Matthew S. Aubuchon. The Xygra EM Gun Simulation Tool. Technical Report SAND2008-8227, Sandia National Laboratories, 2008.

[2] Allen C. Robinson, Thomas A. Brunner, Susan Carroll, Richard Drake, Christopher J. Garasi, Thomas Gardiner, Thomas Haill, Heath Hanshaw, David Hensinger, Duane Labreche, Raymond Lemke, EdwardLove, Christopher Luchini, Stewart Mosso, John Niederhaus, Curtis C. Ober, Sharon Petney, William J. Rider, Guglielmo Scovazzi, O. Erik Strack, Randall Summers, Timothy Trucano, V. Greg Weirs, Michael Wong, and Thomas Voth. ALEGRA: An Arbitrary Lagrangian-Eulerian Multimaterial, Multiphysics Code. In *46th AIAA Aerospace Sciences Meeting and Exhibit 2008*. American Institute for Aeronautics and Astronautics (AIAA), 2008. doi: 10.2514/6.2008-1235. URL `https://doi.org/10.2514/6.2008-1235`.

[3] Eric R. Keiter, Scott A. Hutchinson, Robert J. Hoekstra, Lon J. Waters, and Thomas V. Russo. Xyce parallel electronic simulator design: Mathematical formulation, version 2.0. Technical Report SAND2004-2283, Sandia National Laboratories, Albuquerque, NM, June 2004.

[4] C. W. Ho A. E. Ruehli and P. A. Brennan. The modified nodal approach to network analysis. *IEEE Trans. Circuits Systems*, 22:505–509, 1975.

[5] Eric R. Keiter, Karthik V. Aadithya, Ting Mei, Thomas V. Russo, Richard L. Schiek, Peter E. Sholander, Heidi K. Thornquist, and Jason C. Verley. Xyce Parallel Electronic Simulator: Reference Guide, Version 6.9. Technical Report SAND2018-5331, Sandia National Laboratories, Albuquerque, NM, 2018.

[6] *Xyce Building Guide*. URL `https://xyce.sandia.gov/documentation/BuildingGuide.html`.

[7] J. H. J. Niederhaus et al. User Manual for Ferroelectric (FE) Modeling in ALEGRA. Technical report SAND2017-13762, Sandia National Laboratories, Albuquerque, NM 87185, 2017.

[8] Allen C. Robinson and Donald W. Robinson. A best approximation evaluation of a finite element calculation. *Linear Algebra and Its Applications*, 302:367–375, 1999.

[9] Thomas J.R. Hughes, Gerald Engel, Luca Mazzei, and Mats G. Larson. The Continuous Galerkin Method Is Locally Conservative. *Journal of Computational Physics*, 163

(2):467 – 488, 2000. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.2000.6577. URL http://www.sciencedirect.com/science/article/pii/S002199910096577X.

[10] Philip M. Gresho, Robert L. Lee, Robert L. Sani, Miriam K. Maslanik, and Brian E. Eaton. The consistent Galerkin FEM for computing derived boundary quantities in thermal and or fluids problems. *International Journal for Numerical Methods in Fluids*, 7(4):371–394. doi: 10.1002/fld.1650070406. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1650070406.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories