**SCREAMER V4.0 – A Powerful Circuit Analysis Code**

# R. B. Spielman , Y. Gryazin

*Idaho State University Pocatello, ID, USA*

*Abstract*

Screamer is a fast, highly optimized circuit-analysis code that was originally developed at Sandia National Laboratories in 1985. Screamer is written in Fortran 77 (with some extensions) and is highly optimized for speed. Screamer V3 solved electrical circuits having a limited, in-line circuit topology with restricted branches. This restricted circuit topology allowed for a very efficient matrix solver. We will describe the mathematical basis of Screamer and show how the topology leads to a very sparse matrix. We will describe the evolution of Screamer V4.0 and show how the development of powerful (> 0.1 TFlops) PCs with large amounts of memory (> 8 GB) enables major extensions to Screamer’s circuit topology without a significant loss in speed. Screamer incorporates many physics-based models such as dynamic loads, gas switching, water switching, oil switching, magnetic switching, and vacuum transmission lines, which are important to the high-voltage, pulsed-power community. Additional circuit models or modifications to existing models can be readily implemented in Screamer. Screamer runs on the Macintosh OS (9 & 10), LINUX, and Windows 7 & 8.

# Introduction

SCREAMER is a special purpose circuit code originally developed in 1985 by Kiefer & Widner1 for pulsed power applications. It is a very fast, accurate and flexible circuit code with extensive physics models built for pulsed power applications. Screamer is written in Fortran 77 (with F90 extensions) and is presently compiled under the GNU framework (gFortran). SCREAMER runs in the Mac OS, LINUX, and Windows environments. We describe here the development of SCREAMER V4.0, a totally new version with greatly reduced topological restrictions and a new full matrix solver. Screamer V4.0 takes advantage of the modern computing capabilities found on desktop computers (RAM > 8 GB and speeds >

0.1 TFlops).

Screamer, as originally written, has a numerical differencing scheme with one time level; is fully implicit; and is second-order accurate.

# Legacy SCREAMER Topology

SCREAMER was designed with a limited linear circuit topology in which circuit elements must be arranged in a linear fashion – a branch. (See Fig. 1.) This may seem to be very restrictive in terms of allowable circuits (no series capacitors or parallel inductors) but it is not. Screamer allows secondary branches across both series elements and parallel elements of the primary or main branch. This effectively permits arbitrary inductors in parallel and capacitors in series to the main branch. In addition, branches allow for the modeling of multi-module systems in which many modules may tie together near the final load. A detailed description of the derivation of the finite difference solution is available from the lead author upon request.



Fig. 1 A linear circuit branch showing a progression of circuit elements added together to make up a complete circuit.

The original version of Screamer implicitly did not allow secondary branches to rejoin the main branch and did not allow branches off of secondary branches. Even with this significant circuit topology restriction, Screamer was a very useful-pulsed power tool.

# The SCREAMER Circuit Model

SCREAMER was deliberately designed to solve a limited circuit in which the primary circuit element is the PI block shown in **Fig. 2**. This was deliberately done to restrict the mathematics to a first-order differential equation.

The PI block is the fundamental element of transmission line segments. In this case, the resolution of the TL is determined by the values of the PI block making up the smallest length of the TL. Individual circuit components such as resistors, inductors, and capacitors are simply PI blocks with the unneeded values in the PI

\*Work supported by Sandia National Laboratories under Purchase Order #1518167.

 email: spierick@isu.edu

block set to 0.

(4)

(5)

Fig. 2 A typical Pi-block showing the fundamental circuit elements making up the block.

### SCREAMER Differencing Scheme

From first principles, we derive the fundamental equations that are used in Screamer. We see the layout of two arbitrary Screamer nodes in **Fig. 2**.

We can now express Eqs. 1 & 2 in terms of new and old variables. It is instructive to note that the presence of a branch at the *ith* node only shows up in the *ith* current equation of the main branch. (This will lead directly to a sparse matrix when branches are used.) We collect the new variables on the left and the old variables on the right. The current equation is in Eq. 6 and the voltage equation is in Eq. 7.

*n o n o n o*

 =

0.5 *Ii*1  *Ii*1  0.5 *Ii*  *Ii* 0.5 *Ibi* + *Ibi*

*n o Ci n o*

We can write the current equation of a single node. The current into the *ith* node (from the *ith* – 1 node) equals the current out of the *ith* node, plus the current out of the *ith*

*n*

0.5 *I*

0.5*Gi Vi*  *Vi*

*n n Ci*

+ *t Vi*  *Vi*

*n*

node into the branch *b* at the *ith* node, and plus the current that flows through the circuit elements to ground from the

*i*1  *Ii*  *Ibi*

 0.5*Gi* + *t Vi* =

*o o o*

0.5 *I*

*Ci o*

*ith* node. Conductance *G* (1/*R*) is the resistive path to ground and the capacitance *C* is the capacitance to

*n o n*

*i*1  *Ii*  *Ibi*

*o*

+ 0.5*Gi*  *t Vi*

(6)

ground. The currents are in Eq. 1 below.

0.5 *Vi*  *Vi*

 0.5 *Vi*+1  *Vi*+1 =

*n o Li n o*

(1)

0.5*Ri Ii*  *Ii* + *t Ii* *Ii*

We can then write the voltage equation between the *ith*

*n n*

*Li n*

and *ith* +1 nodes. The voltage drop between the *ith* node and the *ith* + 1 node is only the voltage drop across the *ith*

0.5 *Vi*  *Vi*+1

 0.5*Ri* + *t Ii* =

0.5 *V o*  *V o* + 0.5*R*  *Li*

*I o* (7)

resistor and *ith* inductor due to the *ith* current. These voltages are shown in Eq. 2 below.

*i*1 *i*

*i t i*

 (2)

We can then express both equations in terms of half time steps in which the values of the current and voltage from the old time step (*o*) are known and the values at the

new time step (*n*) are to be calculated. The use of split

The RHS of both equations are constants are derived only from the circuit variables in prior time steps. The circuit values on the LHS of the equations in the parentheses are also constants. Let’s rewrite Eqs. 6 & 7.

*n n n n*

*Ci*

*t*

0.5 -*I*  *I*  *I*  0.5*G* 

time steps increases the fundamental accuracy of the

*i*  1 *i bi*

*i Vi* =

calculation. The voltage and current expressions for half

*o o o o*

0.5 *I*

*Ci*

*t*

time steps are below in Eq. 3.

*i*1  *Ii*  *Ibi* 

*n n Li n*

* 0.5*Gi Vi*

(8)

0.5 *Vi*  *Vi*+1

 0.5*Ri*  *t Ii* =

(3) *o o Li o*

The partial derivative of (*LI*) and (*CV*) with time is simply expressed as a difference in the values of *I* and of

0.5 *Vi*  *Vi*+1 

*t*  0.5*Ri Ii*

(9)

*V* divided by the time step. This is shown in Eqs. 4 & 5. Note the use of the difference in current and voltage in the partial derivative with the assumption that *L* and *C* are not functions of time.

We can further simplify the equations by defining the four constants separately in Eq. 10. We use exactly these four constants in Screamer. The *Ai* constants depend only on circuit values. The *Bi* constants can be populated with circuit values and from the results of the prior time step.



(10)

By substituting these variables into Eqs. 8 & 9 we see that we have a simple set of 2•*N* equations and 2•*N* unknowns. Where *N* is the total number of nodes and *i* = 1, *N*. Thus, we finally have the simple, linear-equation representation of the circuit equations in Eqs. 11 & 12. Each of these equations is recalculated moving forward time step by time step and we leave out the notation indicating a specific time step for clarity.

##  (11)

 (12)

We can take the coupled set of Eqs. 11 & 12 and rewrite them in terms of alternating current and voltage with increasing node index *i* in Eqs. 13 & 14. Finally, for simplification of the notation, we will drop the superscript *n* (which was indicative of the “new” value of voltage, which is now assumed to be calculated), add a superscript *k* indicative of the branch number, where *k* = 1, *nb* and *nb* is the total number of branches.

(13)

(14)

For only the case of i > 1 and I < *nk* (*i*), we multiply both Eqs. 13 & 14 by 2 and Eq. 14 equation by -1. While these choices seem arbitrary at this point, we have determined that this reduces number of calculations in the solver. The constant *nb* remains the total number of branches in the problem. Eq. 13 is the current equation for the *kth* branch and in the *ith* node of that *kth* branch, current leaves the *kth* branch into the first node a new *lth* branch. In Screamer, the branch index inequality *k* < *l* must hold. (In the limitation for the legacy solver, branches cannot have branches – except branch *k* = 1 – and all of the current and voltage equations for the branches (*k* > 1) do NOT have a branch term.)

(15)

(16)

Eqs. 15 & 16 are the equations we will use to populate the full problem matrix and vector in the section below for *i* > 1 and *i* < *nk*(*i*).

### SCREAMER Matrix Population

Assume a simple example with no branches. We can express this simple problem having three nodes as a vector with 6 elements in the matrix formation shown in Eqs. 13 – 16. The *AVi* and *AIi* constants are used to populate the ***A*** matrix and the *BVi* and *BIi* constants are used to populate the ***b*** vector. The first row has no current from the prior node and the last row has no voltage in the following node. This is true even with branches. Be careful with the first and last nodes of all branches, they are based on Eqs. 13 & 14.



(17)

We can express this matrix and the two vectors in simple matrix notation. Where the ***x*** vector is the solution vector.

***A*** • ***x*** = ***b*** (18)

### How SCREAMER Actually Builds the A matrix and b vector

The discussion above is mathematically correct but Screamer actually generates a slightly different node structure and different node counts than the simple discussion of circuit elements above might suggest. Building a circuit from Screamer circuit blocks requires open nodes for connectivity to the next block. In practice, this means that we need *every* circuit block to end in a “dangling” resistor/inductor series block (RLseries). With this circuit configuration one can connect any type of circuit block in series. (Consider the trivial case if you tried to connect two RCgrounds one after the other.)

Screamer, as written, creates an extra *phantom* block

and nodes that separate actual circuit elements. For example, a simple resistor & capacitor to ground block (RCground) mathematically requires only a single *ith* node and the *G* & *C* elements to ground but Screamer adds a phantom RLseries element following the *ith* node. In this case, the *R* & *L* in the *ith* phantom RLseries element are set to zero.

Similarly, a resistor and inductor in series block (RLseries) would naturally be placed between the *ith* and *ith*+1 nodes. Phantom RCground elements are placed at the *ith* and *ith*+1 nodes. A phantom RLseries element follows the *ith* + 1 node just as is done for the RCground block. In this case, the *G*’s (1/*R*) & *C*’s in the RCgrounds are set to zero and the *R* & *L* in the second, phantom, RLseries are set to zero. Only the first RLseries element has non-zero values.

Fig. 1 above shows the nodes and elements of a simple example problem (without branches) that would be solved in Screamer with the phantom nodes.

For example, with an RCground at node 1, there is no *actual* series element between nodes 1 and 2 to connect to the next block, so a *phantom* RLSeries element must be included, in which the parameters *L1* and *R1* are both set to

0. Screamer puts a connecting “wire” in the circuit. This means that *V(i)* = *V(i+1)* for all time steps or *V(i+1) - V(i)* = *0* for all time steps. Think of an RCground block as having a phantom RLseries element with the *Ri* & *Li* values zero.

There is an RLseries block between nodes 2 & 3 with two phantom RCground elements with their contents, *C* & *G*, set to zero and a dangling phantom RLseries block with all of its parameters zero following node 3. If one did not place a node immediately after the RLseries block after node 2 then connection to the next node would be difficult. The first RCground is at the 2*nd* node and the second RCground is at the 3*rd* node. Thus, an RLseries block actually populates ALL of the nodes that are available in that block and ends the block with a dangling phantom RLseries element.

In the end, all this does is greatly simplify the way that arbitrary circuit elements can be tied together without consideration for internal code quirks. This has the very unfortunate effect of increasing the number of nodes that a full matrix solver will have to handle AND, more unfortunately, forcing many of the *A* and *B* constants to have a value of zero.

### SCREAMER Legacy Solver

The matrix solution to the sparse tri-diagonal matrix described in Section C above has significant mathematical problems. Many of the diagonal elements will always have a value of zero due to the large number of phantom nodes. See Eqs. 13 – 16 and Eq. 17. A matrix having diagonal values that are zero cannot be solved with the usual methods due to divide by zero problems in the solver algorithms.

Kiefer & Widner1 addressed this problem by inverting the order of the current and voltage equations in the ***A*** matrix and the ***b*** vector (except for the first and last nodes). This creates a pentadiagonal matrix in which the diagonal elements are guaranteed to be non-zero. Thus, the legacy solver in Screamer should properly be considered a very sparse, pentadiagonal solver. (There are tests in the Screamer solver to make sure that the first node diagonal coefficients are non-zero in all branches.) In the equations below we can see that each equation has two elements either following or preceding the new diagonal element. The example below is the inverted populated matrix from the End Branch example. Only the second and third rows are flipped in this case. In this case the diagonal elements can never be zero.

The legacy version of Screamer used a custom matrix

solver in which every attempt was made to reduce memory usage and maximize speed. Branches show up as off diagonal elements that limit sparseness.

Screamer compressed this sparse pentadiagonal matrix into a single vector and managed the indices. The primary problem was that the solver was totally undocumented and the indices handling was obscure at best.

This sparse matrix structure allowed a very fast solver that, even today, represents the fastest possible solver for a very sparse matrix of this type. Typical memory usage for the original version of Screamer was less than 100 MB.



(19)

### E. SCREAMER Limitations

From the discussion above, one immediately sees that the limitation of branches in secondary branches is significant. This is done to prevent a very sparse matrix from becoming significantly less sparse and breaking the existing matrix solver.

This branch limitation prevents the detailed modeling of multi-module generators. It often forced users to model such large pulsed-power systems as lumped generators with multiple modules reduced to a single module.

SCREAMER V4.0 was developed to remove this topological restriction.

# SCREAMER V4.0

The impressively fast personal computers that are commonly available today also sport impressive memory sizes. It is common to see a personal computer having

> 8 GB of memory and approaching a teraflop of processing power. Furthermore, personal computer capabilities are continuing to increase very rapidly. We anticipate that 64-GB DRAM will be the default memory level by 2018 and that the development of > 8-core CPUs will result in multi-teraflop speeds on the desktop. This continuing improvement in processing capability suggested that we revisit the Screamer matrix-solver algorithm and convert from a circuit-limiting, very sparse matrix solver to a more general sparse matrix solver - even if, in the short term, there was a real increase in the computational time needed for typical problems. When we started this effort we did not know what the slow down factor would be.

### SCREAMER V4.0 Goals

Our goals for writing a the new Screamer V4.0 were:

1. To implement a matrix solver that fundamentally allows unlimited branches in secondary (and even, tertiary) branches,
2. To have the problem matrix ***A*** and the problem vector ***b*** explicitly defined and directly traceable to the mathematical derivation of the problem,
3. To be able to diagnose the matrix elements for all population problems,
4. To have the matrix solver in a separate subroutine connected to Screamer only through passed variables

– no common blocks, and

1. To do all of this with a minimal reduction in performance.

We also wanted to use this opportunity to make the matrix solver portion of SCREAMER more transparent and better documented (commented). This is critical for future support. All we would have to do is to change a single subroutine if a new matrix solver were needed at some time in the future.

### The SCREAMER V4.0 Solver

In this section we consider the implementation of the fast, direct linear solver developed for Screamer 4.0. The coefficient matrix in the resulting linear system is sparse and cannot be efficiently stored in the band formats adopted in LINPACK and LAPACK. Our attempt of using direct band solvers from the LAPACK package based on the band storage formats resulted in significant lost of efficiency on several benchmark problems in comparison with the existing fully optimized solver used in the previous Screamer release. On some of the problems the CPU times increase by factors much more than 10. This forced us to adopt a sparse direct solver.

We choose to develop our own sparse direct solver instead of using existing sparse solvers such as the PARDISO solver (http://www.pardiso−project.org) for two reasons: first, the sparse matrix storage format adopted in the Screamer is quite different from the standard sparse formats such as COO, CSR, CSC (see e.g. <http://en.wikipedia.org/wiki/Sparse> matrix) and so on; and, second, is the simplicity of the distribution pattern of entries in the coefficient matrices produced in Screamer. The newly developed solver exhibits the efficiency of the standard sparse solvers, i.e. the number of multiplications in the proposed solver is *O(N)*, where *N* is the number of non-zero entries in the sparse matrix. It also has a very transparent programming implementation due to the efficient storage format and a simple, repeating sparse- matrix structure. Next, we describe the details of the developed sparse direct solver.

SCREAMER’s topology and restrictive circuit

elements result in a linear system ***Ax*** = ***b*** that can be

presented in the following block form.



In this system an *Ak* block corresponds to the *kth* branch. *Ak*, in turn, also has two rows of block structure corresponding to the nodes in the *kth* branch. In the system above we skipped the indication of the branch connections in the off diagonal elements but we will consider them later. We also use the notation  for the first row in the *lth* block.

Then the *ith* and (*i* + 1)*st* 2-rows block of *Ak* can be presented in pentadiagonal form below.



In this matrix the *ith* block has already been transformed into row-reduced echelon form and we consider the details of the transformation of the (*i* + 1)*st* block. For convenience, in this matrix we indicate just one branch connection term  where  is the number of branches *kj* connected to the branch *kl* with *kj* < *k* < *kl*. It also includes the connection of the *lth* branch to the *kth* branch at the *vth* block, if *v* < *i*. It can be seen that the transformation of the (*i* + 1)*st* block of the *kth* branch to the reduced echelon form requires the elimination of two entries in the positions (2*i* + 2, 2*i*) and (2*i* + 2, 2*i* + 1) in the matrix block corresponding to the *kth* branch. The first step in the process requires only additions.



Note that the second term in the second equation is almost always zero.

The second elimination step requires just two multiplications.



The scaling of the (2*i* + 2)*th* row results in additional  multiplications. This shows that the forward elimination step for the (*i* + 1)*st* block of the *kth* branch requires  multiplication operations.

Next, we consider the forward Gaussian elimination step at the first node of a branch *kl*. We assume that the branch *kl* is connected to the ith block of the branch *k* (*k* < *kl*). Also we consider that all rows above the first row in the block corresponding to the *lth* branch have already been transformed into row-reduced echelon form. Then we can present the relevant rows of the matrix in the form below.



Here,  and . If we denote the difference between these column positions by , then we need ξ*l* iterations of a 2-step process to sequentially move the first non-zero entry to the right and eventually eliminate it. The first step in the process is:

*al* = *al ak* ,

connections: If a branch *kl* connected to a branch *km* then the entry in the unknown vector corresponding to the first node of the branch *kl* must come after all entries in the unknown vector corresponding to nodes in the branch *km*.

# SCREAMER V4.0 Implementation

We implemented the new solver routines in Screamer in two, very clear steps:

* 1. We carefully populated the solution matrix with the Screamer-generated matrix coefficients with a separate subroutine (*solver\_band\_matrix1.f* ) and
	2. We created a completely separate, stand-alone matrix solver subroutine (*solver\_mdgauss.f* ) – in parallel with the existing solver. The matrix solver uses only passed variables.

This approach allowed us to gain a clear view of the structure of the solution matrix that was generated by Screamer and to be able to directly compare potential new matrix solvers with the existing legacy solver.

Different matrix solvers can be implemented easily by changing a single subroutine.

# Conclusion

We have completed a rewrite of the Screamer matrix solver subroutine. This new solver is only 2X slower than the legacy Screamer solver but is capable of solving

2*bk l*1, 2 *j*

2*bk l*1, 2 *j*1

2 *j*1, 2 *j*

problem matrices that are not as sparse. We populate the

*al* = *al*  *al ak* ,

2*bk l*1, 2 *j*+1

2*bk l*1, 2 *j*+1

2*bk l*1, 2 *j*1

2 *j*1, 2 *j*+1

***A*** matrix in a clear and transparent fashion. This allows

*bl* = *bl*  *al bk* .

for a simple diagnostic analysis of the problem matrix and

2*bk l*1

2*bk l*1

2*bk l*1, 2 *j*1

2 *j*1

a direct comparison with the problem mathematics.

We also can note that the first term in the right hand side of the second equation is almost always zero. The second step can be written as

*al* = *al*  *al ak* ,

The new matrix solver allows us to attack more complex problems that populate the ***A*** matrix with many more off diagonal components. We refer to this in Screamer as allowing “branches-in-branches”. Version

4.0 of Screamer allows unlimited tertiary branches. This

2*bk l*1, 2 *j*+1

2*bk l*1, 2 *j*+1

2*bk l*1, 2 *j*

2 *j*, 2 *j*+1

means that in addition to secondary branches off of the

*al* = *al*  *al ak* , ** = 1, ..., *mi*,*k* ,

main branch, we allow tertiary branches off of secondary

2*bk l*1, 2*bk *

2*bk l*1, 2*bk *

2*bk l*1, 2 *j*

2 *j*, 2*bk *

*j*,**

branches. This single change allows for vastly more

*bl* = *bl*  *al k*

2*bk l*1

*k l*

2*bk l*1

2*b* 1, 2 *j b* 2 *j* , *j* = 1, ..., *bk*  1 .

*l*

complex machine topologies to be modeled. Screamer V4.0 is now fully operational on Macintosh, LINUX,

Given *N* equations in the matrix, the number of multiplications required on the back substitution step is estimated by  . We can see that if the number of branches remains fixed, both the forward and back substitution steps are approximately *O(N)*. So, the resulting algorithm presents a very efficient, direct

linear solver.

We only allow connections of secondary branches to the main branch or a tertiary branches to secondary branches connected to the main branch. But, as a matter of fact, the constructed direct linear solver has no limitation on the number of embedded branch levels. Essentially, there is only one restriction on the branch

UNIX, and Windows platforms.

# REFERENCES

[1] M. L. Kiefer & M. M. Widner, “SCREAMER – A Single-Line Pulsed-Power Design Tool”, in Proc. of the 5th IEEE Pulsed Power Conference, Arlington, VA, June 10-12, 1985, p. 685.