Matrix Solution: The DSP control equations can be expressed using matrix algebra as shown in Figure 1. Assume there are j states that need to be evaluated, with k of them having a delay history. The equations can be arranged as shown with all trivial solutions at the bottom of the matrix. Let Hn be the history value Hn. Then the j+k by j sub matrix at the top will have its right hand side equal to zero. After solving the matrix the Vn values are substituted into the Hn RHS for the next iteration. There may be more states than history because some of the states may include input and outputs. The main diagonal is scaled to be 1 so that there is no divide required in the solution. For large j, the matrix coefficients should be sparse and non zero values should be near the main diagonal. That's equivalent to having a number of blocks with a single input and output cascaded. If the original matrix had non-zero coefficients below the main diagonal, then the matrix solves an algebraic set of simultaneous equations. DSP's can be made to have to all zero values below the main diagonal by judicious use of backward euler integration to break up the signal flow. That has the side effect of adding delays and reducing controller bandwidth.

LU decomposition, following the forward substitution gives us exactly what's needed [2]. Then backward substitution is a multiply accumulate series for all non zero coefficients followed by division by the main diagonal value. If mixed precision is used, the main diagonal can be normalized to unity; eliminating the division. If integer or fractional scaling is used, the result can be multiplied by a predetermined constant, formed by dividing the scaling value by the main diagonal value, then applying the inverse of the scaling value to the outputs. The solution proceeds from the jth row and j+1 column, summing the products of the non-zero coefficient with their associated states. An array of coefficients is made in the order they will be used and a corresponding array of state-pointers can be made to make maximum use of the DSP multiply accumulate capability.

```
const int16  coef[numRowCoef];
iInt16 * varptr[numRowCoef];
int16 Vn;
while (numRowCoef--)
       Vn += *Coef++ *  *(*varptr++));
```

C compilers will figure this out; but there's always hand coded assembly language to fall back on.

For Reduced Instruction Set Computers, RISC, it may be necessary to limit the range of variable index change from one computation to the next. This can be accomplished by moving the rows with H coefficient up until they are just below the first coefficient used in that column, and the moving the column left to place the unit value on the main diagonal. Such a movement doesn't change the Lower triangle zero condition; but it tends to cluster coefficients along the main diagonal. Then the varptr usage shown above is replaced as shown below:

```
const int16  coef[numRowCoef];
iInt16  offset[numRowCoef];
int16 *varptr;
int16 Vn;
while (numRowCoef--)
       Vn  +=  *Coef++  *    *(varptr  +
*offset++);
```

This form may need some adjustment depending on how the C compiler does its optimization. If the user identifies states that need a solution, then unwanted states can be eliminated by matrix manipulation. That reduces the number of MAC initializations and result storage; making a faster solution.

$$\begin{bmatrix} 1 & a12 & a13 & \dots & \dots & a1j & \dots & & a1(j+k) \\ 0 & 1 & a23 & & & a2j & \dots & & \dots \\ 0 & 0 & 1 & a34 & & a3j & \dots & & \dots \\ 0 & 0 & 0 & 1 & & a4j & \dots & & \dots \\ \dots & & & & \dots & ajj & \dots & & a(j+k)(j+k) \\ \dots & & & & & \dots & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} V1 \\ V2 \\ \dots \\ \dots \\ Vj \\ Hk \\ H3 \\ \dots \\ H1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dots \\ \dots \\ 0 \\ Vkp \\ V3 \\ \dots \\ V1p \end{bmatrix}$$

Figure 1, A matrix solution has RHS(0 thru j)=0

TODO: **Extract matrix from spice**
Eliminate trivial data (0 current stuff)
Eliminate unwanted states
Code Generation
If-then-else resolution