AIAA-2000-4395

VECTORIZATION BLOCKS FOR SIMULINK

Giampiero Campa‡, Mario Innocenti†,
Department of Electrical Systems and Automation, University of Pisa, 56126 Pisa, ITALY

Abstract

The paper presents two blocks that extend Simulink range of applicability to varying matrices, allowing the user to build (without writing a single line of code) dynamical systems that can be both complex and suitable for real time simulations. First of all, the three common techniques to build dynamical system with Simulink will be presented, showing the advantages and disadvantages of each of them. Then we will explain why (when dealing with complex multivariable systems) neither of the three can be at the same time easy to use and suitable for real time simulations. Two Simulink blocks will then be proposed that solve this problem by allowing varying matrices to be handled directly in the Simulink environment. The aspect, the user interface, and the most important part of the code of these blocks will be discussed, and their use will finally be explained by means of some examples. In particular, we will present an example on how to implement a complex multilayer adaptive neural network using these two blocks.

Introduction: S-functions

S-functions are the way in which a dynamical system is represented inside Simulink environment. Every Simulink object (block, composition of blocks, schemes) is a dynamical system; hence, every object (included the entire simulation scheme) is represented by an S-function.
Typically, the built in Simulink blocks are associated with precompiled S-functions, while user added S-functions are obtained joining blocks or directly written in a language such as Matlab, C, C++, Fortran.

†Professor, Associate Fellow AIAA
‡Research Assistant Professor, currently with Department of Mechanical and Aerospace Engineering, West Virginia University, Morgantown, WV, USA

Whatever the language in which the S-function is written in, it must interact with Simulink in the same way, that is, it must have the structure shown in [1].

Different ways to build S-functions

Joining together Simulink blocks

Perhaps the most common way to build dynamical systems (hence S-functions) in Simulink, is just to join together Simulink blocks. For example, if we have two blocks that represent two systems and we would like to have the system consisting of the series of the two, the easiest way to obtain the series system is just drawing a wire that connects the output of the first system to the input of the second one.

We can then group the two systems together using the Simulink "Group" function in order to obtain a block that appears as any Simulink block and represents our "new" dynamical system.

The ease of operations like the one depicted above has been maybe the principal cause of Simulink success as an interface to deal with dynamical systems, indeed in this way we can build a dynamical system straightforwardly without the need to write a single line of code.

In some cases however assembling subsystems may not be the best choice, this is true especially if we already have the input-state-output equations of the system that we are going to build, and especially for nonlinear multivariable systems.
Let's consider for example a system in which the output depends from the state vector in this way:

$$\boldsymbol{y} = C(\boldsymbol{x})D(\boldsymbol{x})\boldsymbol{x}$$

where each entry of the matrices C and D is a given function of $\boldsymbol{x}$. If we want to obtain $\boldsymbol{y}$ by connecting some blocks that transform the signals $\boldsymbol{x}$ into $\boldsymbol{y}$, then the only solution is to do the multiplication in a wire by wire fashion, if both C and D are 2 by 2 matrices we have:

$$\begin{aligned} y_1 &= (\boldsymbol{C}_{11}(\boldsymbol{x})\boldsymbol{D}_{11}(\boldsymbol{x}) + \boldsymbol{C}_{12}(\boldsymbol{x})\boldsymbol{D}_{21}(\boldsymbol{x}))\boldsymbol{x}_1 \\ &\quad + (\boldsymbol{C}_{11}(\boldsymbol{x})\boldsymbol{D}_{12}(\boldsymbol{x}) + \boldsymbol{C}_{12}(\boldsymbol{x})\boldsymbol{D}_{22}(\boldsymbol{x}))\boldsymbol{x}_2 \\ y_2 &= (\boldsymbol{C}_{21}(\boldsymbol{x})\boldsymbol{D}_{11}(\boldsymbol{x}) + \boldsymbol{C}_{22}(\boldsymbol{x})\boldsymbol{D}_{21}(\boldsymbol{x}))\boldsymbol{x}_1 \\ &\quad + (\boldsymbol{C}_{21}(\boldsymbol{x})\boldsymbol{D}_{12}(\boldsymbol{x}) + \boldsymbol{C}_{22}(\boldsymbol{x})\boldsymbol{D}_{22}(\boldsymbol{x}))\boldsymbol{x}_2 \end{aligned} \quad (1)$$

For instance, if we use the two blocks *C(x)* and *D(x)* to construct the respective matrices, the resulting Simulink diagram is:

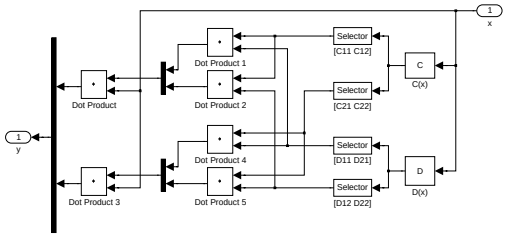

Figure 1. Matrix Multiplication Example

Needless to say, this scheme is not general nor it is easily adaptable if the dimensions involved vary, and especially if they rise above 3 or 4. Where does all this complexity come from?

The main problem in this case is that, although the wires at the output of the blocks *C(x)* and *D(x)* carry all the four elements of the two matrices, when we have to multiply the two matrices we have to decompose them into rows or columns. In conclusion, even though constructing systems by connecting blocks has in general several advantages, in some cases (especially when we have to handle nonlinear multivariable systems) it gets very difficult.

Writing S-functions as M-files

Another option is to write directly the S-function that we need using Matlab language. Since Matlab notation is very similar to the standard mathematical notation, this method, when properly used, naturally leads to the most elegant and compact way to describe dynamical systems. For example, in the case of the system considered above, we could write the following M-file:

```
function
[y,x0,st,ts]=mysys(t,x,u,flag)
if flag==0       % initial info
%  0 continue and discrete states
%  2 inputs, 2 outputs
   y=[0,0,2,2,0,0,1];
   x0=[];
   st=[];
   ts =[0 0];    % sample time
elseif flag == 3 % outputs
   C=… {compute C(x)}
   D=… {compute C(x)}
   y=C*D*x;
else
   y=[];
end
```

Without going into the details we will just point out that once we have *C(x)* and *D(x)* computed and stored in the variables *C* and *D*, it takes just the simple expression

```
y=C*D*x;
```

to compute the output. If we save this file as "mysys.m" then we can use it just by specifying the name "mysys" in the mask of the predefined Simulink S-function block:

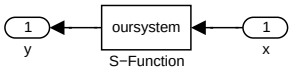

Figure 2. Simulink S-function block

This block will then be equivalent to the scheme in Figure 1. Although we no longer "see" how the system is built directly from the Simulink scheme, when dealing with complex systems this method seems to be by far the simplest one. Writing S-functions as M-files has only one serious drawback: since we use Matlab language the Matlab interpreter has to be called to evaluate each row of the S-function at every simulation step, this fact prevents the scheme from being used with Real Time Workshop, (which is the standard real time code generator for Matlab) and, last but not least, may slow down the simulation from 10 to 50 times.
There exist compilers that translate M-files to C-files and then to mex (Matlab executable) files, but unfortunately none of them (yet) produce a code that can be used with Simulink, so the only way in which we can use such compilers is the following:

1. write the main S-function as a frame that in turn calls other subfunctions written as M-files themselves.
2. use the compiler on these subfunctions.

This method can help to speed up the simulation when the subfunctions carry out a significant amount of computation, but since we still need to call the interpreter, we can't use Real Time Workshop yet.

Writing S-functions as C-files

Instead of using Matlab language, we could write the S-function in C language directly, following the "Simulink level 2" conventions [1]. In this way, the S-function could be compiled so that it behaves as a built-in Simulink block, so the interpreter no longer needs to be called, Real Time Workshop can be used, and the simulation is as fast as possible.

Unfortunately, writing an S-function as a C file is in no way as simple as writing it as an M-file, and requires, other that a good C programming knowledge, a careful study on how Simulink works, on how to pass variables and parameters from the Simulink block to the inner s-function, and so on. But the biggest problem is that the Matlab language,

and its types are no longer available so we cannot rely on indexing, built in functions, and above all we cannot rely on matrix operations.

In the above example, the only way for us to perform the output computation with a line of code as simple as it was in the Matlab case, i.e.:

```
y = C*D*x;
```

would be to define a class matrix, a subclass vector, and overload the operator *, otherwise we have to write a function mat_product that performs the matrices multiplication, and write something like:

```
mat_product(tmp1,D,x,2,1);
mat_product(y,C,tmp1,2,1);
```

the same is true if we want to perform in C any other Matlab built in operation.

In conclusion, writing the S-function in C it is often the best thing from the performance point of view but unfortunately it's not very practical and requires, some programming knowledge and a considerable amount of work.

Vectorization Blocks

Main Point

The signals that flow through Simulink wires, which in general are the only quantities allowed to vary with time, can be **scalars** or **vectors**.

It is not permitted to carry a matrix through a wire, namely we could certainly carry a p by n matrix as a vector of p*n entries (as we do in Figure 1 with the matrices $C(x)$ and $D(x)$) but all the built in blocks that will process this vector will still treat it just like a vector, so every time we have to process the p by n matrix we have to decompose it into its elements. These decompositions, and the successive compositions make it very uncomfortable to deal with varying matrices in the Simulink environment. All this could be avoided if we had some blocks that would be capable to deal with matrices without decomposing them into elements.

This is exactly the main idea behind the "vectorization blocks". Each input of them is a vector that represents a matrix ordered columnwise according to dimensions provided through its mask, (so row vectors, column vectors and even scalars are just a particular case), and the output is another vector that represents the "output matrix", which is the result of some operations on the input matrices, (product, transposition, norm, pseudo-inversion and so on).

These blocks are based on S-functions written in C (following the Simulink 3 optimized "level 2" conventions) and then compiled so that their execution is as fast as if they were built in blocks. We will present the blocks for matrix transposition and matrix multiplication, since they can cover the implementation of the majority of the control systems. The blocks for pseudo inversion, inversion, norm, determinant, (the inversion block would very useful for simulating non-rigid bodies), are currently under construction, that is, these blocks exists but they are not yet usable within real-time workshop.

Multiplication Block

The first block, which is based on the S-function "vmult.c", performs matrix multiplication. This block accepts as inputs two vectors each one representing a matrix whose elements are taken columnwise, and accepts as parameters 3 numbers:

1. n = number of rows of the first matrix.
2. p = number of columns of the first matrix = number of rows of the second one.
3. m = number of columns of the second matrix.

The two matrices are then reconstructed from the two vectors, and the first one is (left) multiplied by the second one, the elements of the n by m resulting matrix are then reordered columnwise into the output vector. The following picture shows the appearance of the block:

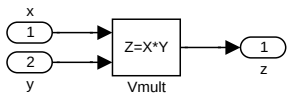

Figure 3. Multiplication Block

By double clicking on the block the following mask appears:

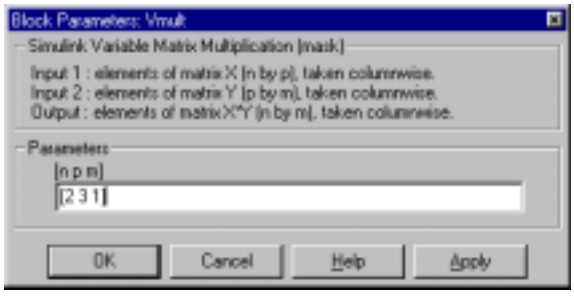

Figure 4. Multiplication Block Mask

The vector [n p m] contains the dimensions that the function needs to build the matrices from the inputs.

Transposition Block

The second block, which is based on the S-function "vtrsp.c", performs matrix transposition. This block accepts as input one vector representing a

matrix whose elements are taken columnwise, and accepts as parameters 2 numbers:

1. n = number of rows of the matrix.
2. m = number of columns of the matrix.

The matrix is then reconstructed from the two vectors, and then the elements of its transpose are reordered columnwise into the output vector. The following picture shows the appearance of the block:

Figure 5. Transposition Block

By double clicking on the block the following mask appears:

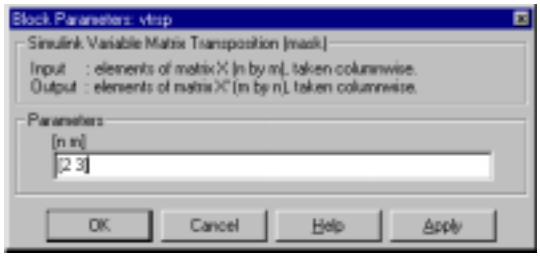

Figure 6. Transposition Block Mask

The vector [n m] contains the dimensions that the function needs to build the matrix from the input.

Redundancy

Knowing the number of inputs, the number of dimensions to be typed in the masks of both blocks is redundant by 1. We choose to retain the present level of redundancy because it allows double-checking and therefore helps preventing errors in connecting the blocks to one another.

Implementation Code

Without going into the details on how to write an S-function in C, we think it could be useful to have a quick look at the heart of the two S-functions: the sub-function "mdlOutputs", which computes the output from the values of inputs.

Vmult.c

Here is the code of mdlOutput for the S-function vmult.c:

```
static void
  mdlOutputs(SimStruct   *S,   int_T
tid)
{
int_T   i, j, k, *n
  =ssGetIWork(S);
real_T *y
  = ssGetOutputPortRealSignal(S,0);
InputRealPtrsType up1
  =
ssGetInputPortRealSignalPtrs(S,0);
InputRealPtrsType up2
  =
ssGetInputPortRealSignalPtrs(S,1);
for(i = 0; i < n[0]; i++)
   for(j = 0; j < n[2]; j++)

for(y[i+j*n[0]]=0,k=0;k<n[1];k++)

y[i+j*n[0]]+=(*up1[i+k*n[0]])*

(*up2[k+j*n[1]]);
}
```

A quick explanation of the code is necessary. Integers i, j, k, are indexes, n is a pointer at the three elements vector of the dimensions, y is a pointer at the output vector, up1 and up2 are pointers at the input vectors. The three for loops perform a matrix multiplication, in particular the first two loops select the row and column of each element of the output matrix, and the inner loop performs the scalar product between the relative row of the left matrix and the relative column of the right matrix.

It is important to notice that we are deliberately using a fast indexing that allows us to handle matrices directly in their vectorized form without actually losing computation time on array construction. Thus the code is very optimized versus speed.

Vtrsp.c

Here is the code of mdlOutput for the S-function vtrsp.c:

```
static void
  mdlOutputs(SimStruct   *S,   int_T
tid)
{
int_T  i, j, k, *n
  = ssGetIWork(S);
real_T  *y
  = ssGetOutputPortRealSignal(S,0);
InputRealPtrsType  up
  =
ssGetInputPortRealSignalPtrs(S,0);
for(i=0;i<n[0];i++)
   for(j=0;j<n[1];j++)
       y[j+i*n[1]]=(*up[i+j*n[0]]);
}
```

Integers i, j, k, are indexes, n is a pointer at the two elements vector of dimensions; y is a pointer at

the output vector, up points to the input vector. The two for loops perform the matrix transposition just by exchanging indexes. As in the preceding case, matrices are handled directly in their vectorized form.

Examples and Considerations

Two multiplications

Let us reconsider the case in which we have the output depending from the state vector in this way:

$\boldsymbol{y} = C(\boldsymbol{x})*D(\boldsymbol{x})*\boldsymbol{x}$

we can construct this function using the Varying Matrices Multiplication Block in this simple way:

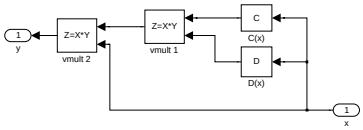

Figure 7. Two matrix Multiplication with vmult

A quick comparison between Figure 1 and Figure 7 is sufficient to appreciate the simplification; moreover, this new Simulink scheme is completely general, that is, if we change the dimensions of the matrices involved, we only have to write the new dimensions into the two Varying Matrices Multiplication Blocks.

MIMO Adaptive Sigmoidal Neural Network

The following example is more complex, here we used the two blocks to implement a continuous time adaptive multilayer MIMO sigmoidal neural network [2], [3], [4] which is represented by the following equation:

$$\begin{bmatrix} \dot{N} \\ \dot{M} \\ v_{ad} \\ \bar{v} \end{bmatrix} = \begin{bmatrix} -G[\bar{x}\zeta M^T \sigma'(N^T\bar{x}) + \lambda|\zeta|N] \\ -F[(\sigma(N^T\bar{x}) - \sigma'(N^T\bar{x})N^T\bar{x})\zeta + \lambda|\zeta|M] \\ M^T\sigma(N^T\bar{x}) \\ K_v(\|Z\|_F + \bar{Z} + K_Z)\zeta \end{bmatrix}$$

There are two inputs to the net, the first is x, which should contain information necessary to the net to approximate a function $\Delta(x)$, and the second is ζ, which is the "error" term that actually drives the network.

The network output is composed of two vectors: the adaptive term v_{ad} and the robustifying term $\bar{v}$, which is made proportional to ζ. The network states are exactly the interconnection weights between different layers, namely all the entries in the two synaptic matrices M (connecting hidden layer to output layer) and N (connecting input layer to hidden layer). The output of the hidden layer is the vector

$$\sigma(N^T\bar{x}) = [1 \quad \sigma_1(n_1\bar{x}_1) \quad \ldots \quad \sigma_h(n_h\bar{x}_h)]^T \qquad (2)$$

where $n_1 \ldots n_h$ are the rows of N^T and σ is the activation function, which is chosen to be:

$$\sigma_i(z) = \frac{1}{1+e^{-az}} \qquad (3)$$

The matrix $\sigma'(N^Tx)$ is the Jacobian of $\sigma(N^Tx)$ with respect to N^Tx evaluated at N^Tx. Z is defined as diag(M,N), and $|Z|_F$ is its Frobenius norm. All other values are constants. Specifically F and G are the learning rates of the first and second layers, and λ is commonly known as the e-modification gain. Constants required for the robustifying term are K_v and K_z. Finally, $\bar{Z}$ is an upper bound for $|Z|_F$.
The following picture, Figure 8, shows the realization of the net as a Simulink scheme.

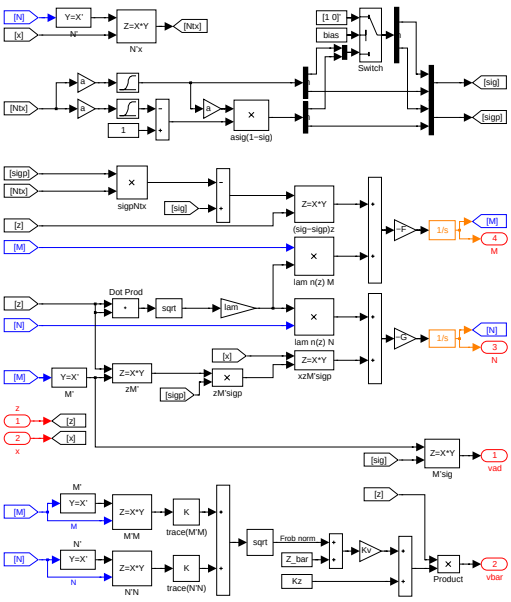

Figure 8. Neural Network Scheme

The upper part of the scheme is devoted to the computation of the hidden layer output (4) and its jacobian matrix. The central part is the heart of the function in that it computes the derivatives of the states N and M, and then integrates them. Finally the lower part of the scheme contains the blocks for the output computation, which involves also the computation of the Frobenius norm of Z.

This scheme has been successfully used to simulate in real time several neural network based

controls [2], [3], although slightly faster, its version as a C program was 420 lines long and had several limitations, a full-general and optimized version would probably be much longer.

Rotation Matrices Library

The multiplication and transposition blocks were used to build a library of rotation matrices, based both on Euler angles and quaternions. These matrices are a classical example of very required varying matrices blocks, since they are very useful for the simulation of rigid bodies in a full 3D environment.
Since their size is fixed, and the elements are just 9, they can be easily implemented without the need of the blocks presented in the paper, but the use of these blocks can make their implementation extremely fast and straightforward.
The picture below shows the rotation matrix that transforms body fixed coordinates into earth fixed coordinates, given the orientation of the body with respect to earth in roll, pitch and yaw angles [5].

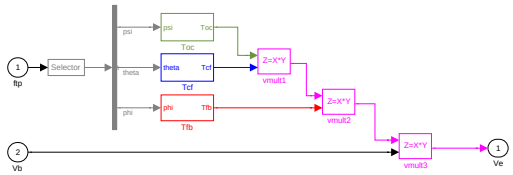

Figure 10. Body2Earth Rotation Matrix

The scheme above is clearly related to the formula:

$$ {}^{e}R_{b} = \mathrm{e}^{\psi S\left(\begin{bmatrix}0\\0\\1\end{bmatrix}\right)} \mathrm{e}^{\vartheta S\left(\begin{bmatrix}0\\1\\0\end{bmatrix}\right)} \mathrm{e}^{\phi S\left(\begin{bmatrix}1\\0\\0\end{bmatrix}\right)} $$

The Multiplication, Transposition, and Inversion blocks, are currently being used by the first author in a recursive frequency domain identification scheme to be used for on-line flight parameter estimation and self adaptive flight control.

Modularity issues

In our opinion, there's also another fact, less evident but very important, that makes a totally Simulink-based implementation best than one which is partially coded in Matlab or C.
Often, the persons who designs a Simulink scheme, has barely taken a quick computer programming course in his/her studies, so it's pointless to expect that he/she can write efficient code with good modularity and reusability characteristics.
In these cases, a totally Simulink-based project generally retains better modularity and reusability characteristics than a partially coded project.
Since as project complexity grows modularity becomes critical, this fact alone can, in our opinion, become the most compelling reason to adopt a totally Simulink-based implementation.

Where to find the blocks

The four blocks, along with all the examples in the paper and instructions files for their use, can be freely downloaded at the official mathworks ftp site:
ftp://ftp.mathworks.com/pub/contrib/v5/simulink/matrix_library

DSP Blockset

If you have the DSP toolbox, then you can use the DSP matrix library, which is very efficient and complete. The presented blocks are fully compatible with the ones of the DSP matrix library, and of course all the considerations above on the opportunity of using the presented blocks, applies as well to the blocks in the DSP matrix library.

Conclusions

In this paper, we briefly explained Simulink's S-functions mechanism, which is used to represent and simulate dynamical systems, in particular we have shown the three common methods to build S-function, and underlined that in some cases neither of them can be at the same time easy and suitable for real time simulations.
Two Simulink blocks were proposed that solve this problem by allowing varying matrices to be handled directly in Simulink environment. Their aspect, their use, and the most important part of their code were explained, and some examples were given.
Finally some good reasons that render the use of these blocks highly desirable were given.

References

[1] Matlab 5.3 and Simulink 3 manuals, The Mathworks Inc.

[2] G. Campa, M. Sharma, A.J. Calise, M. Innocenti, "Neural Network Augmentation of Linear Controllers with Application to Underwater Vehicles" *American Control Conference 2000*, June 2-4, 2000 San Diego.

[3] F. Nardi, R.T. Rysdyk, A.J. Calise, "Neural Network Based Adaptive Control of a Thrust Vectored Ducted Fan," *AIAA Guidance, Navigation and Control Conference*, AIAA 99-3996, Portland, OR, 1999.

[4] F.L. Lewis, S. Jaganathan, A. Yesilderek, *Neural Network Control of Robot Manipulators and Nonlinear Systems*, Taylor & Francis, London, 1999, pp. 150-170.

[5] Guidance and Control of Ocean Vehicles, Thor J. Fossen John Wiley and Sons 1994.