TS3ph Block Library Conversion Guide

Adam Sumner
Illinois Institute of Technology

1 Overview

This document will act as a guide to follow during the conversion of the models present in the source files. Previously, industry-standard machine and control models existing in Siemens PSS/E have been converted from their respective block diagrams (frequency domain) to a state-space formulation (time domain). From here, each model was carefully analyzed, and a set of differential equations governing the system's behavior was produced. In each model, common blocks may be encountered (e.g. Lag blocks and Lead-Lag blocks). A block library has been developed to streamline the process of deriving the equations from these common blocks. This library will be used in the ResidualFunction files as well as the ResidualJacobian files of the source code. This guide will not cover the process of deriving the state-space equations, but it will instruct on how to successfully implement the code to do so. This document will demonstrate this process through example.

2 Block Function Evaluation

This section will explain how each type of block is implemented in the block library to familiarize you with the library's applications to this project.

2.1 BlockInfo Structure

To create an abstraction of a control block in its respective block diagram, we have developed a structure. For those unfamiliar with C programming, a structure is simply a group of related variables put under a common name. Because C is a procedural language, it is impossible to create objects like we are able to do in Java or C++. This is why structures can be extremely

useful because it allows the use of objects without using an object oriented language. The structure used in this project is called `BlockInfo`. Its implementation is shown below:

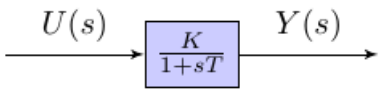
```
typedef struct{

   PetscScalar *input;     // Input signal
   PetscScalar *state;     // Internal state value
   PetscScalar *state_dot; // Petsc time term
   PetscScalar output;

   // Block parameters
   PetscScalar *K;
   PetscScalar *Tden;
   PetscScalar *Tnum;
   PetscInt *nmode; //Network mode

   // Limit Parameters
   PetscScalar *lmax;
   PetscScalar *lmin;

}BlockInfo;
```

As you can observe in the code, this structure has ten different variables associated with itself. These are the input signal, the value of the internal state variable, the Petsc time term, the output signal, a constant K, a constant in the denominator, a constant in the numerator, the network mode, the maximum value of the limit, and the minimum value of the limit. This structure acts as a generic block so do not expect to assign each variable of every BlockInfo structure a value. The values used in this structure will be completely dependent on the block diagram that is being converted.

2.2 Block Library

Currently there are three common types of blocks denoted as type A, B, and C illustrated below in each respective section. Please refer to **TS3ph Dynamic Device Modeling Project Instructions** *Part 1: Conversion of Block Diagrams to State-Space Formulation* for more information on block types. Before you can evaluate the block, it is necessary to first identify the type of block, along with declaring the parameters of the block you are analyzing. Each subsection goes through a step by step process on how to use the library with each type of block.

2.2.1 Block-A



$$\dot{y} = \frac{1}{T}(Ku - y)$$

Figure 1: Block-A in the Frequency Domain [1]

We must first initialize our block using the `BlockInfo` structure. For example, I will initialize a new block by typing:

```
BlockInfo blockA;
```

For block-A it is necessary to define the following parameters:

```
blockA.input = &inputA;
//the input signal

blockA.state = &xgen_arr[A_idx];
//the state variable for this block

blockA.state_dot = &xdot_arr[A_idx];
// the derivative of the state variable

blockA.K = &MODEL->K[idx_modeltype];
//The constant K in the numerator of the block
diagram (refer to Figure 1)

blockA.Tden= &MODEL->T[idx_modeltype];
//The value of T in the denominator (refer to Figure
1)

blockA.nmode = &Gen->net_mode;
```

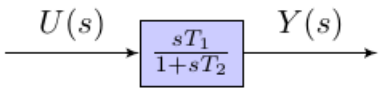
Depending on the block you are working with, the specific variables used to initialize the Block-A structure will be different. I have used a generic form of how each variable is labeled. The variables you will be working with may not look the same. After the structure has been given these values, we can now use the function `fvalBlockA` to generate code in the

ResidualFunction source file to accurately model this block in state-space. This is done by typing:

```
fgen_arr[A_idx] = fvalBlockA(&blockA);
```

Once this has been completed, the block analyzed of type A has successfully been converted from it's frequency-domain representation to the state-space representation in the ResidualFunction file of the model you are working on.

2.2.2 Block-B



$$\begin{aligned} \dot{x} &= \frac{1}{T_2}(T_1 u - x) \\ y &= \frac{1}{T_2}(T_1 u - x) \end{aligned}$$

Figure 2: Block-B in the Frequency Domain [1]

We must first initialize our block using the `BlockInfo` structure. For example, I will initialize a new block by typing:

```
BlockInfo blockB;
```

For block-B it is necessary to define the following parameters:

```
blockB.input = &inputB;
//the input signal

blockB.state = &xgen_arr[B_idx];
//the state variable for this block

blockB.state_dot = &xdot_arr[B_idx];
// the derivative of the state variable

blockB.K = PETSC_NULL;
//No constant value

blockB.Tnum = &MODEL->Tn[idx_modeltype];
//The value of T_1 in the numerator (refer to Figure 2)
```

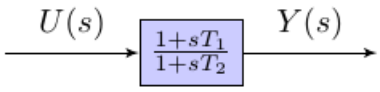
```
blockB.Tden= &MODEL->Td[idx_modeltype];
//The value of T2 in the denominator (refer to Figure
2)

blockB.nmode = &Gen->net_mode;
```

Depending on the block you are working with, the specific variables used to initialize the Block-B structure will be different. I have used a generic form of how each variable is labeled. The variables you will be working with may not look the same. After the structure has been given these values, we can now use the function `fvalBlockB` to generate code in the ResidualFunction source file to accurately model this block in state-space. This is done by typing:

```
fgen_arr[B_idx] = fvalBlockB(&blockB);
```

Once this has been completed, the block analyzed of type B has successfully been converted from it's frequency-domain representation to the state-space representation in the ResidualFunction file of the model you are working on.

2.2.3 Block-C



$$\begin{aligned} \dot{x} &= u - \tfrac{1}{T_2}(T_1 u + x) \\ y &= \tfrac{1}{T_2}(T_1 u + x) \end{aligned}$$

Figure 3: Block-C in the Frequency Domain [1]

We must first initialize our block using the `BlockInfo` structure. For example, I will initialize a new block by typing:

```
BlockInfo blockC;
```

For block-C it is necessary to define the following parameters:

```
blockC.input = &inputC;
//the input signal

blockC.state = &xgen_arr[C_idx];
//the state variable for this block

blockC.state_dot = &xdot_arr[C_idx];
// the derivative of the state variable

blockC.K = PETSC_NULL;
//No constant value

blockC.Tnum = &MODEL->Tn[idx_modeltype];
//The value of T1 in the numerator (refer to Figure 3)

blockC.Tden= &MODEL->Td[idx_modeltype];
//The value of T2 in the denominator (refer to Figure
3)

blockC.nmode = &Gen->net_mode;
```

Depending on the block you are working with, the values used to initialize the Block-C structure will be different. I have used a generic form of how each variable is labeled. The variables you will be working with may not look the same. After the structure has been given these values, we can now use the function `fvalBlockC` to generate code in the ResidualFunction source file to accurately model this block in state-space. This is done by typing:

```
fgen_arr[C_idx] = fvalBlockC(&blockC);
```

Once this has been completed, the block analyzed of type C has successfully been converted from it's frequency-domain representation to the state-space representation in the ResidualFunction file of the model you are working on.

2.3 Example: ResidualFunctionEXST1

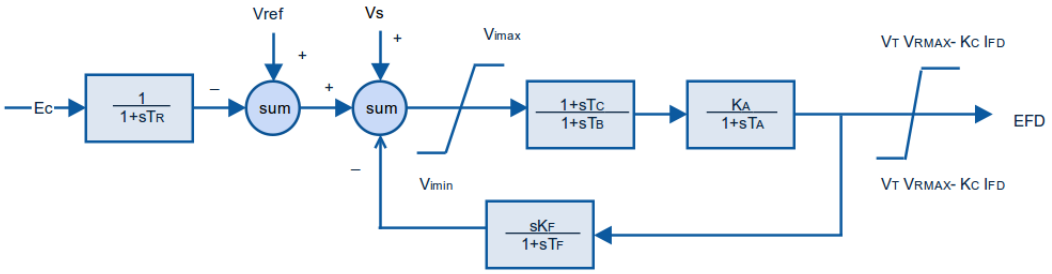


Figure 4: EXST1 Block diagram

Above is the block diagram for exciter model EXST1. In this section I will convert this diagram into C code for use with the ResidualFunction file. The entire model will not be converted, but I will walk through enough of this diagram to demonstrate the key concepts needed to convert further models. As with any block diagram, we start with the initial signal Ec. In our implementation, we name this input `Ecomp`. We also have labeled the first block's internal state variable as E_T. From the block diagram, we can easily conclude that this block is of type A. This means that we need to enter values for six variables in the structure. We do so by typing the following code:

```
BlockInfo blockEt;

blockEt.input = &Ecomp;
blockEt.state = &xgen_arr[Et_idx];
blockEt.state_dot = &xdot_arr[Et_idx];
blockEt.K = PETSC_NULL;
blockEt.Tden = &EXST1->Tr[idx_exc];
blockEt.nmode = &Gen->net_mode;
```

As explained earlier in Section 2.2.1, this is the code that will complete our abstraction of the first block encountered in our source code. From here, we must now evaluate the block by typing:

```
fgen_arr[Et_idx] = fvalBlockA(&blockEt);
```

This completes our code for the first block. The next elements evaluated in the block diagram are then the two summation blocks. These blocks are evaluated using simple arithmetic. In our case, there is a feedback signal

associated with one of the blocks so we must calculate this output in order to proceed further. In our implementation, we denote the output of the feedback block as V_F. This is calculated using the equation $V_F = \frac{K_F}{T_F V_R} - R_F$ where K_F is the constant in the numerator of the feedback block, T_F is the constant in the denominator, V_R is the output of the block of type A before the limiter , and R_F is the state variable associated with the feedback block. Please refer to Figure 2 for how this equation was derived. The equivalent source code for this expression would be:

```
Vf = (Kf/Tf*Vr - Rf);
```

Once this value is calculated, we can now evaluate the summation blocks. These blocks are simple. If the signal going into the block has a $-$ sign, then we subtract this signal. If the signal has a $+$ sign, then we add the signal. For EXST1, the summation blocks would look like:

```
Vsum = Vref - Et + Vothsg + Vuel + Voel - Vf;
```

where $V_s = V_{othsg} + V_{uel} + V_{oel}$. After we calculate the summation blocks, we encounter a limiter. To denote if the signal hit or did not hit any limits, we use a flag variable. If the signal did not hit any limits, it is set to 0. If it hit the upper limit, it is set to 1, and if it hit the lower limit then it is set to -1. If the value of the signal is beyond the maximum or minimum limit, it is necessary to set the signal value to either the max or min value depending on the situation. The signal is set to the max limit value if it went above the max limit, and it is set to the minimum limit value if the signal is below the min limit. The equivalent code for the limit encountered in the diagram would then be:

```
PetscInt Vsum_flag = 0; // Default value
if(Vsum>Vimax){
   Vsum = Vimax;
   Vsum_flag = 1;
}
if(Vsum<Vimin){
   Vsum = Vimin;
   Vsum_flag = -1;
}
```

The next block encountered in the diagram is of type C. With this knowledge, we know we will be entering seven variables for our structure. The state variable for this block is V_{LL} and its input is `Vsum`. We code this block as follows:

```
BlockInfo blockVll;

blockVll.input = &Vsum;
blockVll.state = &xgen_arr[Vll_idx];
blockVll.state_dot = &xdot_arr[Vll_idx];
blockVll.Tden = &EXST1->Tb[idx_exc];
blockVll.Tnum = &EXST1->Tc[idx_exc];
blockVll.nmode = &Gen->net_mode;
blockVll.K = PETSC_NULL;

fgen_arr[Vll_idx] = fvalBlockC(&blockVll);
```

As explained earlier in Section 2.2.3, this is the code that will complete our abstraction of the second block encountered in our source code. Before we go onto the third block, we check if the constant in the denominator is 0. If it is, the value of the output of the block will simply be the state variable `Vll`, otherwise we use `blockVll.output`. A variable called `aux_Vr` was created to act as the signal in between these two blocks. The code for this is:

```
PetscScalar aux_Vr;
if(Tb == 0)
   aux_Vr = Vll;
}else{
aux_Vr = blockVll.output;
}
```

Moving onto the third block, we notice it is similar to the first block we encountered, the only difference being the number in the numerator. The state variable for this block is V_R and its input is `aux_Vr`. The code for this would be:

```
BlockInfo blockVr;

blockVr.input = &aux_Vr;
blockVr.state = &xgen_arr[Vr_idx];
blockVr.state_dot = &xdot_arr[Vr_idx];
blockVr.K = &EXST1->Ka[idx_exc];
blockVr.Tden = &EXST1->Ta[idx_exc];
blockVr.nmode = &Gen->net_mode;

fgen_arr[Vr_idx] = fvalBlockA(&blockVr);
```

*Note that for the variable K in the first block, we set it to `PETSC_NULL` whereas in this case, we set this value to `Ka`. This is because the first block

encountered has a numerator value of 1 where this block has a numerator value of the variable `Ka`.

For simplicity's sake, the rest of the diagram will not be shown. Both the feedback block, the remaining limiter, and the output EFD must still be converted into C code. Please refer to current source code of Model EXST1 for more information.

3 Jacobian Library

This section will explain how to generate the jacobian structure for the model you will be converting. It is meant to help familiarize you with the library and its application to the project.

3.1 Signl Structure

For the ResidualJacobian function, we have developed a structure called `signl` that is an abstraction of individual signals when evaluating the jacobian structure. The implementation of this structure is shown below:

```
typedef struct {
   PetscInt n;
   PetscInt idx[SIGNAL_SIZE];
   PetscScalar gain[SIGNAL_SIZE];
}signl;
```

In our implementation, `n` tells us how many signals are within the structure, `idx[SIGNAL_SIZE]` tells us the state index of the signal, and `gain[SIGNAL_SIZE]` tells us the gain of the signal. Note that both the state index and the gain are arrays. This is because when we add more signals to the structure, specific indices of each array will correspond to each signal. Currently the SIGNAL_SIZE is defined to be 20. It is important to recognize the difference between the structure `signl` and an individual signal. **The structure is one piece of data that contains individual signals.**

3.2 Jacobian Library Functions

We have developed several functions to aid in writing entries into the jacobian matrix. Each type of block has its own jacobian functions associated with itself. Before we introduce the specific functions, we'll

introduce the general `signl` tools. This set contains the following functions:

1. `initializeSignal(signl *sig)`
2. `addSignal(signl *sig, PetscInt svar, PetscScalar gain)`
3. `addSignalVoltage(signl *sig, PetscInt idx_node, PetscScalar *vgen)`
4. `resetSignal(signl *sig)`
5. `joinSignal(signl *sig1, signl *sig2)`
6. `negativeSignal(signl *sig1)`

These functions will not be covered in detail in this document, however the purpose of each function will be explained.

- `initializeSignal` does as expected. It sets the number of signals in the structure to 0 and sets the state index values along with the gain values all to zero.
- `addSignal` adds a signal to the `signl` structure. It sets the value of the state index and the gain to the values passed into the function and increments the amount of signals in the structure.
- `addSignalVoltage` adds the derivative of the three-phase averaged voltage to the `signl` structure.
- `resetSignal` as it insists in its title resets the signal. It does so by assigning the variable `n` to 0.
- `joinSignal` will join two `signl` structures into one. If the sum of the individual signals contained in each structure is greater than `SIGNAL_SIZE`, then this function will do nothing. If the sum is less than `SIGNAL_SIZE`, it will combine the state index and gain arrays of the second `signl` structure with the state index and gain arrays of the first `signl` structure.
- `negativeSignal` makes all gains in the array negative.

3.2.1 Lag Block (Type-A)

The functions that are specific to the Lag Block are:

1. `LagJaco(PetscScalar Tden, PetscScalar K1, PetscInt st_idx, signl sig, PetscInt nmode, PetscScalar *val, PetscInt *col, PetscReal a, Mat J)`

2. `LagGain(signl *sig, PetscInt svar, PetscScalar Tden, PetscScalar K1)`

The purpose of `LagJaco` is to write entries into the jacobian matrix for a block of type A. In order to do this, the function requires the block parameters, a `signl` structure, and the state variable of the block. The implementation of this function will not be explained due to the complexity, however, its application will be demonstrated later in an example with exciter model EXST1. The first two parameters correspond to the values in the block diagram, in this case the variable in the denominator and the constant in the numerator respectively. The next parameter refers to the state index, and `nmode` refers to the net_mode variable. The `*val` and `*col` parameters refer to the dimensions of the jacobian structure (number of rows and columns). For the final two parameters, we pass the variable a, which is a variable of the `Gen` structure, and J.

The purpose of `LagGain` is to update the signal structure. The implementation of this function will also not be explained due to the scope of this document, however, its application will be demonstrated later. The parameters of this function are the `signl` structure being used, the state variable, the variable in the denominator, and the constant K in the numerator.

3.2.2 LeadLag Block (Type-C)

The functions that are specific to the LeadLag Block are:

1. `LeadLagJaco(PetscScalar Tden, PetscScalar Tnum, PetscInt st_idx, signl sig, PetscInt nmode, PetscScalar *val, PetscInt *col, PetscReal a, Mat J)`

2. `LeadLagGain(signl *sig, PetscInt svar, PetscScalar Tden, PetscScalar Tnum)`

The purpose of both of these functions is the same as described in Section 3.2.1. The difference between the LeadLag functions and the Lag functions are the implementation of each function and a few parameters. Since we are not going into an in depth study of each function, it is only necessary to know how to use them. Both of the LeadLag functions take nearly identical parameters, except the parameters for these functions pertain to the structure of a LeadLag block. Refer to Figure 3 and Section 3.2.1 to determine which variables to pass to the function.

3.2.3 Feedback Block (Type-B)

The functions that are specific to the Feedback Block are:

1. `derivFeedback(PetscScalar Tden, PetscScalar K1, PetscInt st_idx, signl sig, PetscInt nmode, PetscScalar *val, PetscInt *col, PetscReal a, Mat J)`

2. `derivFeedbackGain(signl *sig, PetscInt svar, PetscScalar Tden, PetscScalar K1)`

The purpose of these functions are comparable to those of both the Lag Block functions and the LeadLag block functions. As previously explained, the parameters entered are similar, but not identical. Refer to the specific block type in order to determine the correct parameters to pass to each function. `derivFeedback` accomplishes the same goal as both `LagJaco` and `LeadLagJaco`, while the function `derivFeedbackGain` also does what `LagGain` and `LeadLagGain` do in that it updates the `signl` structure.

3.3 Example: ResidualJacobianEXST1

Looking back to Figure 4, we again follow the block diagram. The first thing we must do is initialize the signal. Furthermore, we check if in the initialization the voltage is constant. The equivalent code for this would be:

```
signl signalOne;
initializeSignal(&signalFeedback);
if (nmode != 1){
   if (EXST1->ECOMP_idx[idx_exc] != -1){
      addSignal(&signalOne, Ecomp_idx, 1);
      }else{
```

```
        addSignalVoltage(signalOne, idx_node, vgen);
   }
}
```

Now that our `signl` is initialized, we can continue with the evaluation of the block. The first block is a Lag Block (Type-A) so we know we will be using `LagJaco` and `LagGain`. Applying these functions to our particular model, the parameters for each function would look like:

```
LagJaco(Tr,1,Vm_idx,signalOne,nmode, &val, &col, a,J
);
LagGain(&signalOne, Vm_idx, Tr, 1);
```

In order to evaluate the summation blocks we must first calculate the feedback block. This is done so by initializing a new `signl` for the feedback block, evaluating the block, and then joining this `signl` with our current `signalOne`. We evaluate the feedback block by typing:

```
signl signalFeedback;
initializeSignal(&signalFeedback);
addSignal(&signalFeedback, Efd_idx, 1);
derivFeedback(Tf1, Kf, Rf_idx, signalFeedback, nmode,
&val, &col, a, J);
derivFeedbackGain(&signalFeedback, Rf_idx, Tf1, Kf);
```

Once this is done, we continue with the summation block. We can see that both `signalOne` and `signalFeedback` have subtraction signs associated with their input so we join them as follows:

```
joinSignal(&signalOne, &signalFeedback);
negativeSignal(&signalOne);
```

Last we check if V_{REF} is variable and join it to the `signl` accordingly. This is done with the following code:

```
if (Gen->net_mode == 1){
   PetscInt vref_idx = Gen->gen_size +2*i;
   addSignal(&signalOne, vref_idx, 1);
}
```

Implementation of the limiter for EXST1 is still underway and will be skipped in this example. Moving on to the LeadLag block we can invoke the following code:

```
LeadLagJaco(Tb, Tc, Vll_idx, signalOne, nmode, &val,
```

```
&col, a, J);
LeadLagGain(&signalOne, Vll_idx, Tb, Tc);
```

To stay consistent with the previous example, I will show the code necessary with the third block. It is:

```
LagJaco(Ta, Ka, Vr_idx, signalOne, nmode, &val, &col,
a, J);
LagGain(&signalOne, Vr_idx, Ta, Ka);
```

Please refer to curent source code of model EXST1 to analyze the rest of the implementation. It is redundant to show it in this document.

References

[1] *TS3ph Dynamic Device Modeling Project Instructions Part 1: Conversion of Block Diagrams to State-Space Formulation.* Illinois Institute of Technology