Faster Than Real-Time (FTRT) Dynamics Simulation (Phase II)

Generic Wind Turbine Model Type 3

Ming Chen
Illinois Institute of Technology

Abstract-In the last summer, our team discovered most of the mysteries in this Generic Wind Turbine Model Type 3 (WT3) introduced by Siemens PSS®E documentation and implemented it into Simulink. The results were accurately matching the PSS®E simulation. Continue the work from the summer, our team translated the block diagram into equations and wrote pseudo-code to develop programmer's guide to help future researchers translate model into C code. Then we built Residual Functions for all the models of WT3 into C code. Finally, necessary files in the server for testing were modified or created. Although models has been implemented, problem occurred while connecting to the grid in the server. Further debugging and testing is needed.

Index Terms-Wind Turbine, Dynamics Simulation, TS3ph

I. INTRODUCTION

A. Project Description

Develop a high fidelity "faster than real-time" dynamics simulator capable of predicting complex, large-scale power system behavior based on real-time measurements. Leverage recent and proposed mathematics and computational advances (e.g., PETSc linear solvers, nonlinear solvers, time-stepping algorithms, memory management, multi-core, many-core and GPU processing) to improve speed of dynamics simulations. Leverage recent modeling and simulation advances (e.g., new three-phase unbalanced network models, single-phase induction motor models, protection system models) to improve fidelity of dynamics simulations.

The goal of "faster than real-time power system dynamics simulation" is to provide actionable information to operators during power system disturbances and cascading outages. The project is developing high performance computer modeling and simulation software for predicting the effects of disturbances faster than real-time. Based on the predictive capabilities of this research, operators will be able to respond before the full effects of a cascading outage are realized, thereby avoiding wide spread blackouts.

B. Background of Type 3 Generic Wind Turbine Model

The WT3 PSS®E wind turbine stability model was developed to simulate performance of a wind turbine employing a doubly fed induction generator (DFIG) with the active control by a power converter connected to the rotor terminals.

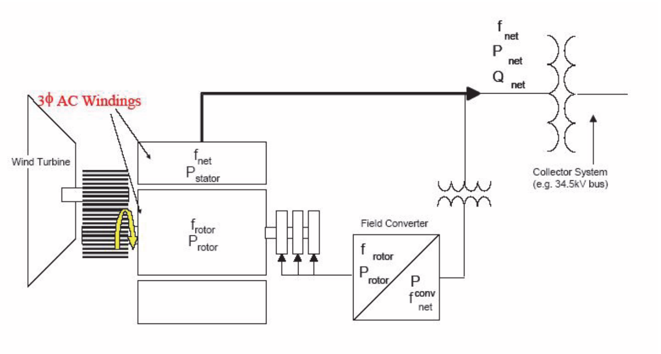


Figure 1-1. Doubly Fed Induction Generator with the Active Control by a Power Converter Connected to the Rotor Terminals

The model is to be used in studies related to the integration of Wind Turbine Generators (WTG) in an Electrical Power System.

There are four components in this model:

- WT3G: generator/converter model
- WT3E: electrical control model
- WT3T: mechanical control (wind turbine) model
- WT3P: pitch control model.

In each component model, there are many control blocks that duplicate integrators, and the controller limits also play an integral role in the dynamic performance of this wind turbine. In addition, the behavior of such models is governed by interactions between the continuous dynamics of state variables, and discrete events associated with limit.

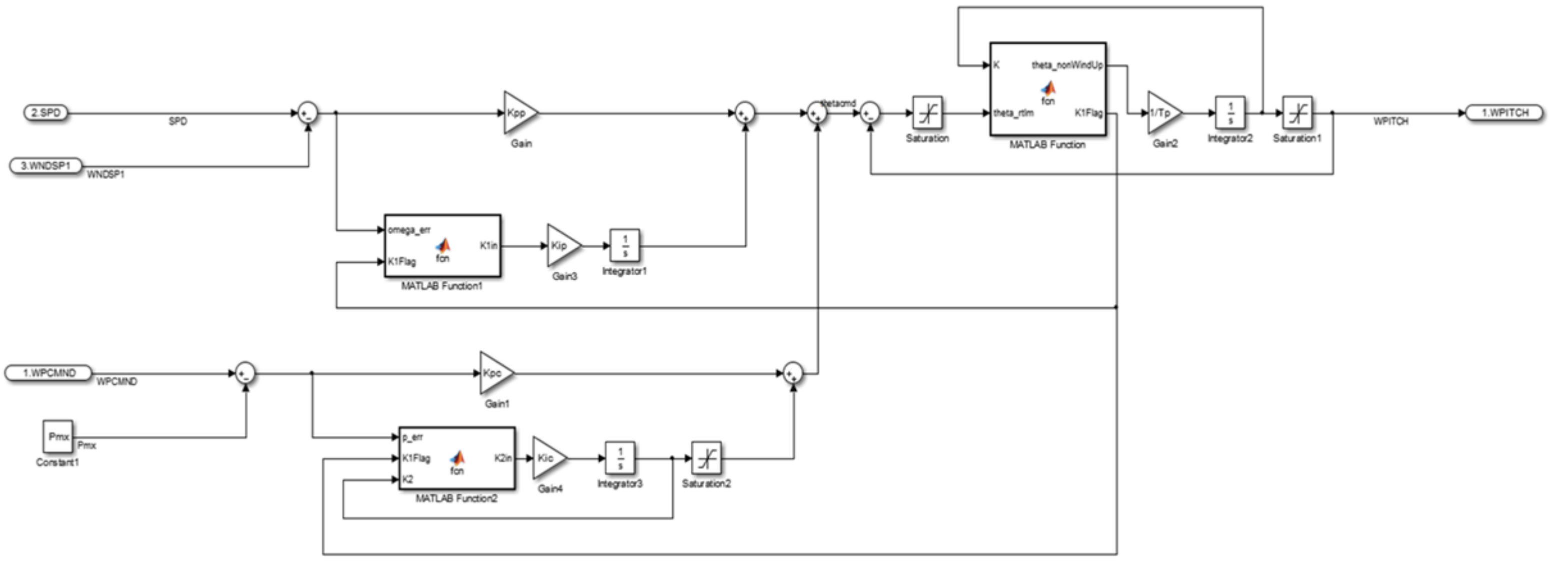


Figure 1-3. WT3P1 Simulink Model

The interaction among generic wind models are shown in Figure 1-2 below:

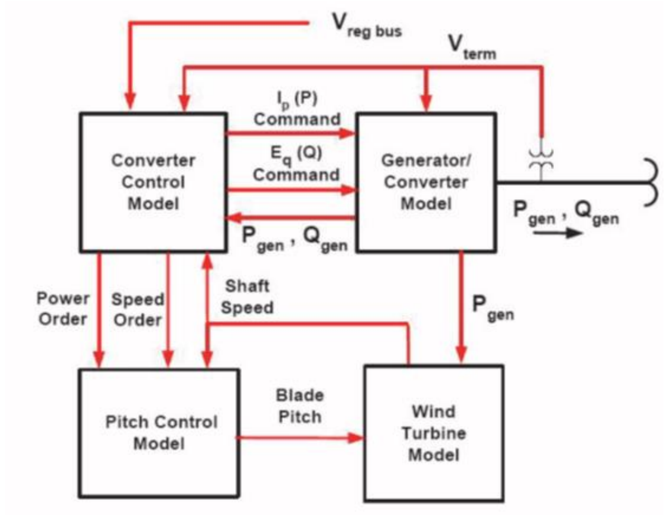


Figure 1-2. Interaction among Generic Wind Models

Since Type 3 Generic Wind Turbine Model are currently the dominant technology for new wind farm developments, they receives considerable attention and it is worthwhile to analyzing it.

C. Part of the summer work

Since the project continues remaining work from the last summer, it is necessary to show part of the summer work providing basic understanding of this project. Figure 1-3 shows how Pitch Control Model is implemented in the Simulink. About more work from the phase one, please refer to my last paper.

II. METHODS

The control device model generic wind turbine type 3 responses were benchmarked in a small test system against a commercial software tool for offline power system dynamics simulation. This small test system we used in the last summer was MATLAB. After the implementation and verification of WT3 in MATLAB Simulink, the control blocks in the model were translated into state-space equations. Then pseudo-code for the basic logic operation were build up to facilitate the further implementation in high level language.

C is the programming language used in the TS3ph program, resulting from its efficiency and simplicity. For a simulator that can show the simulation ahead of the current state, time efficiency is the biggest concern. PETSc is the computational library used to solve the differential equations. Also, Git is the version control system used by our team to cooperate with each other better. Considering the security issue, all the code and simulation are done in the secure server in school's ECE department building. Putty, emacs, GDB are the tools used throughout the project.

III. RESULTS

A. Develop Residual Function based on Simulink models

Based on the Simulink model we built in the past, we rewrote all graphical models of WT3 into equations. An example of WT3G1 lower branch is provided.

Figure 3-1 shows the original model in PSS®E documentation of WT3G1 lower branch. State K+ 2 has a non-windup limiter, which is a major challenge for implementing in Simulink and in C code.

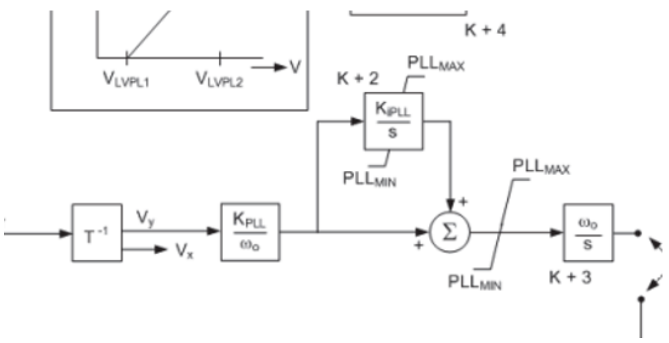


Figure 3-1. Original Model in PSS®E

Figure 3-2 shows how this portion of diagram is implemented in Simulink, which is the work from the past.

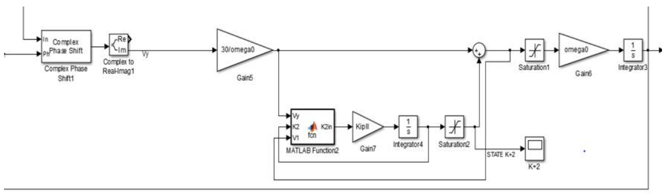


Figure 3-2. Model built in Simulink

In order to put these abstract and complicated diagrams into code, programmer's guide with should be prepared before implementation. As shown in Figure 3-3, basic logical operation like sum and normal limiter were written in a more arithmetic form. This example is corresponding to Figure 3-1 and 3-2.

Sum Function:

$$V_1 = K_{2s} + V_y * K_{pll}/omega0$$

Saturation3:

$$V_{1s} = \begin{Bmatrix} Pllmax \;\; if \; V_1 > Pllmax \\ V_1 \;\; if \;\; V_1 \leq Pllmax \; and \; V_1 \geq -Pllmax \\ -Pllmax \;\; if \; V_1 < -Pllmax \end{Bmatrix}$$

Figure 3-3. Equations for basic logical operations.

Conversion of transfer function to state-space makes the computation in C possible. Typical control blocks conversion equations are shown in Figure 3-6. For example, block 1/(1+0.02s) in Figure 3-4 was translate into the equation shown in Figure 3-5. Although it is a differential equation, PETSc solver can solve that and give us the result of the variable.

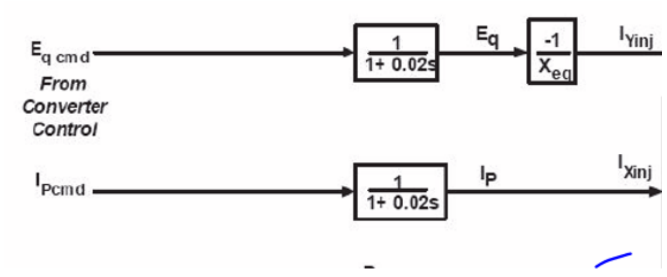


Figure 3-4. Control Blocks in WT3G1

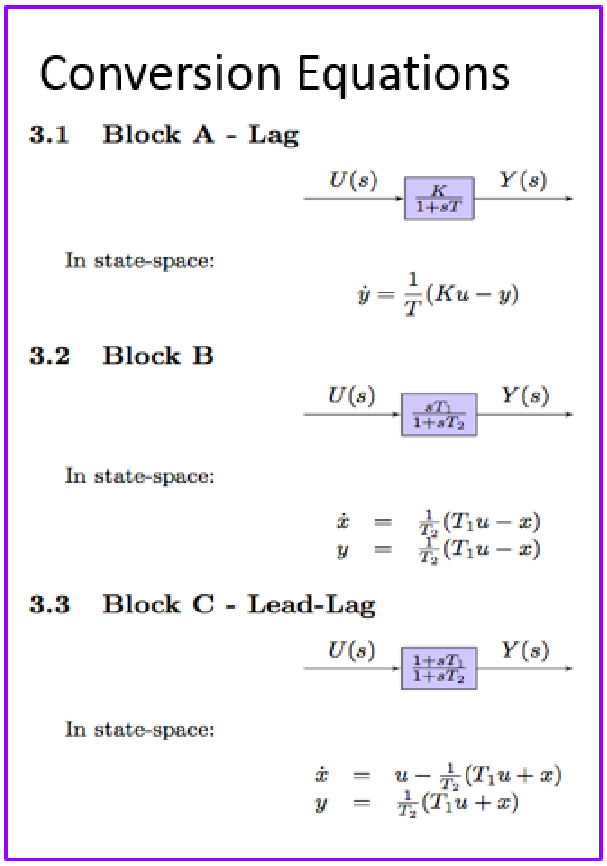
Transfer Fcn (state-space1): $\frac{1}{1+0.02s}$

$$\dot{I}_{Xinj} = \frac{-1}{0.02} I_{Xinj} + \frac{1}{0.02} I_{pcmd2}$$

Transfer Fcn2 (state-space2): $\frac{1}{1+0.02s}$

$$\dot{E_q} = \frac{-1}{0.02} E_q + \frac{1}{0.02} E_{qcmd}$$

Figure 3-5. State-space equations converted from control Blocks



Conversion Equations

3.1 Block A - Lag

$U(s)$ $\frac{K}{1+sT}$ $Y(s)$

In state-space:

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

3.2 Block B

$U(s)$ $\frac{sT_1}{1+sT_2}$ $Y(s)$

In state-space:

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

3.3 Block C - Lead-Lag

$U(s)$ $\frac{1+sT_1}{1+sT_2}$ $Y(s)$

In state-space:

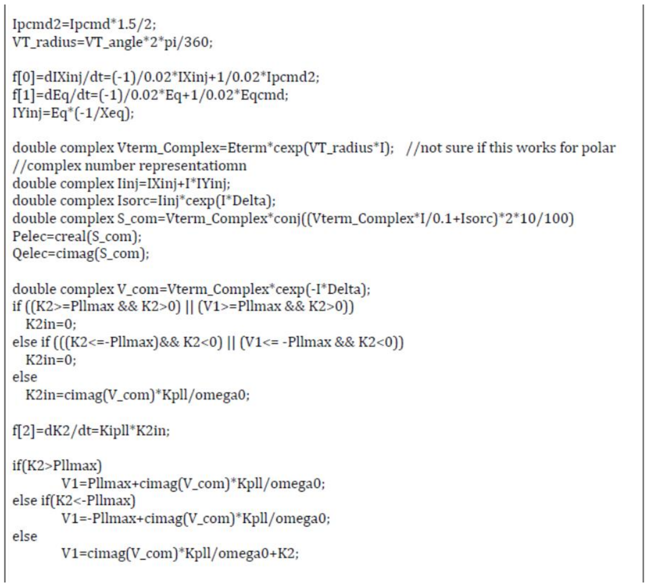
$$\begin{aligned} \dot{x} &= u - \tfrac{1}{T_2}(T_1 u + x) \\ y &= \tfrac{1}{T_2}(T_1 u + x) \end{aligned}$$

Figure 3-6. Rules for conversion

Repeat the above steps, all four sub-models had the separate logical expression translated from the Simulink diagram.

B. Build pseudo-code to get a whole picture of each model

Following the sequence of operation in the model, a complete computational flow were written in pseudo code. This could not be done easily without the step A. By combining the logical expressions prepared in A, draft program was created for each model. Figure 3-7 shows the partial pseudo code for WT3G1.



```
Ipcmd2=Ipcmd*1.5/2;
VT_radius=VT_angle*2*pi/360;

f[0]=dIXinj/dt=(-1)/0.02*IXinj+1/0.02*Ipcmd2;
f[1]=dEq/dt=(-1)/0.02*Eq+1/0.02*Eqcmd;
IYinj=Eq*(-1/Xeq);

double complex Vterm_Complex=Eterm*cexp(VT_radius*I);   //not sure if this works for polar
//complex number representatiomn
double complex Iinj=IXinj+I*IYinj;
double complex Isorc=Iinj*cexp(I*Delta);
double complex S_com=Vterm_Complex*conj((Vterm_Complex*I/0.1+Isorc)*2*10/100)
Pelec=creal(S_com);
Qelec=cimag(S_com);

double complex V_com=Vterm_Complex*cexp(-I*Delta);
if ((K2>=Pllmax && K2>0) || (V1>=Pllmax && K2>0))
   K2in=0;
else if (((K2<=-Pllmax)&& K2<0) || (V1<= -Pllmax && K2<0))
   K2in=0;
else
   K2in=cimag(V_com)*Kpll/omega0;

f[2]=dK2/dt=Kipll*K2in;

if(K2>Pllmax)
        V1=Pllmax+cimag(V_com)*Kpll/omega0;
else if(K2<-Pllmax)
        V1=-Pllmax+cimag(V_com)*Kpll/omega0;
else
        V1=cimag(V_com)*Kpll/omega0+K2;
```

Figure 3-7. Partial pseudo code for WT3G1

Until this point, programmer's guide was created for all of the WT3 models, including WT3E1, WT3G1, WT3G2, WT3T1 and WT3P1. Actual C code implementation is the next.

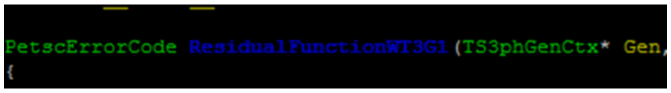
C. Build Residual Function for each model in C code

Building Residual Function is the main task of our project. It is the actual meat. Everything else is working for it.

Each model has a corresponding residual function file. They were firstly built in skeletons one by one. Then each file was added different sections used for different purposes. After debugging and compiling, they will be tested with the simulator and finally be connected to each other.

Take WT3G1 as an example to see how a residual function file is built up.

Figure 3-8 shows the title of the residual function file, including some input arguments.

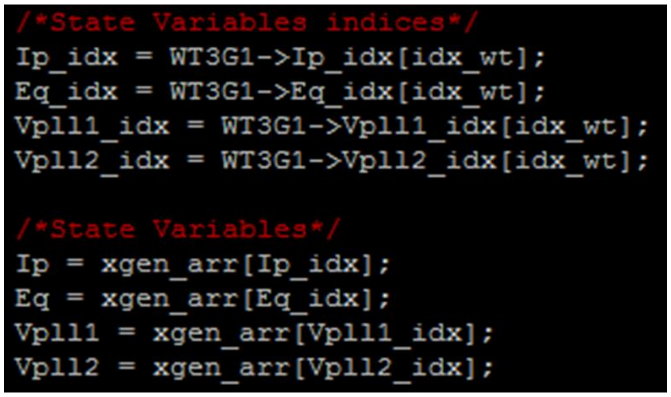


```
PetscErrorCode ResidualFunctionWT3G1(TS3phGenCtx* Gen,
{
```

Figure 3-8. Function title of WT3G1

After the declaration of different variables, the program retrieves the State Variables from the array after PETSc evaluated it in last iteration.

Figure 3-9 shows the partial code.

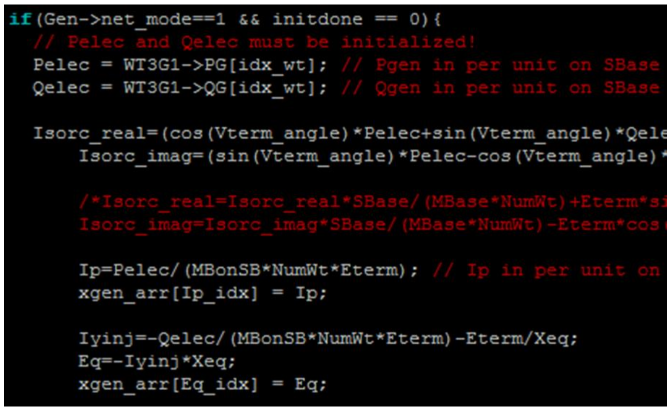


```
/*State Variables indices*/
Ip_idx = WT3G1->Ip_idx[idx_wt];
Eq_idx = WT3G1->Eq_idx[idx_wt];
Vpll1_idx = WT3G1->Vpll1_idx[idx_wt];
Vpll2_idx = WT3G1->Vpll2_idx[idx_wt];

/*State Variables*/
Ip = xgen_arr[Ip_idx];
Eq = xgen_arr[Eq_idx];
Vpll1 = xgen_arr[Vpll1_idx];
Vpll2 = xgen_arr[Vpll2_idx];
```

Figure 3-9. Retrieving state variables

Initialization for state variables is important in the first integration. It is done by calculating each state value backward.

Figure 3-10 shows the actual implementation of initialization. This should be done in the very beginning of the simulation.



```
if(Gen->net_mode==1 && initdone == 0){
  // Pelec and Qelec must be initialized!
  Pelec = WT3G1->PG[idx_wt]; // Pgen in per unit on SBase
  Qelec = WT3G1->QG[idx_wt]; // Qgen in per unit on SBase

  Isorc_real=(cos(Vterm_angle)*Pelec+sin(Vterm_angle)*Qele
      Isorc_imag=(sin(Vterm_angle)*Pelec-cos(Vterm_angle)*

      /*Isorc_real=Isorc_real*SBase/(MBase*NumWt)+Eterm*si
      Isorc_imag=Isorc_imag*SBase/(MBase*NumWt)-Eterm*cos(

      Ip=Pelec/(MBonSB*NumWt*Eterm); // Ip in per unit on
      xgen_arr[Ip_idx] = Ip;

      Iyinj=-Qelec/(MBonSB*NumWt*Eterm)-Eterm/Xeq;
      Eq=-Iyinj*Xeq;
      xgen_arr[Eq_idx] = Eq;
```

Figure 3-10. Initialization of Dynamic States

Then the program computes normally with basic logic operations until it reaches control blocks. As shown in Figure 3-11 and Figure 3-12, to evaluate control blocks, fgen_arr stores the state variebles and give it to PETSc solver, which will solve it for us and store the result in x_gen array.

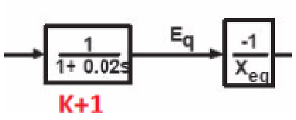
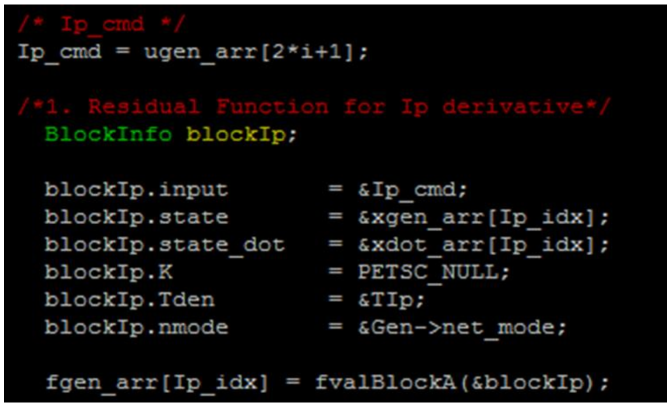


Figure 3-11. State K+1 in WT3G1 control block example

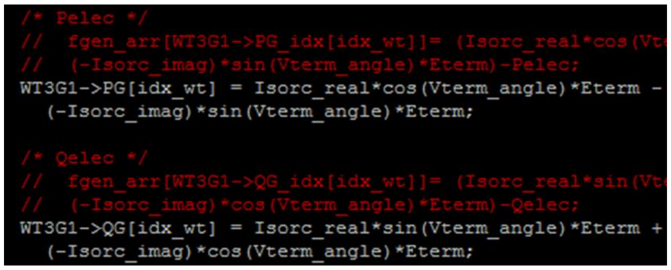


```
/* Ip_cmd */
Ip_cmd = ugen_arr[2*i+1];

/*1. Residual Function for Ip derivative*/
  BlockInfo blockIp;

  blockIp.input        = &Ip_cmd;
  blockIp.state        = &xgen_arr[Ip_idx];
  blockIp.state_dot    = &xdot_arr[Ip_idx];
  blockIp.K            = PETSC_NULL;
  blockIp.Tden         = &TIp;
  blockIp.nmode        = &Gen->net_mode;

  fgen_arr[Ip_idx] = fvalBlockA(&blockIp);
```

Figure 3-12. Evaluation of control block in C

Finally it reaches to the end of the model, in this case (Figure 3-13), the output is Pelec and Qelec. Put the updated results in the particular arrays and the program finishes with one iteration.

The process runs very quick and infinitely till reaching to the total time set by user.



```
/* Pelec */
//  fgen_arr[WT3G1->PG_idx[idx_wt]]= (Isorc_real*cos(Vt
//   (-Isorc_imag)*sin(Vterm_angle)*Eterm)-Pelec;
WT3G1->PG[idx_wt] = Isorc_real*cos(Vterm_angle)*Eterm -
  (-Isorc_imag)*sin(Vterm_angle)*Eterm;

/* Qelec */
//  fgen_arr[WT3G1->QG_idx[idx_wt]]= (Isorc_real*sin(Vt
//   (-Isorc_imag)*cos(Vterm_angle)*Eterm)-Qelec;
WT3G1->QG[idx_wt] = Isorc_real*sin(Vterm_angle)*Eterm +
  (-Isorc_imag)*cos(Vterm_angle)*Eterm;
```

Figure 3-13. Store new output value

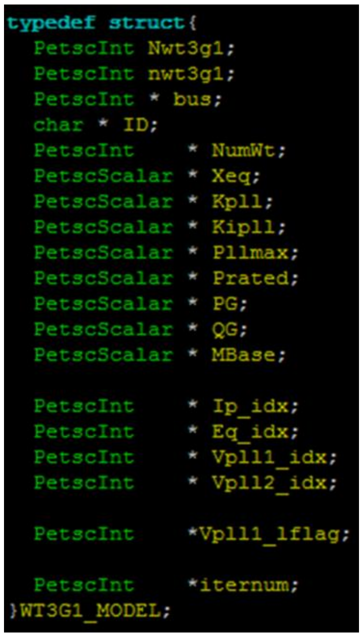
All sub-models' residual functions have been built but only WT3G1 has been compiled.

D. Connect to the TS3ph Simulator in ECE server

In the server, there are many files needed to be set up to actually let the main code run.

Some examples are header files for each models in WT3, Read DYR files, compiling and running sequence files, DYR data files, and so on.

C language is very powerful and simple, but it needs to manually build some advanced data structures to show its real power. Figure 3-14 shows how we built linked list of WT3G1 model in C. It can also be seen as an object with its attributes in there.



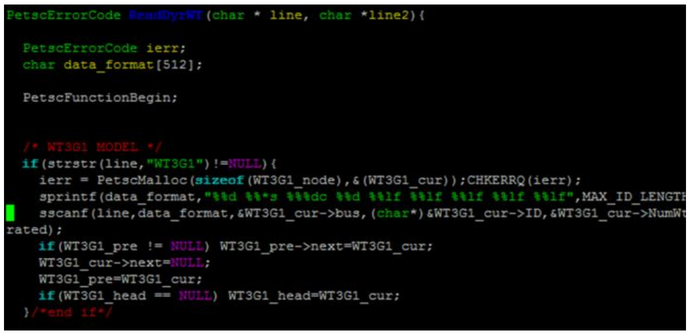
```
typedef struct{
  PetscInt Nwt3g1;
  PetscInt nwt3g1;
  PetscInt * bus;
  char * ID;
  PetscInt    * NumWt;
  PetscScalar * Xeq;
  PetscScalar * Kpll;
  PetscScalar * Kipll;
  PetscScalar * Pllmax;
  PetscScalar * Prated;
  PetscScalar * PG;
  PetscScalar * QG;
  PetscScalar * MBase;

  PetscInt    * Ip_idx;
  PetscInt    * Eq_idx;
  PetscInt    * Vpll1_idx;
  PetscInt    * Vpll2_idx;

  PetscInt    *Vpll1_lflag;

  PetscInt    *iternum;
}WT3G1_MODEL;
```

Figure 3-14. Build Abstract Object in C

DYR file is a file to configure the grid and models in PSS®E. It is set up by user so that the simulation will have huge flexibility.

Figure 3-16 shows the example DYR file in PSS®E. It needed to be prepared in the server by creating a similar format of file.

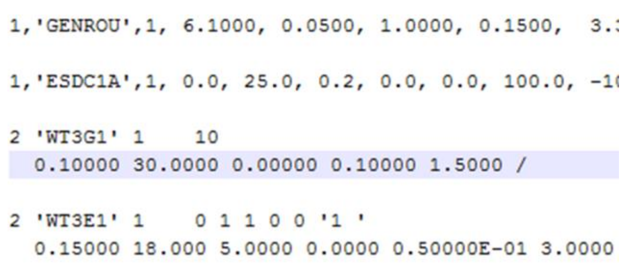
ReadDyrWT (Figure 3-15) is a function to read WT3 DYR data and store them into the particular data storage.



```
PetscErrorCode ReadDyrWT(char * line, char *line2){

  PetscErrorCode ierr;
  char data_format[512];

  PetscFunctionBegin;


  /* WT3G1 MODEL */
  if(strstr(line,"WT3G1")!=NULL){
    ierr = PetscMalloc(sizeof(WT3G1_node),&(WT3G1_cur));CHKERRQ(ierr);
    sprintf(data_format,"%%d %%*s %%%dc %%d %%lf %%lf %%lf %%lf %%lf",MAX_ID_LENGT
    sscanf(line,data_format,&WT3G1_cur->bus,(char*)&WT3G1_cur->ID,&WT3G1_cur->NumWt
rated);
    if(WT3G1_pre != NULL) WT3G1_pre->next=WT3G1_cur;
    WT3G1_cur->next=NULL;
    WT3G1_pre=WT3G1_cur;
    if(WT3G1_head == NULL) WT3G1_head=WT3G1_cur;
  }/*end if*/
```

Figure 3-15. Program of reading Dynamic Data file

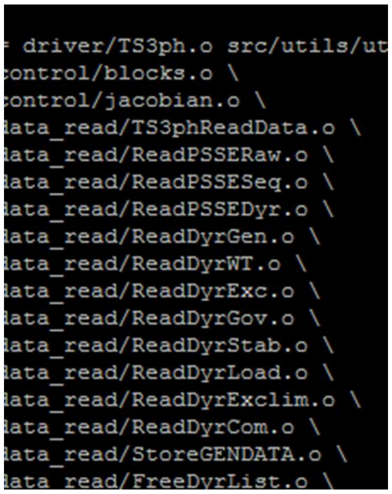
The code in the Read Dyr data file is able to read the input parameters by user.



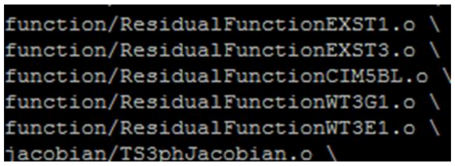
```
1,'GENROU',1, 6.1000, 0.0500, 1.0000, 0.1500,  3.:

1,'ESDC1A',1, 0.0, 25.0, 0.2, 0.0, 0.0, 100.0, -1(

2 'WT3G1' 1    10
  0.10000 30.0000 0.00000 0.10000 1.5000 /

2 'WT3E1' 1    0 1 1 0 0 '1 '
  0.15000 18.000 5.0000 0.0000 0.50000E-01 3.0000
```

Figure 3-16. Example Dyr file in PSS®E for WT3G1

Last but not least, modifying compile sequence file by adding ReadDyrWT.o and ResidualFunctionWT3**.o. Then when the simulator runs, it will compile and run the code we generated.



```
= driver/TS3ph.o src/utils/ut
ontrol/blocks.o \
ontrol/jacobian.o \
data_read/TS3phReadData.o \
data_read/ReadPSSERaw.o \
data_read/ReadPSSESeq.o \
data_read/ReadPSSEDyr.o \
data_read/ReadDyrGen.o \
data_read/ReadDyrWT.o \
data_read/ReadDyrExc.o \
data_read/ReadDyrGov.o \
data_read/ReadDyrStab.o \
data_read/ReadDyrLoad.o \
data_read/ReadDyrExclim.o \
data_read/ReadDyrCom.o \
data_read/StoreGENDATA.o \
data_read/FreeDyrList.o \
```



```
function/ResidualFunctionEXST1.o \
function/ResidualFunctionEXST3.o \
function/ResidualFunctionCIM5BL.o \
function/ResidualFunctionWT3G1.o \
function/ResidualFunctionWT3E1.o \
jacobian/TS3phJacobian.o \
```

Figure 3-17. Compile sequence File

IV. DISCUSSION

In this fall semester, our team had some new members and it turned out the outcome was successful.

First, residual function document for WT3G1, WT3E1, WT3T1, WT3P1 has been written for programming guideline and future research reference.

Then, residual function code for WT3G1, WT3E1, WT3T1, WT3P1 is almost ready for final testing.

In addition, read Dyr file, data structure and other necessary files to run the simulation of WT3G1 and WT3E1 has been prepared for running partial model testing.

The implementation of control block with non-windup limiter in C code was the most challenging task while integrating them in C code, but we successfully found a way to solve the problem. Example block diagram is shown in Figure 4-1.

Original Matlab code is shown as Figure 4-2.

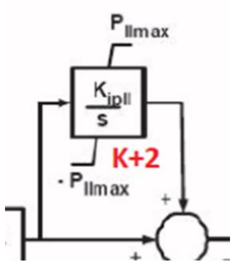
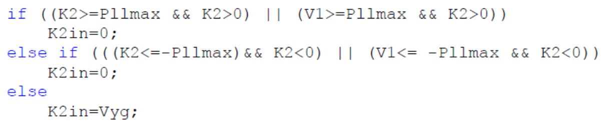
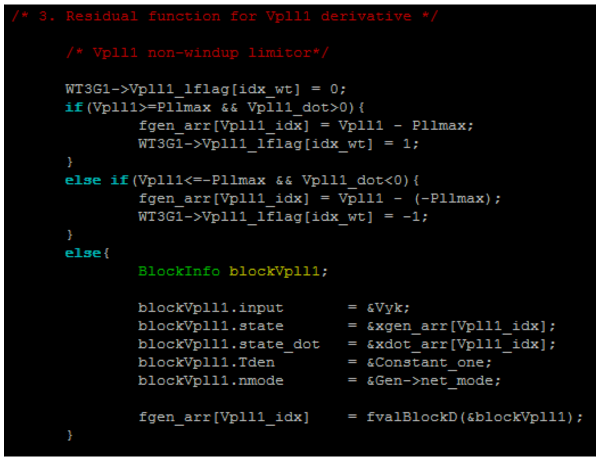


Figure 4-1. State K+2 in WT3G1 control block example



```
if ((K2>=Pllmax && K2>0) || (V1>=Pllmax && K2>0))
    K2in=0;
else if (((K2<=-Pllmax)&& K2<0) || (V1<= -Pllmax && K2<0))
    K2in=0;
else
    K2in=Vyg;
```

Figure 4-2. Matlab code for the block in Figure 4-1



```
/* 3. Residual function for Vpll1 derivative */

      /* Vpll1 non-windup limitor*/

      WT3G1->Vpll1_lflag[idx_wt] = 0;
      if(Vpll1>=Pllmax && Vpll1_dot>0){
              fgen_arr[Vpll1_idx] = Vpll1 - Pllmax;
              WT3G1->Vpll1_lflag[idx_wt] = 1;
      }
      else if(Vpll1<=-Pllmax && Vpll1_dot<0){
              fgen_arr[Vpll1_idx] = Vpll1 - (-Pllmax);
              WT3G1->Vpll1_lflag[idx_wt] = -1;
      }
      else{
              BlockInfo blockVpll1;

              blockVpll1.input        = &Vyk;
              blockVpll1.state        = &xgen_arr[Vpll1_idx];
              blockVpll1.state_dot    = &xdot_arr[Vpll1_idx];
              blockVpll1.Tden         = &Constant_one;
              blockVpll1.nmode        = &Gen->net_mode;

              fgen_arr[Vpll1_idx]     = fvalBlockD(&blockVpll1);
      }
```

Figure 4-3. Successfully build up non-wind up limiter in C

Figure 4-3 shows how to implement a basic control block with non-windup limiter in C.

However, accomplished such huge amount of work, our team encountered an unknown problem while trying to run the WT3G1 program in the simulator. It seemed like the simulator didn't recognize this type of wind turbine. Although we have consulted some graduate students and our professor, the problem is still remain unsolved. Comparison with PSS®E can be started immediately after we solve the problem.

Future researchers need to test the code by connecting models into the grid by a following order: WT3G1, WT3E1, WT3T1, WT3P1.

V. CONCLUSION

Several conclusions can be drawn from this work. Some main ones are listed below.

- Residual Function document for type 3 wind turbine model has been written for future use. Providing resources for future researchers and guideline for programming.
- Residual Function for all models of WT3 are written in C code.
- Debugging and testing are needed for future development to make sure the simulation result matches PSSE.
- The project will add an important feature to the TS3ph simulator and thus contributes to the understanding of how Type 3 wind turbines affect grid stability.

VI. ACKNOWLEDGEMENTS

The author would like to acknowledge ACE Undergraduate Research Program from Illinois Institute of Technology and National Department of Energy for funding and supporting this project. In addition, the author would like to thank Dr. Alexander Flueck for his consistent mentoring and advising. Last, the author would like to thank Zhuoyi Chen, Xinyi Yu, Cam Ketchmark, Sida Liu for the awesome teamwork and innovative ideas.

VII. REFERENCES

[1] I. A. Hiskens, *Dynamics of Type-3 Wind Turbine Generator Models*, 2012.

[2] EPRI, *Converter Model Grid Interface.*

[3] EPRI, *WECC Type 3 Wind Turbine Generator Model – Phase II*, Knoxville, Tennessee, 2014.

[4] EPRI, *Proposed Changes to the WECC WT4 Generic Model for Type 4 Wind Turbine Generators*, Knoxville, Tennessee, 2011.

[5] *Typical Machine Data for Power System Simulation Modeling*, 2014.

[6] National Renewable Energy Laboratory, *WECC WIND GENERATOR DEVELOPMENT*, 2010.

[7] Q. Xu and Z. Xu, *A Survey on General Models of Wind Turbine Generators in PSS/E*, Hangzhou, Zhejiang, 2010.

[8] Siemens Energy, Inc., *Memorandum*, 2011.