Power System Toolbox 4

–User Manual–
Documentation Version 1.0.0

Last Document Build: October 13, 2020

Thad Haines

Power System Toolbox Introduction

Power System Toolbox (PST) is MATLAB code used to: 1) solve power flow problems; 2) simulate power-system transients; and 3) to linearize a power system model. All code is open-access and free. PST is widely used by researchers studying problems related to power-system dynamics. Its open access format enables researchers to customize models and operations to meet their unique needs. Being MATLAB based code, one can incorporate powerful MATLAB functions into their problem solving process.

PST was the brain child of Dr. Joe Chow of Rensselaer Polytechnic Institute with the original version written by him and Dr. Kwok W. Cheung in the early 1990s. The late Mr. Graham Rogers made significant contributions and included it in his book [7]. Many others have added customized models and variations over the past two plus decades.

The basis for the version of PST presented in this document is version 3, available from Dr. Chow's webpage (https://www.ecse.rpi.edu/ chowj/) with added customizable current-injection functions written by Dr. Dan Trudnowski from Montana Technological University.

User Manual Introduction

The purpose of this user manual is to document work done on PST 3 to form PST 4. It is meant only to augment the previous user manuals and other available documentation about PST [1]–[4]. Major code changes presented include how global variables are handled and the functionalization of the non-linear simulation routine.

Descriptions of new models created for inverter based resources and automatic generation control are also presented. Additionally, work to add functionality to PST such as variable time step integration routines, generator tripping, and code compatibility with Octave is documented.

To demonstrate and debug new and old PST capabilities, an example library has been created and brief results of select examples is included in this document. Unfortunately, due to time constraints, details are likely lacking. Sections in this document that are unfinished are denoted by a WIP in the header. Despite lack of documentation, the code examples have been checked for functionality and *should* work as designed.

Finally, it is worth noting that all source code, examples, and other research documentation can be accessed at https://github.com/thadhaines/MT-Tech-SETO/tree/master/PST.

Table of Contents

List of Tables

List of Figures

List of Equations

List of Listings

Glossary of Terms

Term	Definition
AC	Alternating Current
ACE	Area Control Error
AGC	Automatic Generation Control
DACE	Distributed ACE
DC	Direct Current
FTS	Fixed Time Step
Hz	Hertz, cycles per second
J	Joule, Neton meters, Watt seconds
ODE	Ordinary Differential Equation
P	Real Power
PI	Proportional and Integral
PSS	Power System Stabilizer
PST	Power System Toolbox
PU	Per-Unit
Q	Reactive power
RACE	Reported ACE
SACE	Smoothed ACE
VAR	Volt Amps Reactive
VTS	Variable Time Step
W	Watt, Joules per second
WECC	Western Electricity Coordinating Council

1 PST Version History

- 1.0 - Original PST circa 1991.
- 2.0 - Updated PST circa 1999.
- 2.1 - Dr. Dan Trudnowski added power and current injection via the pwrmod model circa 2015.
- 2.2 - Dr. Dan Trudnowski added arbitrary generator tripping functionality circa 2019.
- 2.3 - Dr. Dan Trudnowski added a voltage behind a reactance model as ivmmod circa 2019.
- 3.0 - Updated PST obtained from Joe Chow's website circa 2020. Includes fixes, alterations, and MIT license. Addition of multiple DC lines and PSS modifications.
- 3.1 - Based on 3.0, incorporates Dr. Trudnowski's pwrmod and ivmmod models in non-linear simulation, machine tripping functionality, and various model patches. pwrmod included in linear simulation as well.
- SETO - Based on 3.1. Experimental version by Thad Haines using a new global structure, automatic generation control, and variable time step (VTS) integration.
- 3.1.1 - Based on 3.1. Version by Ryan Elliott at Sandia National Labs with energy storage and updated linear simulation along with various other fixes and code cleanup alterations.
- 3.1.2 - Based on 3.1.1. Work in progress version by Ryan Elliott to clean up global variables.
- 4.0.0 - Released circa October 2020. Based on SETO version. Includes a refined VTS routine, confirmed multi-generator tripping, improved AGC action and modulation, code cleanup, example library, and documentation. Represents the end result of four months of work by Thad Haines.

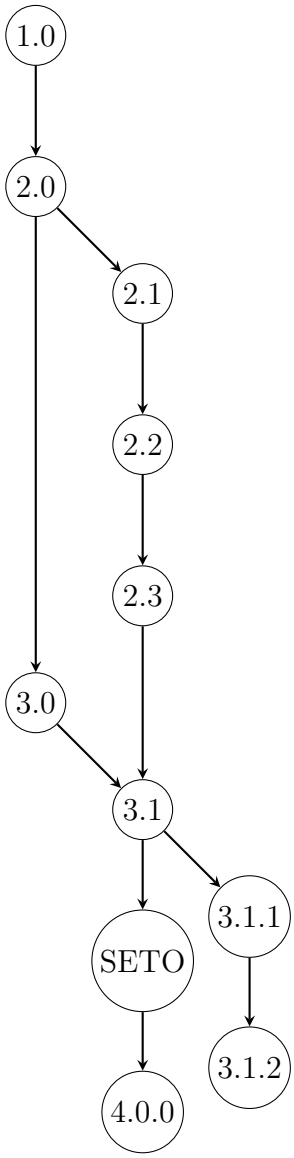


Figure 1.1: Development of PST.

2 Code Fixes in PST 4

Notable code fixes to code from PST 3 are collected in this chapter.

2.1 exc_dc12

In 2015 there were 'errors' corrected in the saturation block of the DC excitation model that create differences between PST version 2 and 3 of the `exc_dc12` model. Effects are noticeable, but a solution has not been investigated. The previous model version is included with PST 4 as `exc_dc12_old.m` but was not updated to use the new global structure.

2.2 exc_st3

There were two notable errors in the IEEE Type ST3 compound source rectifier exciter model `exc_st3` that were corrected:

- `theta` index variable changed to `n_bus` from `n` per Ryan Elliott.
- To use proper multiplication, the simple `*` was changed to `.*` in the `if ~isempty(nst3_sub)` section.

2.3 lmod

Load modulation had an error with state limiting. If over-limit, the derivative was set to zero, but the state was not set to the maximum/minimum value correctly. This issue has been resolved.

2.4 mac_tra

The transient machine model had commented code that prevented the setting equal of the transient reactances. These comments have been removed so that reactances are set equal.

2.5 rlmod

The same state limiting issues associated with `lmod` was found in `rlmod`. The issue was corrected in the same manner as `lmod`.

3 Changes and Additions in PST 4

This chapter contains information on changes and additions to PST since version 3.

3.1 Area Definitions

Lacking in previous versions of PST was a formal way to define areas. While many examples were commonly categorized as multi-area, they were handled the same as single area systems by PST.

PST 4 allows a user to create areas via an `area_def` array defined in a system data file. An example of a two area `area_def` array is shown in Listing 3.1.

Listing 3.1: Area Definition Example

```
%% area_def data format
% NOTE: should contain same number of rows as bus array (i.e. all bus areas defined)
% col 1 bus number
% col 2 area number
area_def = [ ...
             1   1;
             2   1;
             3   1;
             4   1;
             10  1;
             11  2;
             12  2;
             13  2;
             14  2;
             20  1;
            101  1;
            110  2;
            120  2];
```

Created areas are stored in the structured global (see Section 3.3.2) and track interchange, average frequency, and area inertia. An area is required for the AGC model to operate (see Section 3.2).

3.2 Automatic Generation Control

Automatic generation control (AGC) is an extended-term model that acts to restore system frequency and area interchange values to set values over the course of minutes. This restoration is accomplished by calculating an area control error (ACE) that is distributed to controlled generation sources to correct any deviation from reference values.

3.2.1 AGC Block Diagrams

The AGC process is shown in Figures 3.1, 3.2, and 3.3. $RACE$ (reporting ACE) and SACE (smoothed ACE) are calculated using PU values assuming B is a positive non-PU value with units of $MW/0.1Hz$. If K_{bv} is not zero, the resulting $RACE$ is not the industry standard (WECC defined) $RACE$ value. The scheduled interchange may be altered by the user via a `mAGC_sig` file that controls the behavior of the IC_{adj} input (see Section 4.4.2).

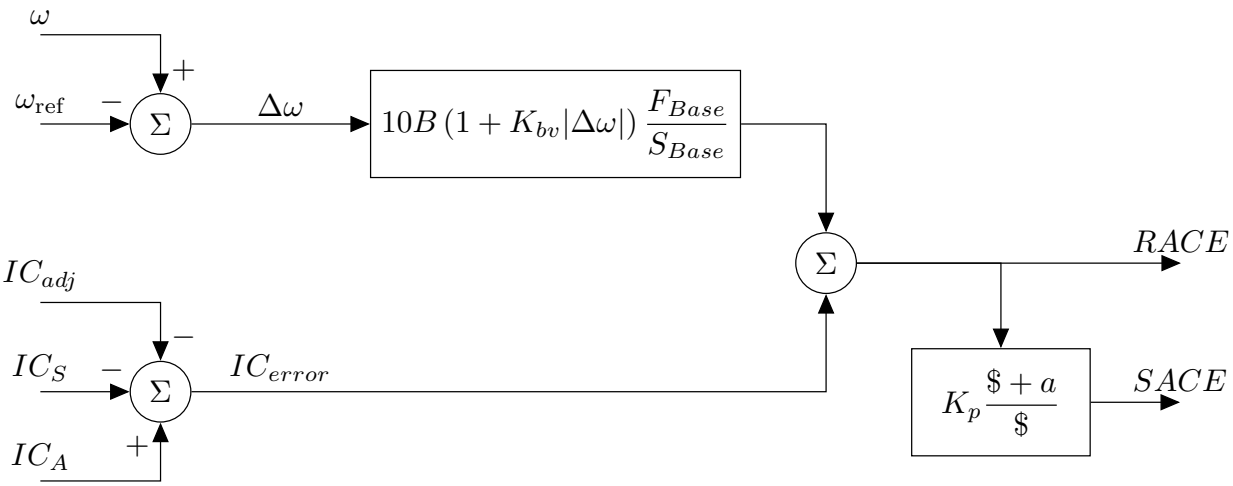


Figure 3.1: AGC calculation of $RACE$ and $SACE$.

RACE and *SACE* are calculated every simulation time step, however distribution of *SACE* is determined by the user defined `startTime` and `actionTime` variables. Assuming action, the conditional $\Delta\omega$ logic is processed before adjusting the `aceSig` value which is then gained to become `ace2dist`.

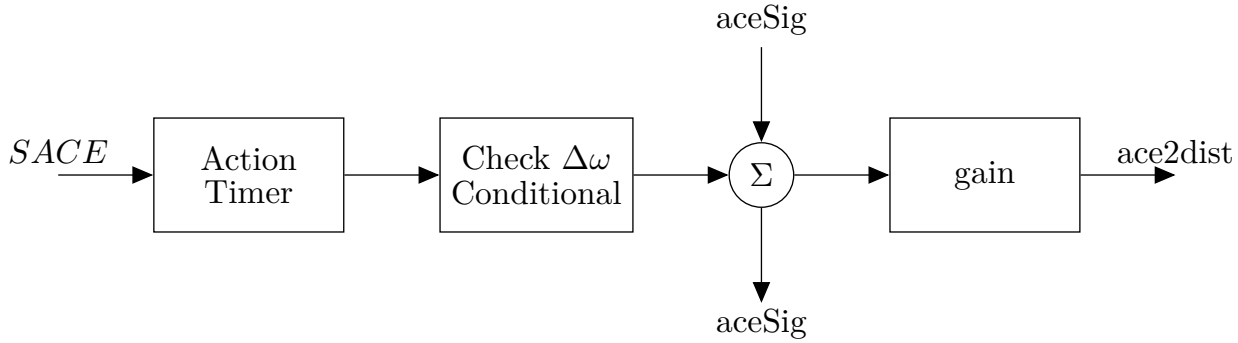


Figure 3.2: AGC calculation of *ace2dist*.

The `ace2dist` value is distributed to all controlled generators associated with the AGC model according to their respective participation factor `pF`. Each `ctrlGen` has a unique low pass filter to allow for different 'ramping' of signals to individual machines. The output of the low pass filter is gained by -1 and added to the existing associated governor `tg_sig` value to drive ACE to zero.

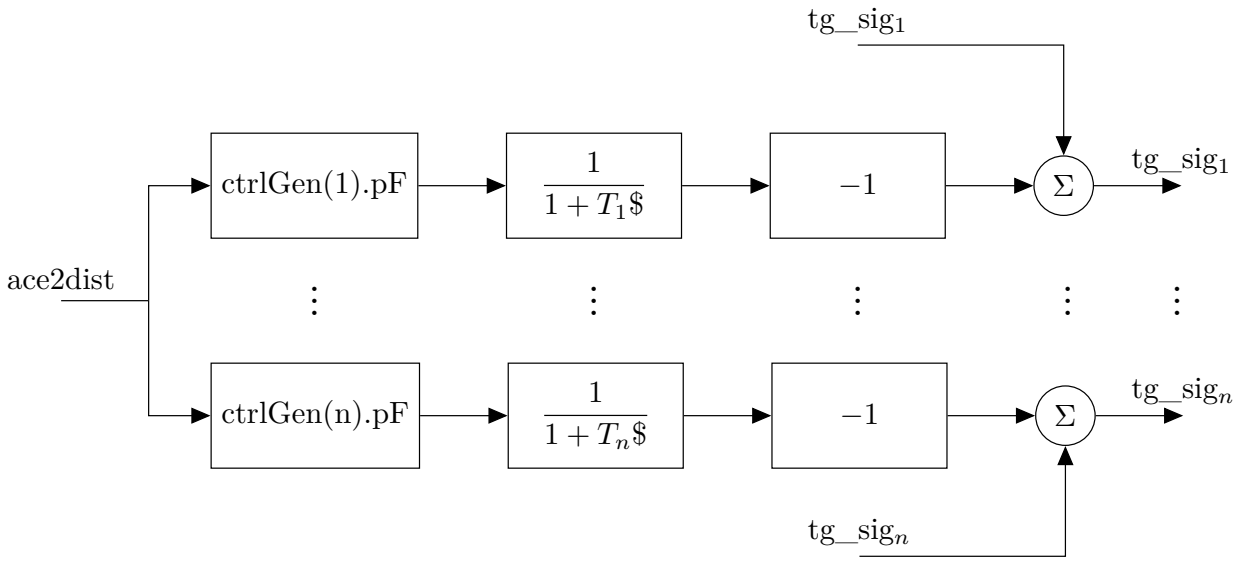


Figure 3.3: AGC handling of *ace2dist* to individual governor signals.

3.2.2 AGC Definition Example

The AGC settings in Listing 3.2 are not realistic, but useful in demonstrating the required user definitions and functionality of the AGC model. Settings from one AGC model can be easily copied to another using `agc(2) = agc(1)` and then changing the area number and `ctrlGen_con` array as desired.

Listing 3.2: AGC Definition Example

```
%{
AGC definition. Each agc(x) has following fields:
    area        - Area Number
    startTime   - Time of first AGC signal to send
    actionTime  - Interval of time between all following AGC signals
    gain        - Gain of output AGC signal
    Btype       - Fixed frequency bias type
        0 - absolute - Input B value is set as Frequency bias (positive MW/0.1Hz)
        1 - percent of max area capacity
    B           - Fixed frequency bias value
    Kbv         - Varaible frequency bias gain used to gain B as B(1+kBv*abs(delta_w))
    condAce     - Conditional ACE flag
        0 - Conditional ACE not considered
        1 - ace2dist updated only if sign matches delta_w (i.e. in area event)
    Kp          - Proportional gain
    a           - ratio between integral and proportional gain (placement of zero)
    ctrlGen_con - Controlled generator information
        column 1 - Generator External Bus Number
        column 2 - Participation Factor
        column 3 - Low pass filter time constant [seconds]
%}
agc(1).area = 1;
agc(1).startTime = 25;  % seconds
agc(1).actionTime = 15; % seconds
agc(1).gain = 2;        % gain of output signal
agc(1).Btype = 1;       % per max area capacity
agc(1).B = 1;           % Use 1% of max area capacity as B
agc(1).Kbv = 0;         % no variable bias
agc(1).condAce = 0;     % conditional ACE ignored
agc(1).Kp = 0.04;       % PI proportional gain
agc(1).a = 0.001;       % Ratio between integral and proportional gain
agc(1).ctrlGen_con = [ 1, 0.75, 15;
                       2, 0.25, 2;  ];
```

3.2.3 Weighted Average Frequency Calculation (calcAveF)

An inertia weighted average frequency is used as the 'actual' frequency ω in ACE calculations. The `calcAveF` function calculates an average weighted frequency for the total system and for each area (if areas are defined). System values are stored in `g.sys.aveF` and area values are stored in `g.area.area(x).aveF` where `x` is the internal area number. The calculation involves a sum of system inertias that changes with generator trips.

In a system with N generators, M areas, and N_M generators in area M, the `calcAveF` function performs the following calculations for each area M:

$$H_{tot_M} = \sum_{i}^{N_M} MVA_{base_i} H_i \tag{3.1}$$

$$F_{ave_M} = \left(\sum_{i}^{N_M} Mach_{speed_i} MVA_{base_i} H_i \right) \frac{1}{H_{tot_M}} \tag{3.2}$$

System total values are calculated as:

$$H_{tot} = \sum_{i}^{M} H_{tot_M} \tag{3.3}$$

$$F_{ave} = \left(\sum_{i}^{M} F_{ave_M} \right) \frac{1}{M} \tag{3.4}$$

If $M == 0$ (areas are not defined), `calcAveF` performs:

$$H_{tot} = \sum_{i}^{N} MVA_{base_i} H_i \tag{3.5}$$

$$F_{ave} = \left(\sum_{i}^{N} Mach_{speed_i} MVA_{base_i} H_i \right) \frac{1}{H_{tot}} \tag{3.6}$$

3.2.4 Other Area Calculations (calcAreaVals)

The `calcAreaVals` function calculates real and reative power generated by all area machines and actual area interchange every time step. It should be noted that power injected via loads, pwrmod, or ivmmod are not included in the `g.area.area(n).totGen(k)` value. An area's actual interchange is calculated using the `line_pq2` function to collect area to area line power flows.

3.3 Global Variable Management

Previous versions of PST rely on the use of over 340 global variables. It was decided to create a global structure that contains all exiting globals to enable easier development and use of PST. After global restructuring, initial simulation results showed a speed up of over 2 times (See Tables 4.1, 4.2, and 4.3).

Inside the global variable `g` are fields that corresponds to models, or groups, of other globals. Essentially, globals defined in the pre-existing `pst_var` script were collected into related fields. For example, the `g.mac.mac_spd` global contains all machine speeds while the `g.bus.bus_v` contains all bus voltages, etc. The following subsections describe the globals contained in each field of the global `g`. Consult [4] for a more detailed description of what pre-existing global variables represent.

3.3.1 agc

As the AGC is a model new to PST 4, the global field is structured slightly differently from other global fields. The `g.agc` field contains the number of AGC models as `g.agc.n_agc` and all other AGC model information is stored in the `a.agc.agc` structure. For example, `g.agc.agc(n).race(k)` would return the RACE value at index `k` of the `n`th AGC model. A description of the variables contained in every AGC structure are shown in Listing 3.3.

Listing 3.3: AGC Global Field Variables

```
a               % Ratio between integral and proportional gain
ace2dist        % Running value of ACE dispatch signal
aceSig          % Ungained ACE dispatch signal
actionTime      % Time (in seconds) between AGC dispatches
area            % Area number to control
B               % User input B value
Bcalc           % B value used for calculations
Btype           % Fixed frequency bias type
condAce         % Flag for conditional ACE
ctrlGen         % Stucture for controlled generator handling
ctrlGen_con     % User defined controlled generator array
curGen          % Running value of total generation from controlled machines
d_sace          % Derivative of SACE
gain            % Gain of aceSig
Kbv             % Variable frequency bias gain
Kp              % Proportional Gain
macBusNdx       % Bus index of controlled machines
maxGen          % Maximum generation value (used for capacity calcs)
n_ctrlGen       % Number of controlled generators
nextActionTime  % Simulation time (in seconds) of next AGC dispatch
race            % Running RACE
sace            % Running SACE
startTime       % Time (in seconds) of first AGC dispatch
tgNdx           % Index of controlled machines governors
```

3.3.2 area

Similar to the `g.agc` field, area globals did not exist prior to PST 4 and are handled in a related fashion. For example, `g.area.area(n).icA(k)` will return the actual interchange value at data index `k` from the `n`th area. A description of notable variables in the the area field are shown in Listing 3.4.

Listing 3.4: Area Global Field Variables

```
areaBusNdx      % Area bus array index
areaBuses       % Area bus external numbers
aveF            % Running area average frequency [pu]
exportLineNdx   % Line index of lines From area to another area
genBus          % External generator bus numbers
genBusNdx       % Generator bus array index
icA             % Actual Interchange - complex PU
icAdj           % Interchange adjustment signal
icS             % Scheduled Interchange - complex PU
importLineNdx   % Line index of lines to area from another area
loadBus         % Load bux external number
loadBusNdx      % Load bus index in bus array
macBus          % Machine bus external numbers
macBusNdx       % Machine bus index in bus array
maxCapacity     % Area maximum capaicty
number          % Area number
totGen          % Running total area generation - complex PU
totH            % Running total area inertia
```

3.3.3 bus

The `g.bus` field contains the user supplied `bus` array and all altered bus arrays associated with fault conditions created in `y_switch`. The bus field also contains the running values for bus voltages and angles in the `g.bus.bus_v` and `g.bus.theta` arrays respectively.

3.3.4 dc

This field contains collected global variables for DC models, calculations, and operations. The global variables collected into the `g.dc` field are shown in Listing 3.5.

Listing 3.5: DC Global Field Variables

```
%% HVDC link variables
global dcsp_con  dcl_con  dcc_con
global r_idx  i_idx n_dcl  n_conv  ac_bus rec_ac_bus  inv_ac_bus
global inv_ac_line  rec_ac_line ac_line dcli_idx
global tap tapr tapi tmax tmin tstep tmaxr tmaxi tminr tmini tstepr tstepi
global Vdc  i_dc P_dc i_dcinj dc_pot alpha gamma
global VHT dc_sig  cur_ord dcr_dsig dci_dsig
global ric_idx  rpc_idx Vdc_ref dcc_pot
global no_cap_idx  cap_idx  no_ind_idx  l_no_cap  l_cap
global ndcr_ud ndci_ud dcrud_idx dciud_idx dcrd_sig dcid_sig
%% States
%line
global i_dcr i_dci  v_dcc
global di_dcr  di_dci  dv_dcc
global dc_dsig % added 07/13/20 -thad
%rectifier
global v_conr dv_conr
%inverter
global v_coni dv_coni
% added to global dc
global xdcr_dc dxdcr_dc xdci_dc dxdci_dc angdcr angdci t_dc
global dcr_dc dci_dc % damping control
global ldc_idx
global rec_par inv_par line_par
```

Some DC related functions reused global variable names for local values but avoided conflict by not importing the specific globals. During global conversion, this coding approach caused some issues with accidental casting to global and overwriting issues. While the non-linear and linear simulations run, there may be issues with this problem yet to be discovered.

For example, the `tap` variable is re-written numerous times during a simulation when calculating line flows. However this variable is only used after being re-written and does not need to be global.

3.3.5 exc

This field contains collected global variables for exciter models, calculations, and operations. The global variables collected into the `g.exc` field are shown in Listing 3.6.

Listing 3.6: Exciter Global Field Variables

```
%% Exciter variables
global exc_con exc_pot n_exc
global Efd V_R V_A V_As R_f V_FB V_TR V_B
global dEfd dV_R dV_As dR_f dV_TR
global exc_sig
global smp_idx n_smp dc_idx n_dc  dc2_idx n_dc2 st3_idx n_st3
global smppi_idx n_smppi smppi_TR smppi_TR_idx smppi_no_TR_idx
global smp_TA smp_TA_idx smp_noTA_idx smp_TB smp_TB_idx smp_noTB_idx
global smp_TR smp_TR_idx smp_no_TR_idx
global dc_TA dc_TA_idx dc_noTR_idx dc_TB dc_TB_idx dc_noTB_idx
global dc_TE  dc_TE_idx dc_noTE_idx
global dc_TF dc_TF_idx dc_TR dc_TR_idx
global st3_TA st3_TA_idx st3_noTA_idx st3_TB st3_TB_idx st3_noTB_idx
global st3_TR st3_TR_idx st3_noTR_idx
```

3.3.6 igen

This field contains collected global variables for induction generator models, calculations, and operations. The global variables collected into the `g.igen` field are shown in Listing 3.7.

Listing 3.7: Induction Generator Global Field Variables

```
%% induction genertaor variables
global tmig  pig qig vdig vqig  idig iqig igen_con igen_pot
global igen_int igbus n_ig
%states
global  vdpig vqpig slig
%dstates
global dvdpig dvqpig dslig
% added globals
global s_igen
```

3.3.7 ind

This field contains collected global variables for induction motor models, calculations, and operations. The global variables collected into the `g.ind` field are shown in Listing 3.8.

Listing 3.8: Induction Load Global Field Variables

```
%% induction motor variables
global  tload t_init p_mot q_mot vdmot vqmot idmot iqmot ind_con ind_pot
global  motbus ind_int mld_con n_mot t_mot
% states
global  vdp vqp slip
% dstates
global dvdp dvqp dslip
% added globals
global s_mot
global sat_idx dbc_idx db_idx % has to do with version 2 of mac_ind
% changed all pmot to p_mot (mac_ind1 only)
```

Two models of this are included as `mac_ind1` (a basic version from 2.3), and `mac_ind2` which is an updated induction motor model. Default behavior is to use the newer model (`mac_ind2`).

3.3.8 ivm

This field contains collected global variables for the internal voltage model signals, calculations, and operations that use the `ivmmod` model. The global variables collected into the `g.ivm` field are shown in Listing 3.9.

Listing 3.9: IVMMOD Global Field Variables

```
global divmmod_d_sigst
global divmmod_e_sigst
global ivmmod_d_sig
global ivmmod_d_sigst
global ivmmod_data
global ivmmod_e_sig
global ivmmod_e_sigst
global mac_ivm_idx
global n_ivm
```

3.3.9 k

To allow for functionalized running, various index values were placed into the global structure in the `g.k` field The global variables collected into the `g.k` field are shown in Listing 3.10.

Listing 3.10: Index Related Global Field Variables

```
global k_inc h ks h_sol k_incdc h_dc
```

3.3.10 line

The `g.line` field contains the user supplied `line` array and all altered line arrays associated with fault conditions created in `y_switch`.

3.3.11 lmod

This field contains collected global variables for real load modulation models, calculations, and operations. The global variables collected into the `g.lmod` field are shown in Listing 3.11.

Listing 3.11: Real Load Modulation Global Field Variables

```
global lmod_con                  % defined by user
global n_lmod lmod_idx           % initialized and created in lm_indx
global lmod_sig lmod_st dlmod_st % initialized in s_simu
global lmod_pot                  % created/initialized in lmod.m
global lmod_data                 % added by Trudnowski - doesn't appear to be used
```

3.3.12 lmon

Line monitoring during simulation is new to PST 4 and, like AGC or area fields, is structured differently from other global fields. For example:
`g.lmon.line(n).sFrom(k)` would return the complex power flow from the `n`th monitored line at index `k`. A description of the logged variables contained in every `g.lmon.line` structure are shown in Listing 3.12.

Listing 3.12: Line Monitoring Global Field Variables

```
iFrom  % Complex current injection at from bus
iTo    % Complex current injection at to bus
sFrom  % Complex power injection at from bus
sTo    % Complex power injection at to bus
```

3.3.13 mac

This field contains collected global variables for machine models, calculations, and operations. The global variables collected into the `g.mac` field are shown in Listing 3.13.

Listing 3.13: Machine Global Field Variables

```
global mac_con mac_pot mac_int ibus_con
global mac_ang mac_spd eqprime edprime psikd psikq
global curd curq curdg curqg fldcur
global psidpp psiqpp vex eterm ed eq
global pmech pelect qelect
global dmac_ang dmac_spd deqprime dedprime dpsikd dpsikq
global n_mac n_em n_tra n_sub n_ib
global mac_em_idx mac_tra_idx mac_sub_idx mac_ib_idx not_ib_idx
global mac_ib_em mac_ib_tra mac_ib_sub n_ib_em n_ib_tra n_ib_sub
global pm_sig n_pm
global psi_re psi_im cur_re cur_im
% added
global mac_trip_flags
global mac_trip_states
```

3.3.14 ncl

This field contains collected global variables for non-conforming load models, calculations, and operations. The global variables collected into the `g.ncl` field are shown in Listing 3.14.

Listing 3.14: Non-Conforming Load Global Field Variables

```
global  load_con load_pot nload
```

3.3.15 pss

This field contains collected global variables for power system stabilizer models, calculations, and operations. The `pss_noT4_idx` was renamed, but doesn't seem to be used. The global variables collected into the `g.pss` field are shown in Listing 3.15.

Listing 3.15: PSS Global Field Variables

```
global pss_con pss_pot pss_mb_idx pss_exc_idx
global pss1 pss2 pss3 dpss1 dpss2 dpss3 pss_out
global pss_idx n_pss pss_sp_idx pss_p_idx;
global pss_T  pss_T2 pss_T4 pss_T4_idx
global pss_noT4_idx % misspelled in pss_indx as pss_noT4
```

3.3.16 pwr

This field contains collected global variables for power or current injection models, calculations, and operations that use the `pwrmod` model. The global variables collected into the `g.pwr` field are shown in Listing 3.16.

Listing 3.16: PWRMOD Global Field Variables

```
global pwrmod_con n_pwrmod pwrmod_idx
global pwrmod_p_st dpwrmod_p_st
global pwrmod_q_st dpwrmod_q_st
global pwrmod_p_sig pwrmod_q_sig
global pwrmod_data
```

3.3.17 rlmod

This field contains collected global variables for reactive load modulation models, calculations, and operations. The global variables collected into the `g.rlmod` field are shown in Listing 3.17.

Listing 3.17: Reactive Load Modulation Global Field Variables

```
global rlmod_con n_rlmod rlmod_idx
global rlmod_pot rlmod_st drlmod_st
global rlmod_sig
```

3.3.18 svc

This field contains collected global variables for static VAR control system models, calculations, and operations. The global variables collected into the `g.svc` field are shown in Listing 3.18.

Listing 3.18: SVC Global Field Variables

```
global svc_con n_svc svc_idx svc_pot svcll_idx
global svc_sig
% svc user defined damping controls
global n_dcud dcud_idx svc_dsig
global svc_dc % user damping controls?
global dxsvc_dc xsvc_dc
%states
global B_cv B_con
%dstates
global dB_cv dB_con
```

There appears to be code related to user defined damping control of SVC, but it does not seem to be described in any available documentation. This damping functionality was added by Graham Rogers circa 1998/1999.

3.3.19 sys

This field contains variables that deal with simulation operations. The global variables collected into the `g.sys` field are shown in Listing 3.19.

Listing 3.19: System Global Field Variables

```
global basmva basrad syn_ref mach_ref sys_freq
% globals added
global sw_con livePlotFlag Fbase t t_OLD
global aveF totH
global ElapsedNonLinearTime clearedVars
```

3.3.20 tcsc

This field contains collected global variables for thyristor controlled series reactor models, calculations, and operations. The global variables collected into the `g.tcsc` field are shown in Listing 3.20.

Listing 3.20: TCSC Global Field Variables

```
global tcsc_con n_tcsc tcsvf_idx tcsct_idx
global B_tcsc dB_tcsc
global tcsc_sig tcsc_dsig
global n_tcscud dtcscud_idx  %user defined damping controls
% previous non-globals added as they seem to relavant
global xtcsc_dc dxtcsc_dc td_sig tcscf_idx
global tcsc_dc
```

Similar to the SVC model, there seems to be some added functionality for controlled damping, but no examples or previous documentation could be found. This damping functionality was added by Graham Rogers circa 1998/1999.

3.3.21 tg

This field contains collected global variables for turbine governor models, calculations, and operations. The global variables collected into the `g.tg` field are shown in Listing 3.21.

Listing 3.21: Turbine Governor Global Field Variables

```
%% turbine-governor variables
global tg_con tg_pot
global tg1 tg2 tg3 tg4 tg5 dtg1 dtg2 dtg3 dtg4 dtg5
global tg_idx  n_tg tg_sig tgh_idx n_tgh
```

It should be noted that the hydro governor model `tgh` has **not** been modified as no examples could be found that use it.

3.3.22 vts

Globals associated with variable time step simulation runs were placed in the `g.vts` field. The collected variables are shown in Listing 3.22.

Listing 3.22: VTS Global Field Variables

```
dataN       % Used as a the data index for logging values
dxVec       % Vector used to collect current dataN derivatives
fsdn        % A cell of fields, states, derivatives, and number of states
fts         % Cell containing any fixed step time vectors
fts_dc      % Cell containing any fixed step time vectors for DC simulation
iter        % Counter to monitor number of solutions per step
n_states    % Total system state count
netSlnCell  % Similar to fsdn, but related to netowrk variables
netSlnVec   % Vector used to store initial network solution results
options     % MATLAB ODE solver options
slns        % A running history of solution iterations per step
solver_con  % User defined array defining what solution method to use
stVec       % Vector used to collect current dataN states
t_block     % A list of time blocks collected from sw_con
t_blockN    % Current time block index being executed
tot_iter    % Total number of solutions
```

3.3.23 y

The `g.y` field contains reduced Y matrices, voltage recovery matrices, and bus order variables created in `y_switch` associated with fault conditions. These variables are later selected in the `networkSolution` to simulate programmed conditions.

3.4 Input Handling Function (handleNewGlobals)

The `handleNewGlobals` function checks for user defined system arrays and puts them into the required global fields. This allows for PST input to remain the same between PST 4 and previous versions.

3.5 Internal Voltage Model (ivmmod)

The `ivmmod` model was created by Dr. Trudnowski to simulate a voltage-behind an impedance generator in PST. The new 'internal voltage model' is meant to mimic the actions of a grid-forming inverter based generator. Figure 3.4 shows how a generators internal voltage E_i and angle δ_i may be manipulated by a user created function `ivmmod_dyn.m`. Any number of model structures can be modeled is the user supplied controls model.

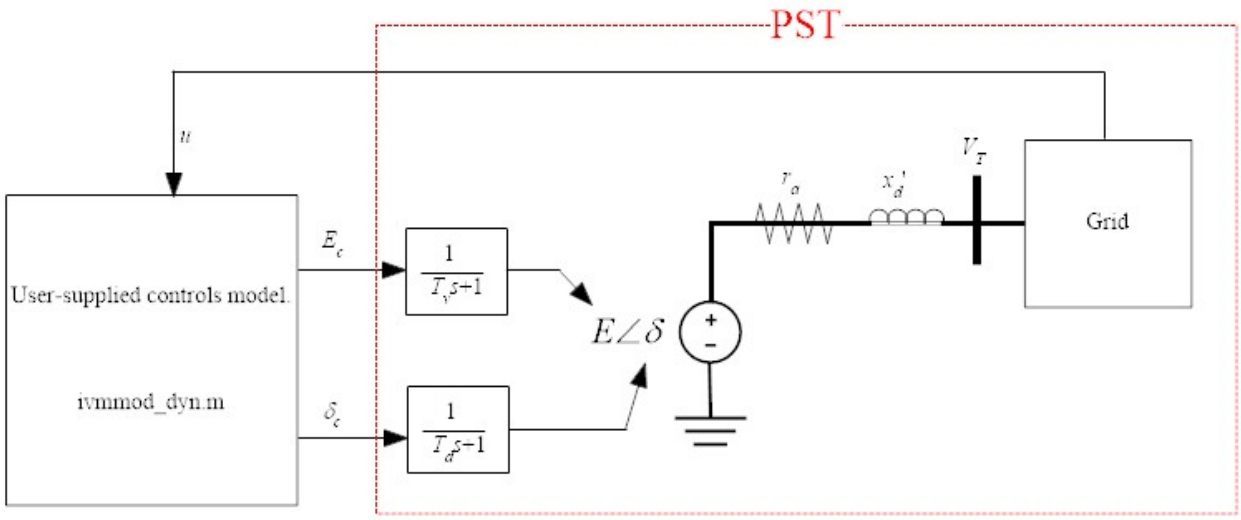


Figure 3.4: Block diagram of internal voltage model.

To define an `ivmmod` generator, the desired bus must be set to type 2 in the `bus` array, and a row in the `mac_con` array is used to define set parameters of the model. Listing 3.23 provides an example of a `mac_con ivmmod` definition. The model is recognized in PST by setting the inertia constant H (column 16) to zero.

Listing 3.23: IVMMOD Definition Example

```
%% IVM definition format:
% col 1       generator number
% col 2       bus number
% col 3       generator MVA base
% col 4       Not used
% col 5       r_a
% col 6       Not used
% col 7       x'_d
% col 8       Not used
% col 9       Td
% col 10      Tv
% col 11      Not used
% col 12      Not used
% col 13      Not used
% col 14      Not used
% col 15      Not used
% col 16      0 (this is H for a syn generator)
% col 17      Not used
% col 18      Not used
% col 19      bus number

% An example:
mac_con = […
% 1   2   3    4  5   6  7    8  9    10   11 12 13 14 15 16 17 18 19
% num bus base NA r_a NA x'_d NA Td   Tv   NA NA NA NA NA 0  NA NA bus
   3   2  100  0  0   0  0.1  0  0.05 0.05 0  0  0  0  0  0  0  0  2];
```

An example of the `ivmmod` is presented in Section 4.6. Currently, the `ivmmod` is only available in non-linear simulation.

3.6 Line Monitoring Functionality (lmon)

Previous versions of PST had line monitoring functionality, but the calculation was performed after the simulation was completed. PST 4 now calculates line current and power flow of monitored lines during non-linear simulation. The updated line monitoring routine allows for controls dealing with line flow monitoring (such as AGC). The definition of the `lmon_con` is the same as previous PST versions and an example is shown in Listing 3.24.

Listing 3.24: Line Monitoring Definition Example

```
%% Line Monitoring
% Each value corresponds to an array index in the line array.
% Complex current and power flow on the line will be calculated and logged during
↪  simulation

lmon_con = [3,10];
```

Calculated values are stored in the respective `g.lmon.line` structure. For example, if the above `lmon_con` was used, `g.lmon.line(2).iFrom(k)` would return the complex current injection at the from bus at data index `k` of the line defined in the line array at index 10.

3.7 Live Plotting Functionality (liveplot)

Default action of PST 3 was to plot the bus voltage magnitude at a faulted bus. However, it may be useful to plot other values or turn off the plotting during a simulation. The live plotting routine is now functionalized to allow users to more easily define what is displayed (if anything) during simulations. The `livePlot` function was designed to be overwritten during batch simulation runs as shown in Listing 3.25.

Listing 3.25: Live Plotting Overwrite Example

```
% PSTpath is the location of the root PST directory
copyfile([PSTpath 'liveplot_2.m'],[PSTpath 'liveplot.m']);     % Plot AGC signals
copyfile([PSTpath 'liveplot_ORIG.m'],[PSTpath 'liveplot.m']);  % Restore functionality
```

There are currently 3 live plot functions:

- `liveplot_ORIG` - Original faulted bus voltage plot.
- `liveplot_1` - Faulted bus voltage and system machine speeds plus any lmod signals.
- `liveplot_2` - AGC signals.

It should be noted that the live plotting can cause extremely slow simulations and occasional crashes. To disable live plotting:

- In stand-alone mode:

  Change the 'live plot?' field from a 1 to a 0 in the popup dialog box.

- In batch mode:

  Create a variable `livePlotFlag` and set as 0 or `false`.

3.8 Machine Trip Logic (mac_trip_logic)

A `mac_trip_logic` file is created by a user and placed in the root directory of PST to control the tripping of machines. This added functionality allows for multiple generator trips during a single simulation (See Section 4.7.1). The procedure PST performs to accomplish such a task is as follows:

During simulation initialization, `g.mac.mac_trip_flags` is initialized as a column vector of zeros that correspond to the `mac_con` array, and `g.mac.mac_trip_states` variable is set to zero. To trip a generator, the `mac_trip_flag` corresponding to the desired generator to trip is set to 1 via the user generated `mac_trip_logic` code. `mac_trip_logic` is executed in the `initStep` function which alters `g.mac.mac_trip_flags` to account for any programmed trips. Specifically, a 0 in the `g.mac.mac_trip_flags` vector is changed to a 1 to signify a generator has tripped.

The `g.mac.mac_trip_flags` vector is summed in the `networkSolution` or `networkSoltuionVTS` function. If the resulting sum is larger than 0.5, the line number connected to the generator in `g.line.line_sim` is found and the reactance is set to infinity (1e7). The reduced Y matrices are then recalculated and used to solve the network solution via an `i_simu` call.

It should be noted that if a machine is tripped, the altered reduced Y matrices are generated every simulation step. This repeated action could be minimized via use of globals and logic checks.

3.9 Octave Compatibility (octaveComp)

To more fully support free and open-source software, PST 4 has been developed to be compatible with Octave. The `octaveComp` function handles a number of known required commands for Octave compatibility. Standard PST actions have been found to be compatible with Octave, however some plotting features or other code inside the supplied examples may not be. It should be noted that more experimental features, like VTS, have not been tested fully in Octave. It is known that not all ODE options that exist in MATLAB are included in Octave. Current Octave compatible solution methods are:

- huens
- ode23
- ode15s

When using Octave, calls to save and load data must include explicit file endings (i.e. `save someData.mat`). Additionally, simulations executed in MATLAB generally run faster than those run in Octave. However, MATLAB costs hundreds of dollars while Octave is free.

3.10 Power Injection Model (pwrmod)

The `pwrmod` model was created by Dr. Trudnowski to inject real and reactive power into a bus. This model works with transient simulation via `s_simu` and linear analysis via `svm_mgen_Batch`. `pwrmod` can be used to model many devices such as wind turbines and solar plants, as well as any controls associated with such a device.

Figure 3.5 is a block diagram describing how power (or current) is injected into a given bus i. The user constructs all devices and modeling equations in the `pwrmod_dyn` function which handles the necessary equations to model the device and associated controls that perform the desired power injection. The model must be constructed in state-space form.

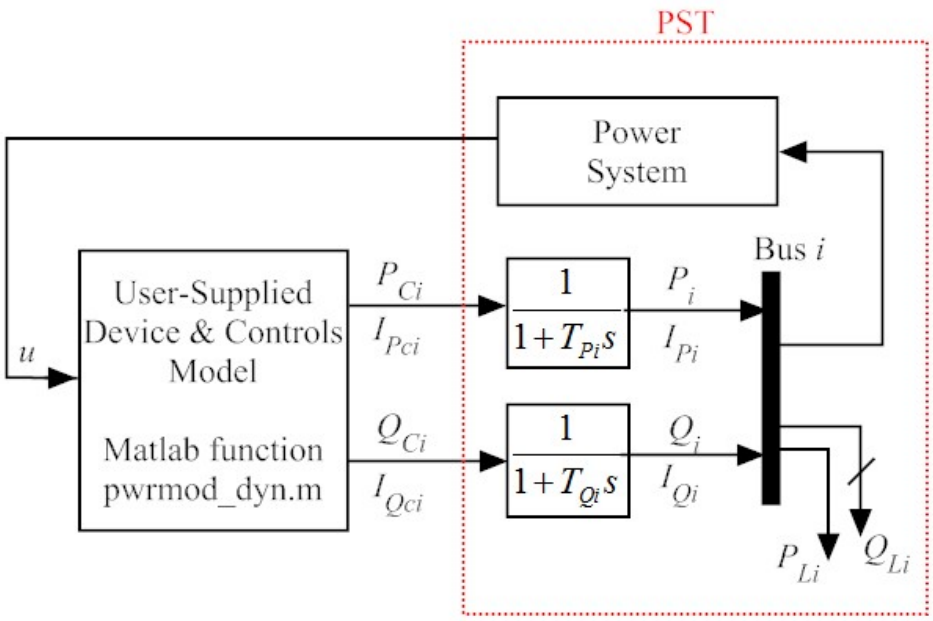


Figure 3.5: Block diagram of power modulation approach.

The insertion of a `pwrmod` model requires the target bus to be type 2 (generator type). Additionally, the bus must be defined as constant power (for power injection) or constant current (for current injection) in the `load_con`. An example of defining bus 2 for a power injection model, and bus 3 for a current injection model using `pwrmod` is shown in Listing 3.26. Other examples of `pwrmod` are described in Section 4.8.

Listing 3.26: PWRMOD Definition Example

```
%% pwrmod_con format
% col 1       bus number
% col 2       real-power time constant (Tp)
% col 3       max real-power modulation (pu on system base)
% col 4       min real-power modulation (pu on system base)
% col 5       reac-power time constant (Tq)
% col 6       max reac-power modulation (pu on system base)
% col 7       min reac-power modulation (pu on system base)

pwrmod_con=[...
%bus  T    Pmax Pmin  Tq   Qmax  Qmin
  2   0.05 1    -1    0.05  1     -1;
  3   0.05 1    -1    0.05  1     -1];

% non-conforming load
% col 1       bus number
% col 2       fraction const active power load
% col 3       fraction const reactive power load
% col 4       fraction const active current load
% col 5       fraction const reactive current load

load_con = [...
%bus Pcont Qconst P_Icont Q_Iconst
  2   1     1     0       0;    % Modulation bus for power injection
  3   0     0     1       1;];  % Modulation bus for current injection
```

3.11 Power System Stabilizer Model (pss)

There was a modification to the washout time constant in the power system stabilizer (PSS) model between PST version 2 and 3 that affects the output of the model in a fairly drastic way. To accommodate for this, PST 4 has two PSS files named `pss2` and `pss3` which mimic the computation of each PST version PSS model respectively. This enables the user to specify which PSS model should be used by copying the numbered PSS files over the non-numbered PSS file. A code example of this overwritting process is shown in Listing 3.27 where `PSTpath` is a variable containing the full path to the root directory of PST 4.

Listing 3.27: PSS Overwrite Example

```
copyfile([PSTpath 'pss2.m'],[PSTpath 'pss.m']); % use version 2 model of PSS
copyfile([PSTpath 'pss3.m'],[PSTpath 'pss.m']); % use version 3 model of PSS
```

It should be noted that the default PSS model used in PST 4 is `pss2`.

3.12 Reset Original Files Function (resetORIG)

As the operation of PST commonly involves replacing system models and modulation files, the need to restore all original files can be very useful. Especially if unexpected behavior is observed in simulation output. The `resetORIG` script replaces all possible model and modulation files with the default versions. As of this writing, the correct operation of this script is as follows:

1. Navigate to root PST 4 directory in MATLAB
2. Execute `resetORIG`

A text message should be displayed stating the restoration process has completed. If other models are preferred for default use, the restoration process can be modified. Simply change the `**_ORIG` file name to the file that should be copied instead.

3.13 s_simu Changes and Functionalization

The `s_simu` script used to run non-linear simulations from PST 3 was relatively long and objectively messy. Work has been done to clean up the pre-existing code and functionalize sections for easier readability and process understanding. Additionally, 'stand alone mode' and 'batch mode' capabilities have been condensed into one file. The difference between batch and stand alone mode is that the later will prompt the user for input of a system file and other simulation parameters while batch mode assumes the file to run is the `DataFile.m` in the root PST directory.

- To use stand alone mode:

  Run `s_simu` after issuing the `clear all; close all` commands.

- To use batch mode:

  Ensure at least 1 non-global variable is in the workspace prior to running `s_simu`.

The use of a single global allowed for easier functionalization of code. The following subsections describe some of the functions that were created from code originally found in `s_simu`.

3.13.1 cleanZeros

The `cleanZeros` function cleans all entirely zero variables from the global `g` and places the names of cleared variables into the `clearedVars` cell that is stored in `g.sys`. The function is executed near the end of `s_simu`.

3.13.2 correctorIntegration

As shown in Listing 3.28, the `correctorIntegration` function performs the corrector integration step of the simulation loop to calculate the next accepted value of integrated states. The executed code was taken directly from `s_simu` and modified to work with the new global `g`.

Listing 3.28: Function Header for correctorIntegration

```
function correctorIntegration(k, j, h_sol)
% CORRECTORINTEGRATION Performs x(j) = x(k) + h_sol*(dx(j) + dx(k))/2
%
%   Input:
%   k - data index for 'n'
%   j - data index for 'n+1'
%   h_sol - time between k and j
```

It should be noted that the two integration functions write new states to the same `j` data index. Additionally, the `h_sol` value is updated in `i_simu` (called during the network solution) from the index of `ks` referencing an `h` array containing time step lengths... While this process seemed unnecessarily confusing and sort of round-about, it has not been changed as of this writing.

3.13.3 dcSolution

The portion of `s_simu` that integrates DC values at 10 times the rate of the normal time step was moved into the `dcSolution` function. This has not been tested with VTS, but was functionalized to enable future development. It appears to work as normal when using Huen's method (FTS), but a thorough testing has yet to be performed as of this writing.

Realistically, it doesn't make much sense to include a multi-rate integration routine into a variable step routine. If DC is to be integrated into VTS, the models should be handled as all other models. While this may defeat the purpose of VTS simulation due to DC models traditionally having very fast time constants, it is an avenue to explore should future development be desired.

3.13.4 dynamicSolution

As the name implies, the `dynamicSolution` function performs the dynamic model calculations at the passed in data index `k` by calling each required model with the input flag set to 2. This functionalized code was taken directly from `s_simu`.

3.13.5 huensMethod

The default integration method used by PST is Huen's Method. The routine has been collected into the `huensMethod` function from previously existing code in `s_simu`. Mathematically speaking, Huen's method is an improved Euler method that could also be described as a two-stage Runge-Kutta method, or as a predictor-corrector method.

The process of a generalized Huen's method is as follows: Suppose the initial conditions of an ODE $f(x, y)$ are given as x_i and y_i. To calculate y_{i+1} using Huen's method, the derivative at x_i, $\dot{x}_i$, is calculated from the initial conditions.

$$\dot{x}_i = f(x_i, y_i) \tag{3.7}$$

A predicted point is calculated using a Forward Euler method where h is the time step size.

$$y_p = y_i + h\dot{x}_i \tag{3.8}$$

The derivative at the predicted point is also calculated.

$$\dot{x}_p = f(x_{i+1}, y_p) \tag{3.9}$$

The next value for y, y_{i+1}, is calculated using an average of the two derivatives

$$y_{i+1} = y_i + \frac{h}{2}(\dot{x}_i + \dot{x}_p). \tag{3.10}$$

As a power system solution is not just a set of differential equations, but a system of algebraic and differential equations, `huensMethod` performs the network, dynamic, DC, and monitor solutions required for each step of the method. A flow chart of these actions is shown in Figure 3.6.

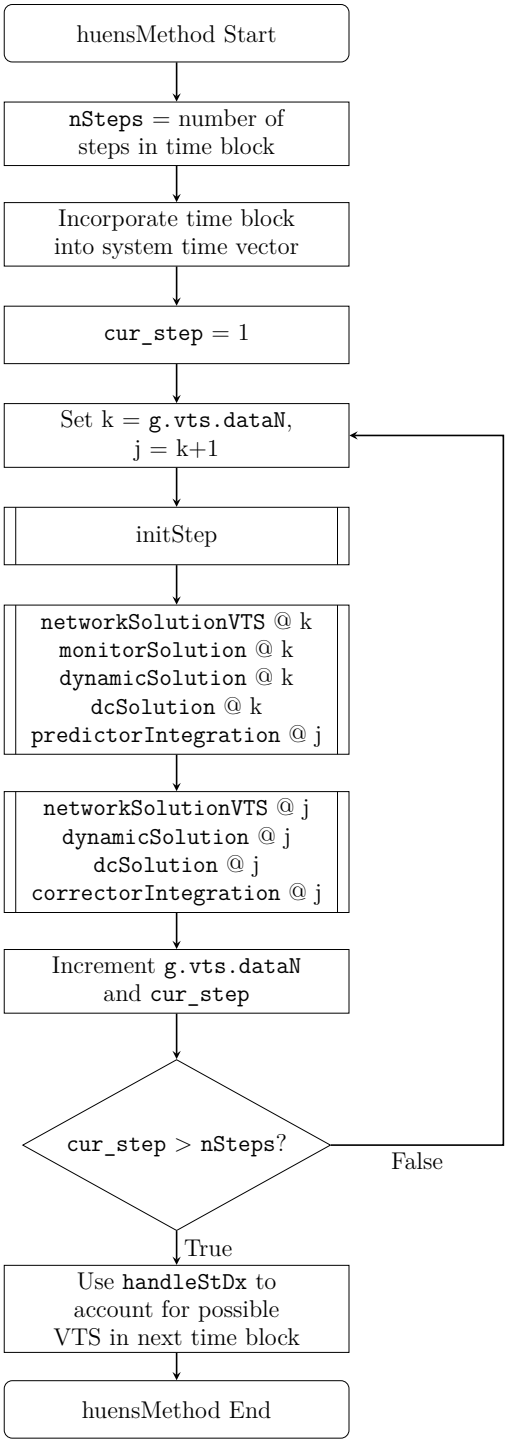


Figure 3.6: Huen's Method Flow Chart.

An added benefit of functionalizing Huen's method is that it provides a clear location to insert alternative integration routines and can act as an example as to how to accomplish such a task.

3.13.6 initNLsim

The `initNLsim` function is a collection of code from `s_simu` that performs initialization operations before a non-linear simulation. This includes the creation of the various Y-matrices used for fault conditions and the calling of the dynamic models with the input flag set to 0.

3.13.7 initStep

Code from `s_simu` that was executed at the beginning of each solution step was collected into `initStep`. Operations are related to setting values for the next step equal to current values for mechanical powers and DC currents and handling machine trip flags.

3.13.8 initTblocks

The `initiTblocks` function analyzes the global `sw_con` and `solver_con` to create appropriate *time blocks* that are used in non-linear simulation. Fixed time vectors associated with time blocks that use Huen's method are created while VTS time blocks are prepared for use. Care was taken to ensure a unique time vector (no duplicate time points). However, with the experimental option to switch between fixed step and variable step methods, this method may require slight modifications/refinements in the future.

3.13.9 initZeros

A large amount of code (≈400 lines) in PST 3's `s_simu` was dedicated to initializing zeros for data to be written to during non-linear simulation. This code has been collected into the `initZeros` function with inputs defining the desired length of vectors for normally logged data and DC data. The function header for `initZeros` is shown in Listing 3.29.

Listing 3.29: Function Header for initZeros

```
function initZeros(k, kdc)
% INITZEROS Creates zero arrays for logged values based on passed in input
%
%   Input:
%   k - total number of time steps in the simulation
%   kdc - total number of DC time steps in the simulation
```

3.13.10 monitorSolution

The `monitorSolution` function takes a single input that defines the data index used to calculate any user defined line monitoring values, average frequencies, and other interchange values for defined areas. It should be noted that these calculations are mostly based on complex voltages that are calculated during the network solution.

3.13.11 networkSolution & networkSolutionVTS

The `networkSolution` function is a collection of code from `s_simu` dealing with calls to dynamic models with the flag set to 1 and Y-matrix switching. The call to `i_simu` (which updates the current time step length `g.k.h_sol`) is also located in this function.

The `networkSolutionVTS` function is essentially the same as the `networkSolution` function, except instead of relying on index number to switch Y-matricies, the switching is done based on a passed in simulation time. This was a required change when using VTS as the previous method relied on a known number of steps between switching events, and that is no longer a reality with the experimental VTS methods. The default behavior of PST 4 is to use `newtowrkSolutionVTS` which requires the input of the current data index `k` and the simulation time.

3.13.12 predictorIntegration

The `predictorIntegration` function performs the predictor (forward Euler) integration step of the simulation loop. The code was taken directly from `s_simu` and uses the same variable names adapted for use with the global `g`. The associated function header is shown in Listing 3.30.

Listing 3.30: Function Header for predictorIntegration

```
function predictorIntegration(k, j, h_sol)
% PREDICTORINTEGRATION Performs  x(j) = x(k) + h_sol*dx(k)
%
%   Input:
%   k - data index for 'n'
%   j - data index for 'n+1'
%   h_sol - time between k and j
```

3.13.13 standAlonePlot

The `standAlonePlot` function is the updated plotting routine based on user input previously found at the end `s_simu`. After a completed simulation, it is called from `s_simu` if stand alone mode is detected. Alternatively, the function can be run independently from a non-linear simulation to analyze a pre-existing global `g` by being invoked as `standAlonePlot(1)`.

3.13.14 trimLogs

As there is no way to accurately predict the amount of (length of) data to be logged during a variable time step simulation, extra space is initially allocated and then all logged values are trimmed to the proper length post simulation. It should be noted that the current size allocation was arbitrary and can be altered as a user deems fit. Typically, an extended term simulation using VTS will requires fewer steps than a fixed step method, but that is not always the case. It's important to note that if not enough space is allocated, the simulation will crash when the code attempts to access data indices outside of the allocated range.

The `trimLogs` function trims all logged values in the global `g` to a given length `k`. It is executed near the end of `s_simu` before `cleanZeros`. The associated function header is shown in Listing 3.31.

Listing 3.31: Function Header for trimLogs

```
function trimLogs(k)
% TRIMLOGS trims logged data to input index k.
%
%   NOTES: nCell not made via logicals - may lead to errors if fields not initialized
↪  (i.e. model not used). Issue not encountered yet, but seems possible
%
%   Input:
%   k - data index
```

3.14 svm_mgen_Batch

Linear analysis was performed by `svm_mgen` in previous versions of PST. While the `svm_mgen` script still exists in PST 4, it is not updated to use the structured global `g`. Instead, the `svm_mgen_Batch` script is used to run linear analysis using a batch style process. The operations performed by the two scripts are similar, though `svm_mgen_Batch` does not prompt the user for any input and **is** modified to use the structured global g. Example cases that use `svm_mgen_Batch` are listed in Section 4.3.

3.15 Sub-Transient Machine Models

There are three versions of the sub-transient machine model (`mac_sub`) included with PST 4. The `mac_sub_ORIG` model is the standard PST model based on the R. P. Schulz, "Synchronous machine modeling" algorithm. The `mac_sub_NEW` model is based on the PSLF 'genrou' model by Dr. John Undrill. The `mac_sub_NEW2` model is the same as the `_NEW` model with minor bug fixes and alterations by Dr. Dan Trudnowski. Any model may be copied over the `mac_sub` file for simulation use as shown in Listing 3.32.

Listing 3.32: mac_sub Overwrite Example

```
copyfile([PSTpath 'mac_sub_NEW2.m'],[PSTpath 'mac_sub.m']); % use genrou model
copyfile([PSTpath 'mac_sub_ORIG.m'],[PSTpath 'mac_sub.m']); % restore model
```

3.16 sw_con Updates

Undocumented changes to the `sw_con` have occurred in version 3. The valid trip options of the `sw_con` row 2 column 6 are:

1 Three phase fault
2 Line-to-Ground
3 Line-to-Line
4 Loss of line with no fault
5 Loss of load at bus
6 No action
7 Clear fault without loss of line

3.17 Variable Time Step Integration

To enable more efficient simulation of extended term events, variable time step (VTS) integration routines were added to PST 4. The main assumption behind VTS simulation is that when a system is moving 'slowly', larger time steps can accurately capture system dynamics. Additionally, a viable VTS routine automatically adjusts the time step so that a known error tolerance is always met. As the time step increases, fewer solutions are required to simulate a set period of time which leads to reductions in the number of calculations and produces a more efficient solution method. Initial results show these assumptions to be valid (see Tables 4.2 and 4.4) VTS simulations were made possible by using standard MATLAB ODE solvers. This decision was made to more quickly explore the viability of applying VTS methods to the power system simulation process.

The MATLAB solvers adjust the simulation time step based upon analysis of system derivatives and error tolerances. To increase efficiency, models that are no longer connected to the system should have their derivatives set to 0. Work has been done to zero the derivatives of tripped machines and exciters, but more work can be done to zero derivatives of other attached models (i.e. PSS and governors).

NOTE: Variable time step simulation is experimental and should be used with caution!

3.17.1 Solver Control Array (solver_con)

To use VTS integration methods, a user will add a `solver_con` to a valid data file. If a `solver_con` is not specified, Huen's method is used for all time blocks (i.e. default PST 3 behavior).

Between each `sw_con` entry, a *time block* is created that is then solved using a the defined solution method in the `solver_con`. As such, the `solver_con` array has 1 less row than the `sw_con` array. An example `solver_con` array is shown in Listing 3.33.

Listing 3.33: Solver Control Array Example

```
%% solver_con format
% A cell with a solver method in each row corresponding to the specified
% 'time blocks' defined in sw_con
%
% Valid solver names:
% huens - Fixed time step default to PST
% ode113 - works well during transients, consistent # of slns, time step stays
↪  relatively small
% ode15s - large number of slns during init, time step increases to reasonable size
% ode23 - realtively consistent # of required slns, timstep doesn't get very large
% ode23s - many iterations per step - not efficient...
% ode23t - occasionally hundereds of iterations, most times not... decent performance
% ode23tb - similar to 23t, sometimes more large solution counts

solver_con ={ ...
    'huens';   % pre fault - fault
    'huens';   % fault - post fault 1
    'huens';   % post fault 1 - post fault 2
    'huens';   % post fault 2 - sw_con row 5
    'ode113';  % sw_con row 5 - sw_con row 6
    'ode23t';  % sw_con row 6  - sw_con row 7  (end)
    };
```

3.17.2 MATLAB ODE Solvers

The VTS implementation in PST revolves around using the built in MATLAB ODE solvers. All these methods perform actions depicted in Figure 3.7.

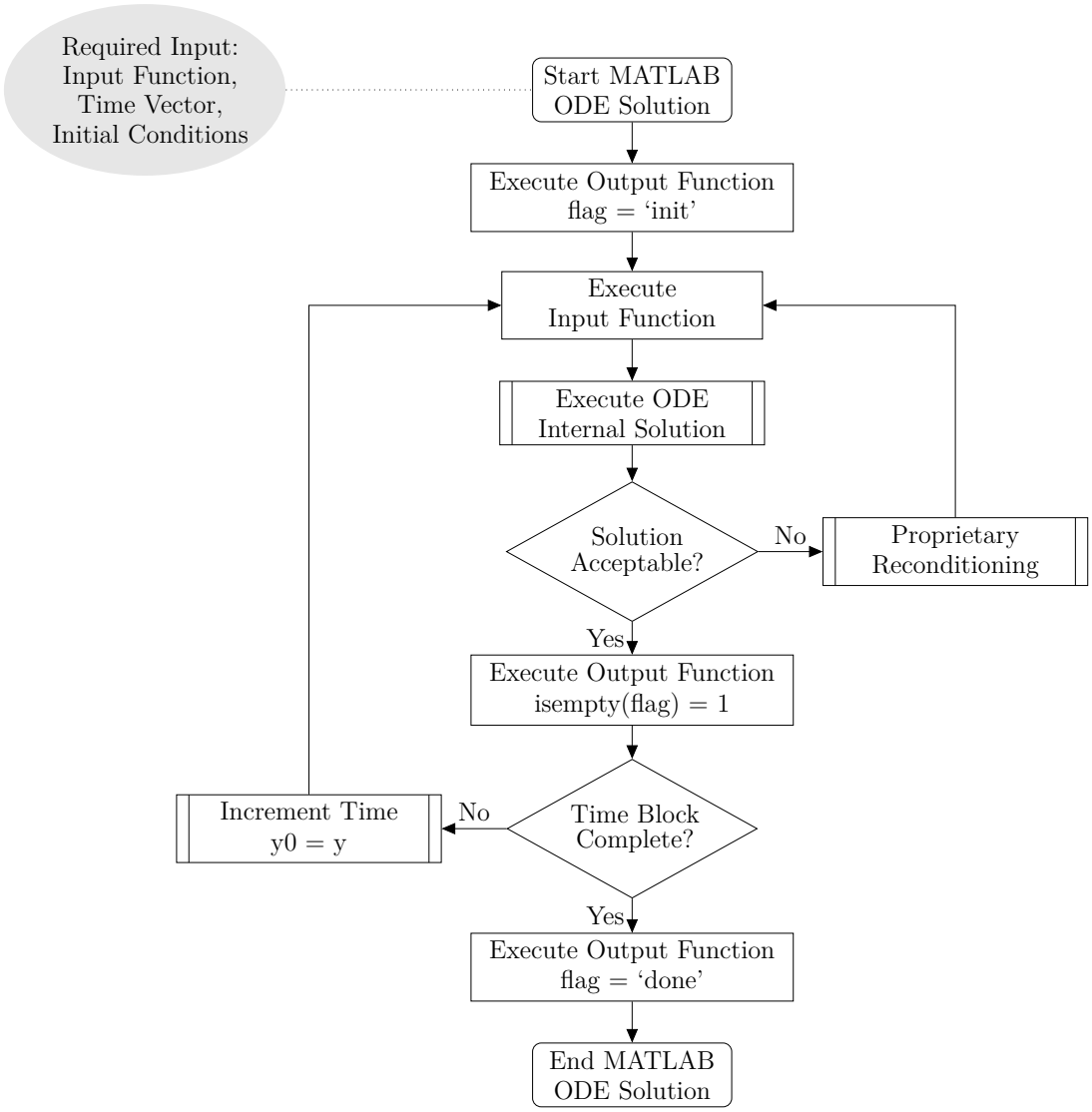


Figure 3.7: MATLAB ODE Flow Chart.

The input to an ODE solver include, an input function, a time interval (time block), initial conditions, and solver options. The current options used for VTS that deal with error tolerance levels, initial step size, max step size, and an optional output function are shown in Listing 3.34.

Listing 3.34: MATLAB ODE Solver Option Example

```
% Configure ODE settings
%options = odeset('RelTol',1e-3,'AbsTol',1e-6); % MATLAB default settings
options = odeset('RelTol',1e-4,'AbsTol',1e-7, ...
    'InitialStep', 1/60/4, ...
    'MaxStep',60, ...
    'OutputFcn',outputFcn); % set 'OutputFcn' to function handle
```

3.17.3 Functions Specific to VTS

A number of new functions were created to handle data or perform other tasks so that VTS could be integrated into PST. The following sections provide information about such functions.

3.17.3.1 vtsInputFcn

MATLAB ODE solvers require a passed in function that returns a vector of derivatives associated with states to integrate. The `vtsInputFcn` was created to perform this critical task. The slightly abbreviated input function is shown in Listing 3.35.

Listing 3.35: Abbreviated vtsInputFcn

```
function [dxVec] = vtsInputFcn(t, y)
% VTSINPUTFCN passed to ODE solver to perfrom required step operations
%
%   NOTES: Updates and returns g.vts.dxVec
%
%   Input:
%   t - simulation time
%   y - solution vector (initial conditions)
%
%   Output:
%   dxVec - requried derivative vector for ODE solver

global g

handleStDx(g.vts.dataN, y, 2)        % update states at g.vts.dataN with y

initStep(g.vts.dataN)
networkSolutionVTS(g.vts.dataN, t)
dynamicSolution(g.vts.dataN )
dcSolution(g.vts.dataN )

if g.vts.iter == 0
    handleNetworkSln(g.vts.dataN ,1) % save first network solution
end

g.vts.iter = g.vts.iter + 1;         % increment solution iteration counter

handleStDx(g.vts.dataN , [], 1)      % update g.vts.dxVec
dxVec = g.vts.dxVec;                 % return updated derivative vector
end % end vtsInputFcn
```

3.17.3.2 vtsOutputFcn

After each acceptable solution, the ODE solver calls a passed in *output function*. The `vtsOutputFcn` was created to handle restoring the original network solution, performing the monitor solution, indexing, iteration accumulation, and logging after each step. The slightly abbreviated VTS output function is shown in Listing 3.36.

Listing 3.36: Abbreviated vtsOutputFcn

```
function status = vtsOutputFcn(t,y,flag)
% VTSOUTPUTFCN performs associated flag actions with ODE solvers.
%
%   Input:
%   t - simulation time
%   y - solution vector
%   flag - dictate function action
%
%   Output:
%   status - required for normal operation (return 1 to stop)

global g
status = 0; % required for normal operation

if isempty(flag)                    % normal step completion
    handleNetworkSln(g.vts.dataN ,2)% restore network to initial solution
    monitorSolution(g.vts.dataN);   % Perform Line Monitoring and Area Calculations

    if g.sys.livePlotFlag
        livePlot(g.vts.dataN)       % Live plotting
    end

    g.vts.dataN = g.vts.dataN+1;    % increment logged data index 'dataN'
    g.sys.t(g.vts.dataN) = t;       % log step time
    g.vts.stVec = y;                % update state vector
    handleStDx(g.vts.dataN, y, 2)   % place new solution results into associated globals

    g.vts.tot_iter = g.vts.tot_iter + g.vts.iter;   % update total iterations
    g.vts.slns(g.vts.dataN) = g.vts.iter;           % log solution step iterations
    g.vts.iter = 0;                                 % reset iteration counter

elseif flag(1) == 'i'               % init solver for new time block
    g.sys.t(g.vts.dataN) = t(1);    % log step time
    handleStDx(g.vts.dataN, y, 2)   % set initial conditions

elseif flag(1) == 'd'               % only debug screen output at the moment

end % end if
end % end function
```

3.17.3.3 handleNetworkSln

The `handleNetworkSln` function was created to store, and restore, calculated values set to global variables during a network solution. The purpose of this function was to allow for the first network solution performed each step to be carried forward after multiple other network solutions may over-write the calculated values at the same data index. This over-writing would occur during the MATLAB ODE solver's repeated calls to the input function. As shown in Listing 3.37, `handlNetworkSln` takes a data index `k` and an operation `flag` as inputs.

Listing 3.37: Function Header for handleNetworkSln

```
function handleNetworkSln(k, flag)
% HANDLENETWORKSLN saves or restores the network solution at data index k
%
%   NOTES: Used to reset the newtork values to the initial solution in VTS.
%
%   Input:
%   k - data index to log from and restore to
%   flag - choose funtion operation
%       0 - initialize globals used to store data
%       1 - collect newtork solution values from index k into global vector
%       2 - write stored network solution to globals at index k
```

3.17.3.4 handleStDx

The `handleStDx` function was created to perform the required state and derivative handling to enable the use of internal MATLAB ODE solvers. Its general operation is probably best described via the internal function documentation shown in Listing 3.38.

Listing 3.38: Function Header for handleStDx

```
function handleStDx(k, slnVec, flag)
% HANDLESTDX Performs required state and derivative handling for ODE solvers
%
%   NOTES:  Requires state and derivative values are in the same g.(x) field.
%           Not all flags require same input.
%
%   Input:
%   k - data index
%   flag - choose between operations
%           0 - initialize state and derivative cell array, count states
%           1 - update g.vts.dxVec with col k of derivative fields
%           2 - write slnVec vector of values to associated states at index k
%           3 - update g.vts.stVec with col k of state fields
%   snlVec - Input used to populate states with new values
```

The new global structure created in PST 4 enables `handleStDx` to complete the stated operations by relying heavily on dynamic field names. Essentially, all required field names, sub-field names, and states are collected into a cell (flag operation 0) that is then iterated through to collect data from, or write data to the appropriate location (all other flag operations).

The usefulness of `handleStDx` is that the standard MATLAB ODE solvers require a single derivative vector as a returned value from some passed in 'input function', and each PST model calculates derivatives and places them into various global variables. Thus, a derivative collection algorithm was needed (flag operation 1). Once the ODE solver finishes a step, the returned solution vector (of integrated states) must then be parsed into the global state variables associated with the supplied derivatives (flag operation 2). At the beginning of time blocks that use the MATLAB ODE solvers, an initial conditions vector of all the

states related to the derivative vector is a required input (updated via flag operation 3). To avoid handling function output, global vectors `g.vts.dxVec` and `g.vts.stVec` are used to hold updated derivative and state vector information.

It should be noted that original PST globals follow the same data structure, however, new models (such as AGC, pwrmod, and ivmmmod) use a slightly different data structure and must be handled in a slightly different way. As of this writing AGC, pwrmod, and ivmmod functionality has been added to `handleStDx`. Additional models that require integration may be added to this function as the need arises.

4 Examples

Examples are located in the `0-examples` folder and are designed to be run in batch mode. This means that each example copies any required system, model, or modulation file to the main PST directory before running. Typically, each example folder contains a `.m` file that starts with `run_` that can be used to run the example. As example cases are located on a github repository, relative file paths are used so that examples can be run without much difficulty from various different machines. While most examples can be run in multiple versions of PST, some functionality can only be found in specific versions.

4.1 Example Operation Overview

The general structure of created examples tend to reflect the following:

1. Clear all variables, close all figures, and clear the command window.
2. Define which version of PST to use.
3. Create a relative path to the root directory of chosen PST version.
4. Copy any system, model, or modulation file to the PST root directory.
5. Run `s_simu`.
6. Save output data.
7. Restore any model or modulation file to original state.
8. Create data plots.

A high level flow chart of the `s_simu` script is presented in Appendix A. It should be noted that in `s_simu`, an optional `g.sys.DEBUG` flag may be set to 1 to display more output to the console.

4.2 Standard Faulting (hiskens)

Standard PST simulations involve some kind of fault defined in the `sw_con` array. This example (located in the `hiskens` folder) is included to showcase some simple differences between PST versions using a standard test case. System data for this example comes from a report by Ian Hiskens [6] which summarized a study of an IEEE 10 generator, 39 bus system. Ryan Elliott at Sandia National Labs recreated the system in a PST format and provided the data file for this project. A one-line diagram of the system is shown in Figure 4.1.

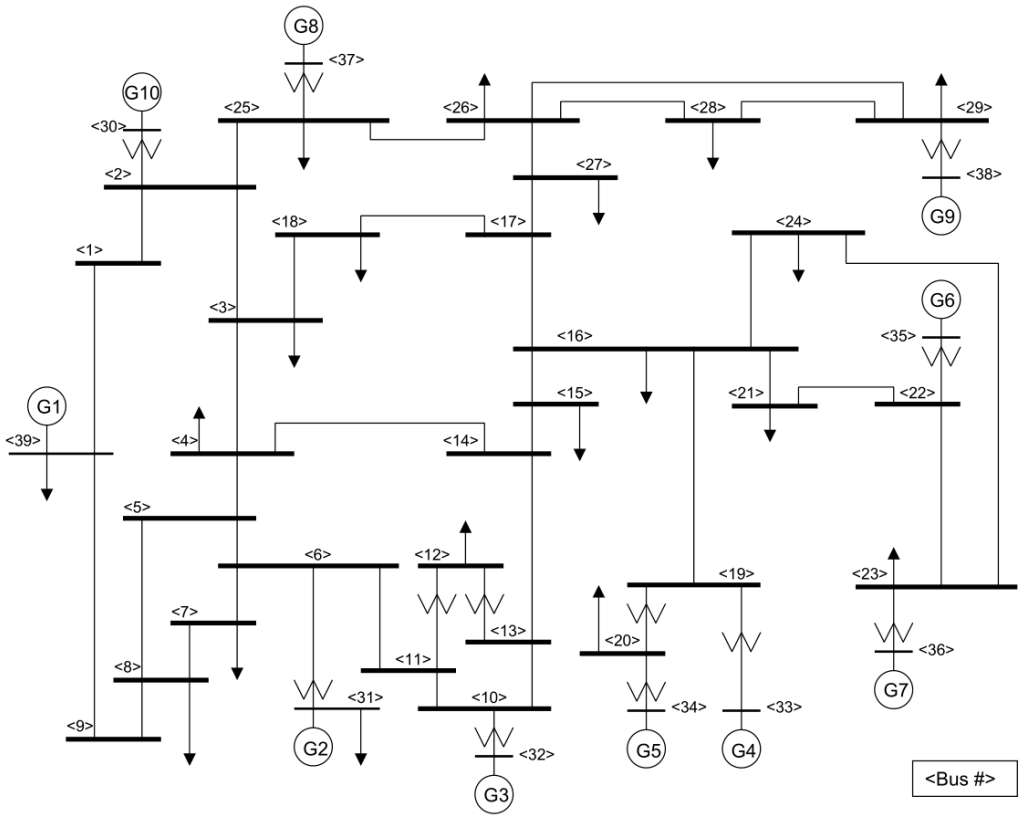


Figure 4.1: IEEE 39 bus network.

The simulated event was a 0.1 second three-phase fault with no loss of line between bus 2 and 3. The `run_datane_hiskens.m` file can be used to run the simulation. Figure 4.2 shows the resulting fault bus voltage magnitude and system generator speeds.

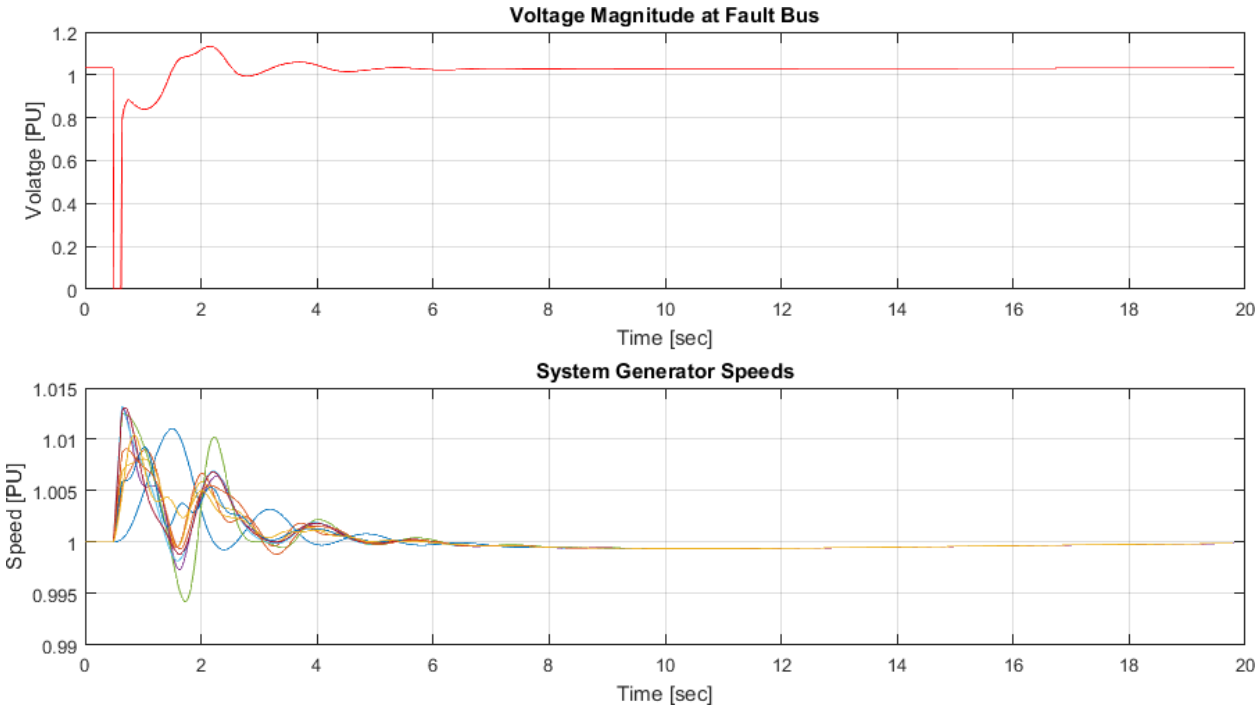


Figure 4.2: Fault Bus Voltage and Generator Speeds from Hiskens Example.

All PST versions (when using the same models) provide the same results, but there are differences in simulation speed and data output. Table 4.1 shows that the PST 4 simulation was roughly twice as fast as PST version 2 or 3, saved less data, and left fewer variables in the MATLAB workspace post simulation. These improvements are likely due to the restructuring of global variables and code to remove any 'all zero' data from being saved.

Table 4.1: PST Version Comparisons of Hiskens Example.

PST Version	Simulation Time [seconds]	Resulting Workspace Variables	Saved Data Size [KB]
2.3	16.56	206	7,549
3.1	16.70	210	7,548
SETO	8.42	24	3,965
4.0	7.96	6	3,974

4.3 Modulation Examples

PST provides numerous ways to input a defined modulation signal into a variety of models. The following list of example folder names work in all versions of PST and typically create linear/non-linear comparison plots. While the examples themselves do not reflect any particular scenario, the provided code may be useful to create a particular scenario.

- `lmod` - Replaces the `ml_sig` file to modulate real load power.
- `mexc` - Replaces the `mexc_sig` file to modulate the exciter reference signal.
- `mtg_sig` - Replaces the `mtg_sig` file to modulate the governor P_{ref} reference signal.
- `pm_sig` - Replaces the `mpm_sig` file to modulate a machines mechanical power.
- `rlmod` - Replaces the `rml_sig` file to modulate reactive load power.
- `SVC` - Replaces the `msvc_sig` file to modulate an SVC's output value.
- `TCSC` - Replaces the `mtcsc_sig` file to modulate an TCSC's output value.

It should be noted that modulation files for PST 2 and 3 assume input to function is (t, k), i.e. full time vector and data index, while PST 4 only requires the data index k.

4.4 AGC - WIP

Automatic generation control (AGC) has been added only to PST 4. As such, only PST 4 can run the AGC examples. Both AGC examples use the modified Kundur 4 machine model shown in Figure 4.3.

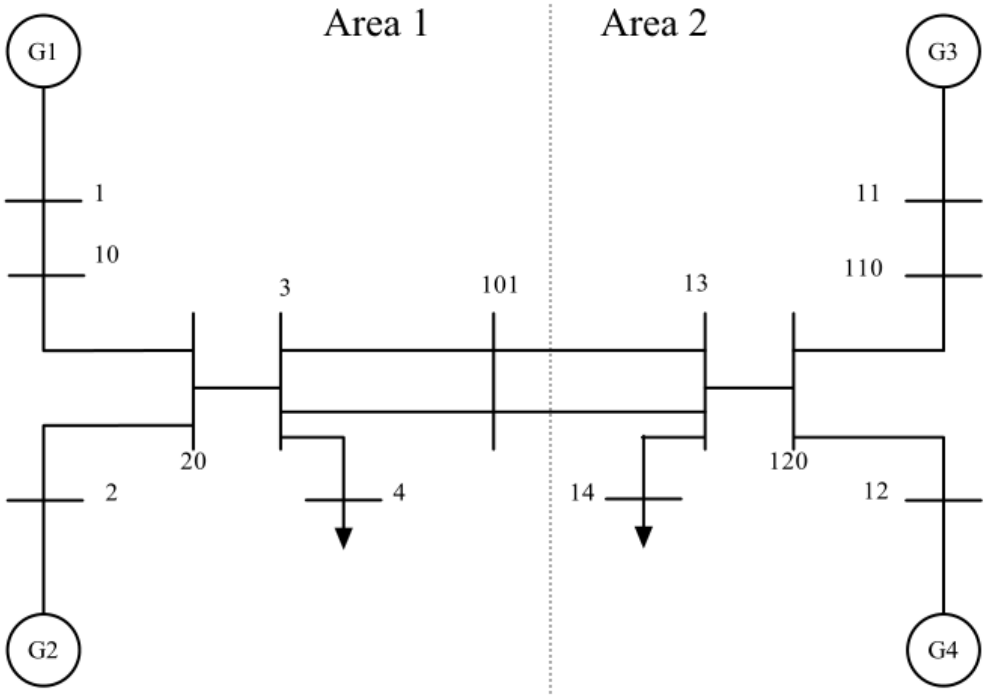


Figure 4.3: One-Line used in run_AGC.

4.4.1 run_AGC - WIP

This example was designed to explore how two area's AGC routine would respond to a load step. At $t = 2$, a 0.5 PU real load step occurs on Bus 4 in Area 1. AGC action was scheduled to begin at $t = 25$ and then again every 15 seconds there after.

4.4.1.1 run_AGC Result Summary - WIP

The 120 second simulation is shown to capture transients due to the load step, governor action arresting frquency change, and then AGC acting to restore system frequency and interchange values.

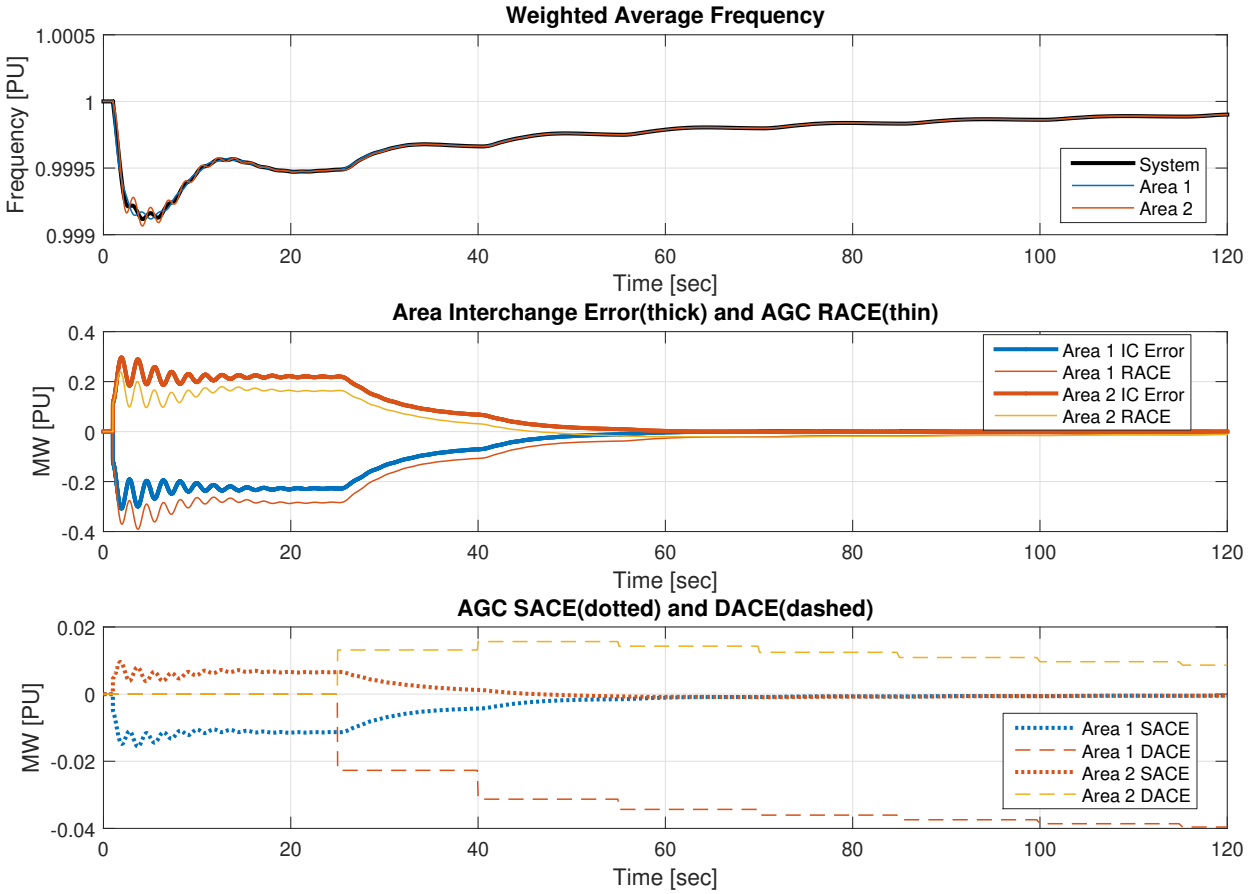


Figure 4.4: Final live Plot output from run_AGC.

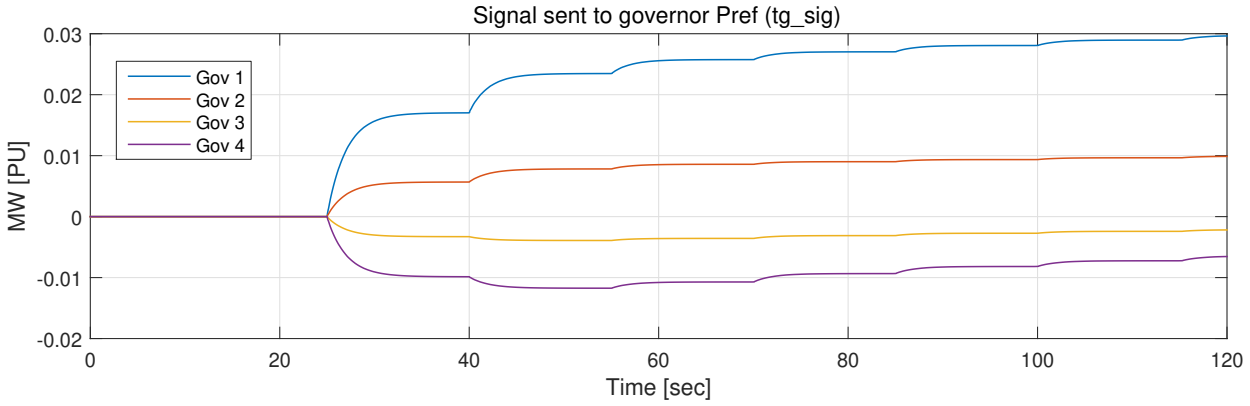


Figure 4.5: Governor modulation signals sent in run_AGC.

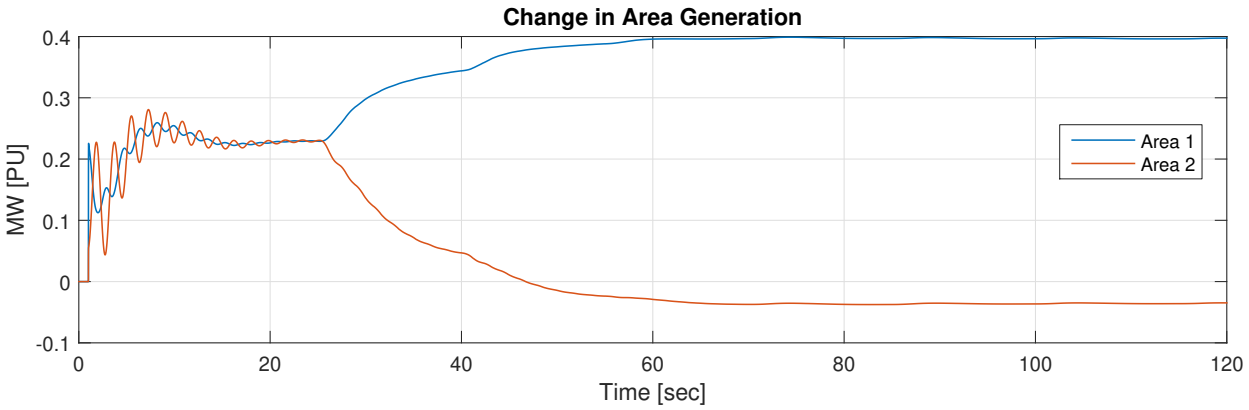


Figure 4.6: Change in area generation during run_AGC.

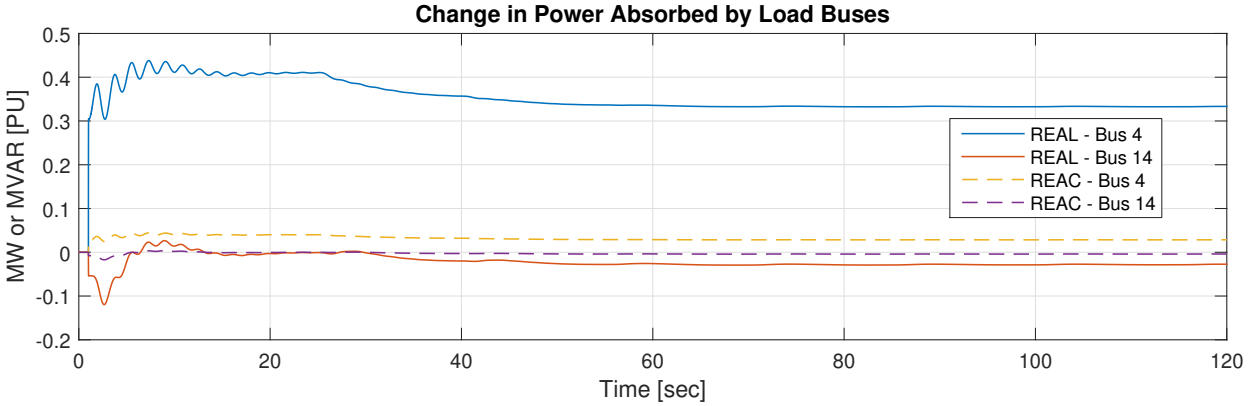


Figure 4.7: Power absorbed by loads during run_AGC.

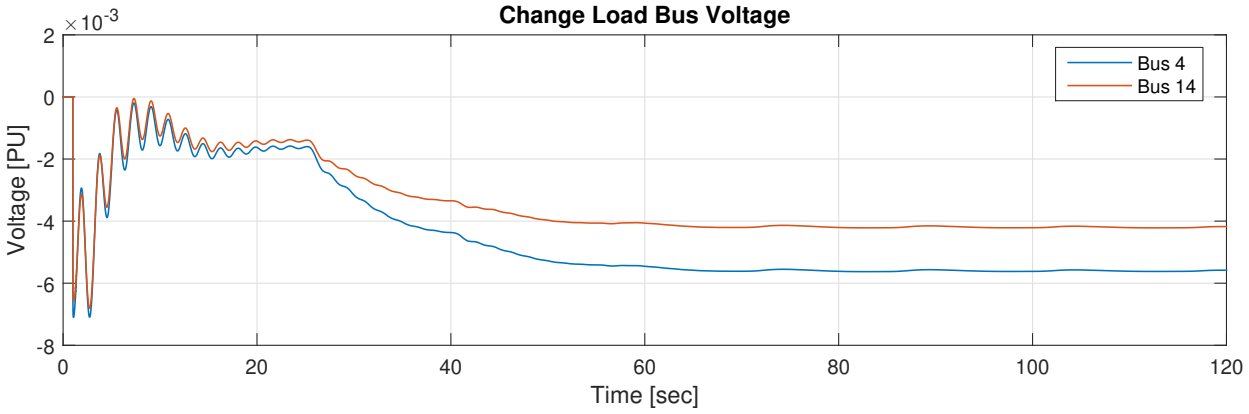


Figure 4.8: Load bus voltage change during run_AGC.

4.4.2 run_AGC_mod - WIP

An extended term simulation may required the adjustment of scheduled interchange to achieve system recovery. Specifically, if an area realizes that their available reserves become lower than was originally allocated for, a resolution may be to import more power from another area. Functionality to accomplish such a task was added via a user defined `mAGC_sig` fuction. Similar to other `m*_sig` files, it is executed every time step and allows for setting an area's `icAdj` value to any value.

The `run_AGC_mod` example adjusts the interchange modulation signal as a perturbance and allows AGC to re-balance line flows accordingly. As a reminder, the one-line diagram of the system used is shown in Figure 4.3. When $t = 5$, Area 2 increased scheduled interchange by 0.2 PU while Area 1 interchange is adjusted by -0.2 PU to keep system in balance. Each area has non-conditional AGC set to act every 15 seconds and is forced to act by user defined code in `mAGC_sig` when the interchange adjustment first takes place. The example showed how Area 2 increased generation while Area 1 reduced generation in response to the interchange modulation signal.

4.4.2.1 run_AGC_mod Result Summary - WIP

Interchange adjustment worked as expected and was accounted for in AGC calculations. The use of `mAGC_sig` was tested as working using FTS or VTS. The `mAGC_sig` file used to adjust interchange and force AGC action is shown as Listing 4.1.

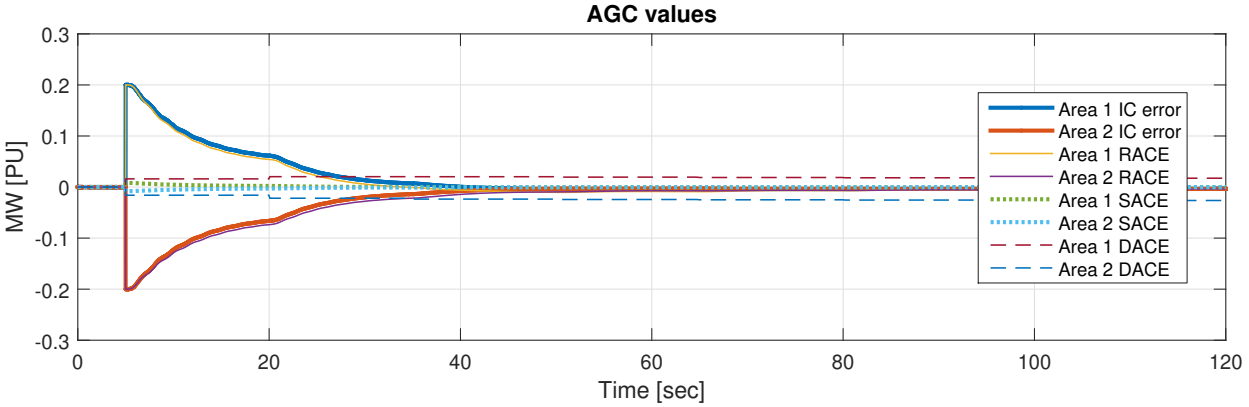


Figure 4.9: AGC signals from run_AGC_mod example.

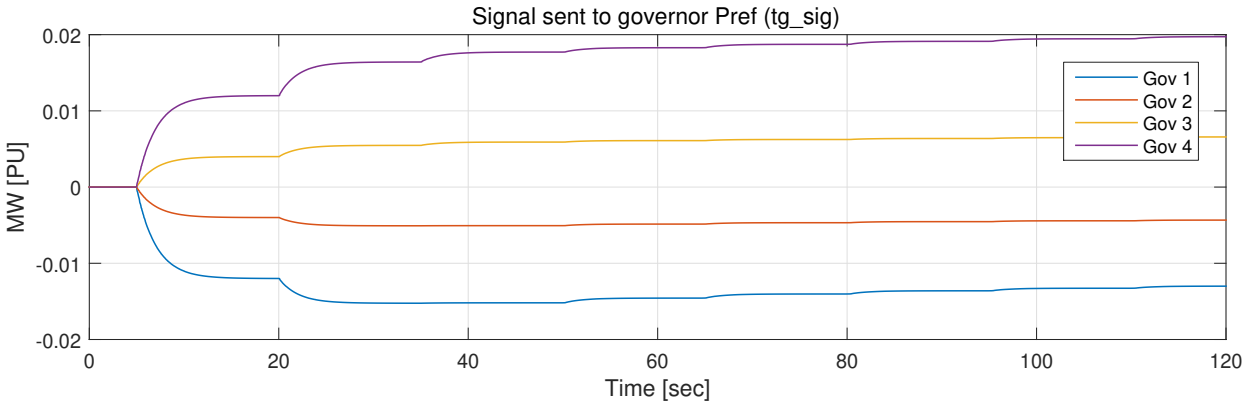


Figure 4.10: Turbine governor modulation signals from run_AGC_mod.

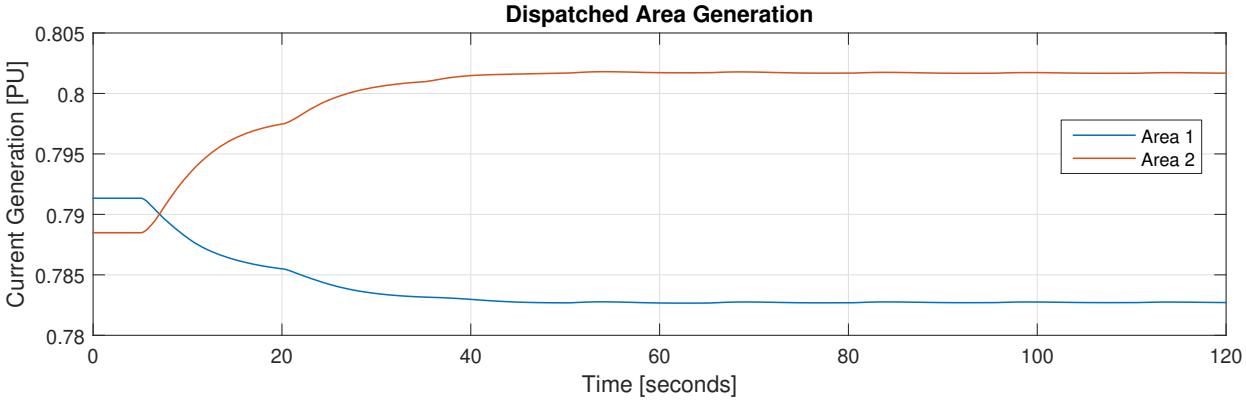


Figure 4.11: Change of dispatched area generation during run_AGC_mod.

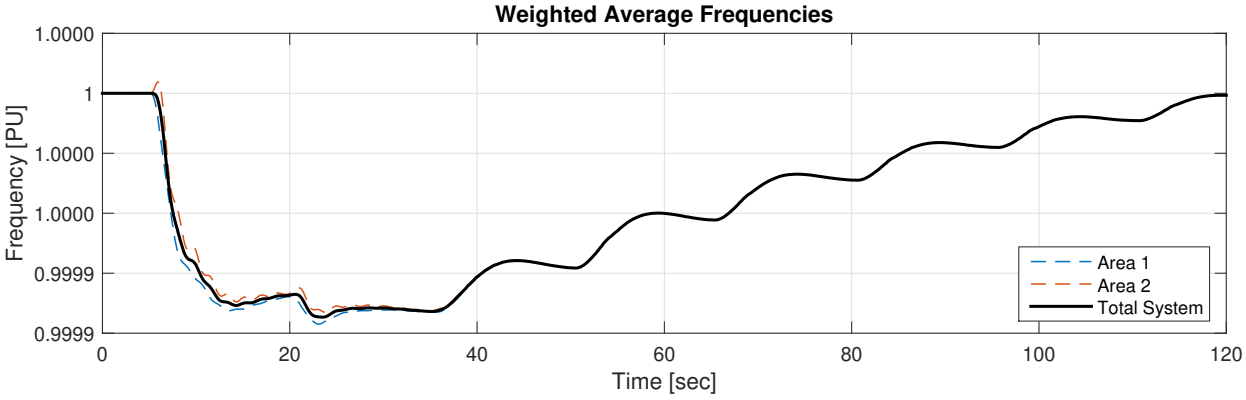


Figure 4.12: Change in system frequency during run_AGC_mod.

Listing 4.1: AGC Modulation Example

```
function mAGC_sig(k)
% Syntax: mAGC_sig(k)
% input k is current data index
% 09:46 08/21/20
% place to define modulation signal for AGC operation

global g

%{
    Scenario:
Area 1 is exporting generation to Area 2 (Interchange value Positive)
Area 2 is importing power from Area 1 (Interchange value is Negative

Area 2 increases scheduled interchage, which reduces its scheduled import and causes
↪  area 2 to increase generation.
Area 1 decreases scheduled interchange to balance area 2 action.
As area 1 is exporting, the negative valued icAdj will reduce the generation in the
↪  area.
%}
persistent ForceDisptach

if g.sys.t(k) >= 5
    % adjust iterchange
    g.area.area(2).icAdj(k) = 0.2;
    g.area.area(1).icAdj(k) = -0.2;

    % force AGC disptatch when interchange adjustment first applied
    if ForceDisptach
        g.agc.agc(1).nextActionTime = g.sys.t(k);
        g.agc.agc(2).nextActionTime = g.sys.t(k);
        ForceDisptach = 0;
    end
else
    g.area.area(2).icAdj(k) = 0;
    g.area.area(1).icAdj(k) = 0;
    ForceDisptach = 1;
end
end
```

4.5 extendedTerm - WIP

(sloppily copied from 200806-ExtendedVersionComp)

4.5.1 Scenario - WIP

- Kundur 4 machine system packaged with PST - Same as used in AGC examples...
- Constant Z load model
- System has governors, exciters, and PSS.
- Governor of generator being perturbed by pm_sig removed
- Perturbance was meant to mimic a solar ramp with various situations of cloud cover:

```
% time [seconds]
% 0-30      - no action
% 30-90     - ramp up 0.5 PU (50 MW)
% 90-150    - hold peak
% 150-210   - ramp down 0.5 PU (50 MW)
% 210-240   - no action

% cloud cover events
% 45-55 - 20% max gen (generation of 0.1 PU)
% 120-140 - 30% cover (generation reduction to 70%)
% 180-190 - 15% cover (generation reduction to 85%)
```

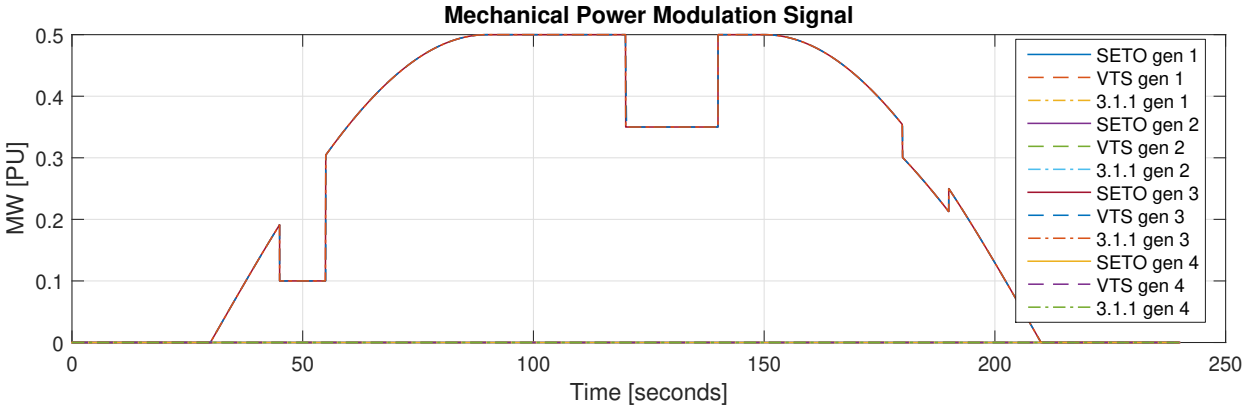


Figure 4.13: Modulation signal extended example.

4.5.2 Result Summary - WIP

1. PST 4 is 2.6 times faster than PST 3.1.
2. Using variable time steps allows for a speed up of 5.9 over PST 3.1.
3. Results from all simulations are very similar.
4. Without creating an explicit time block at the beginning of an event, VTS events may not occur at the exact time they are programmed.
5. VTS reduces logged data size by ≈3 times.

Table 4.2: PST Version Comparisons of Extended Term Example.

	Step Size [seconds]				Solutions Per Step				
PST Version	Max.	Min.	Ave.	Total Steps	Ave.	Max.	Total Slns.	Sim. Time	Speed Up
3.1	4.00E-03	4.00E-03	4.00E-03	59,975	2	2	119,950	740.35	1.00
SETO	4.00E-03	4.00E-03	4.00E-03	59,975	2	2	119,950	284.37	2.60
4.0 - VTS	23.2	1.36E-04	1.38E-02	17,353	2	96	27,243	125.53	5.90

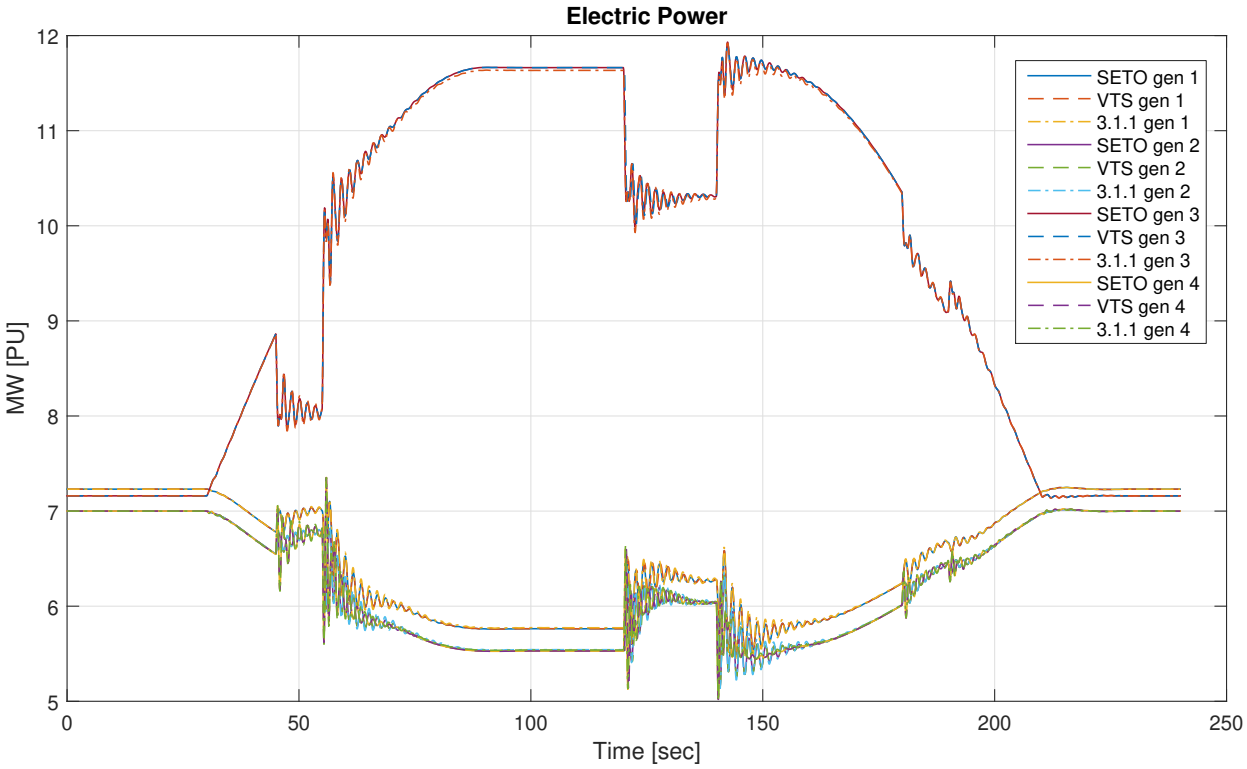


Figure 4.14: Electric Power from extended example.

NOTE: Initial time steps before t=30 are much larger than the other time steps (multiple seconds) and are plotted off the axis.

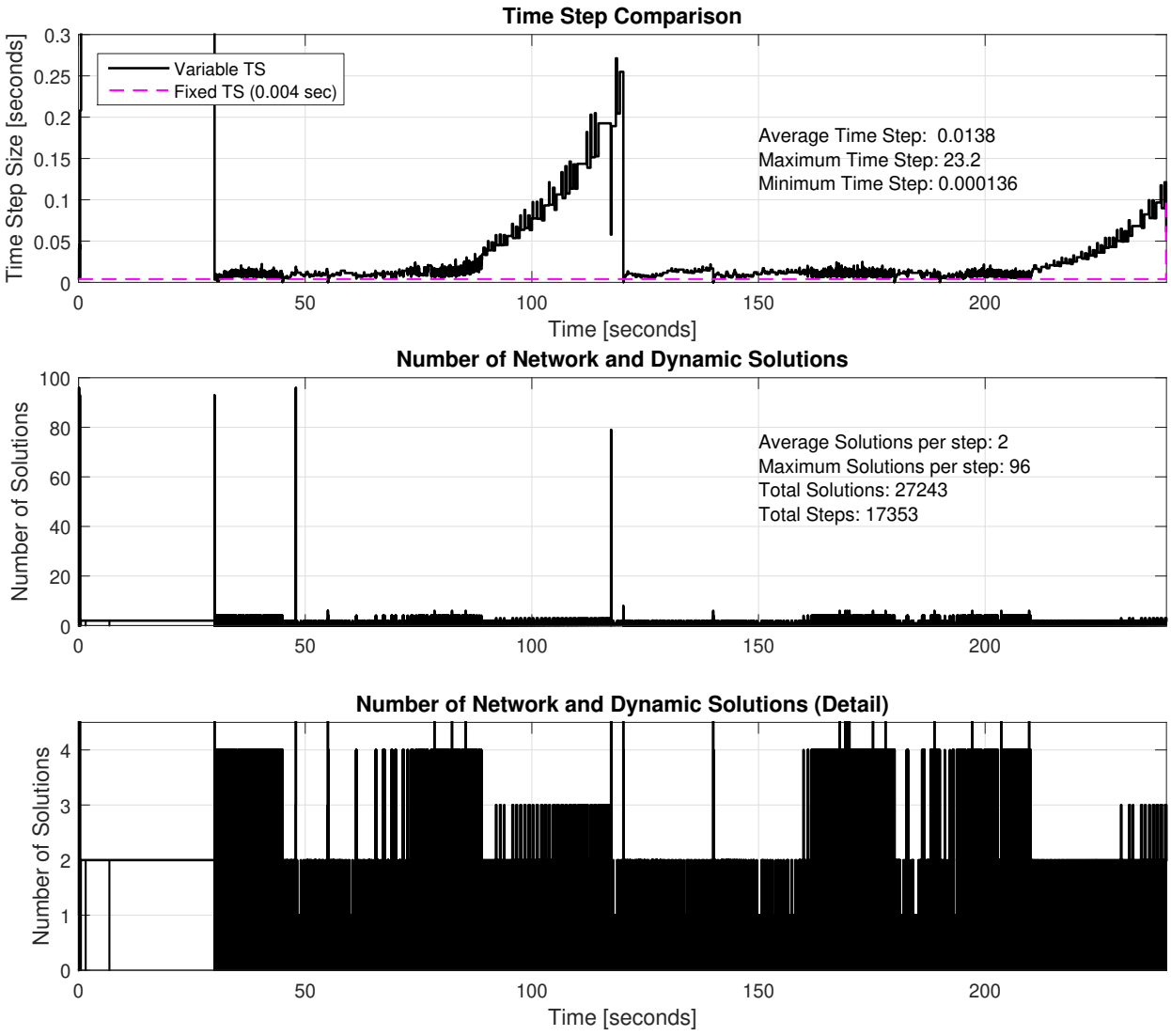


Figure 4.15: Step Size / Number of Solutions Comparison extended.

NOTE: 3.1 does not calculate average system frequency.

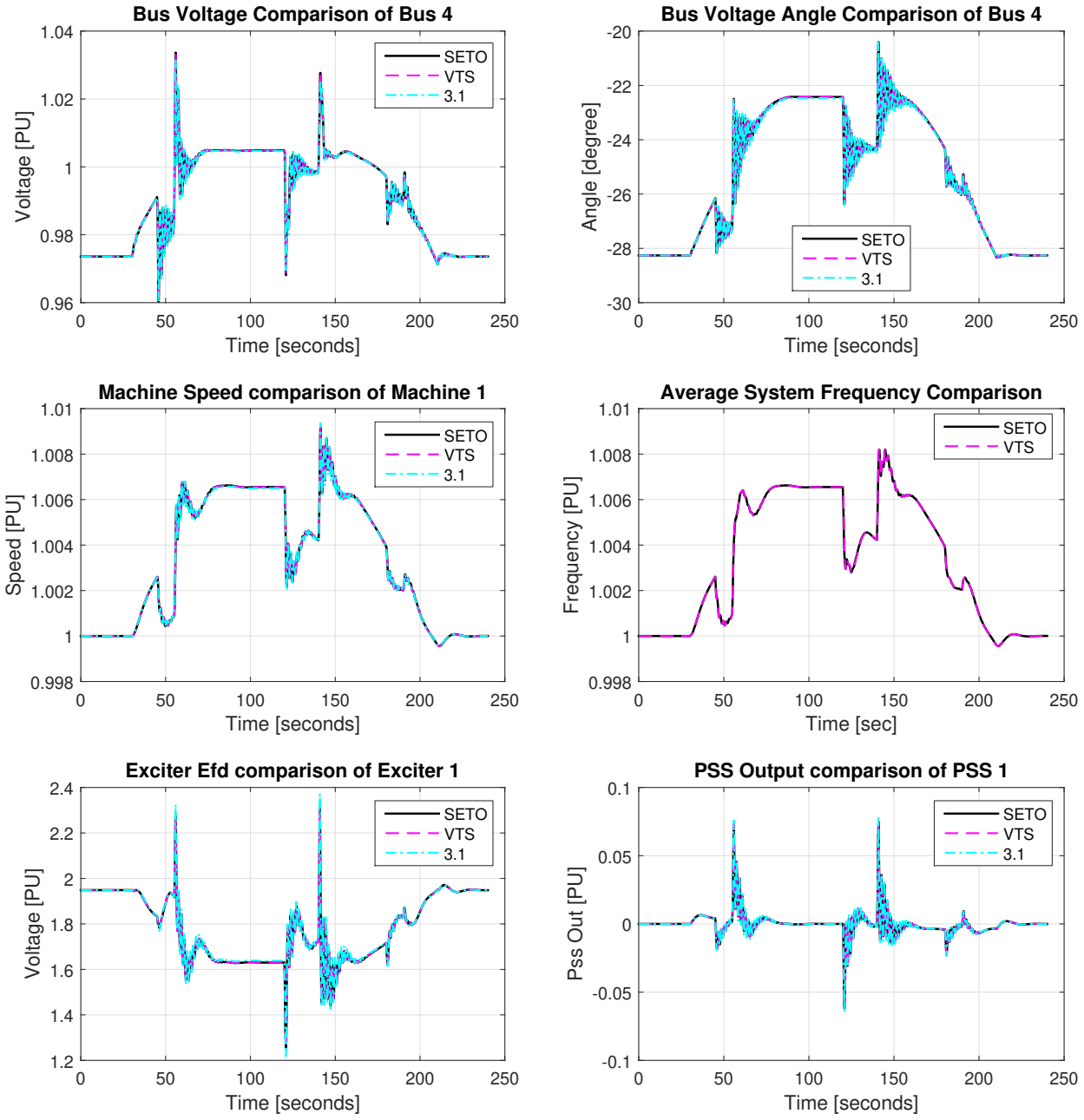


Figure 4.16: Various Comparisons extended.

NOTE: PST 3.1 uses the `pss3` model while other PST versions use `pss2`

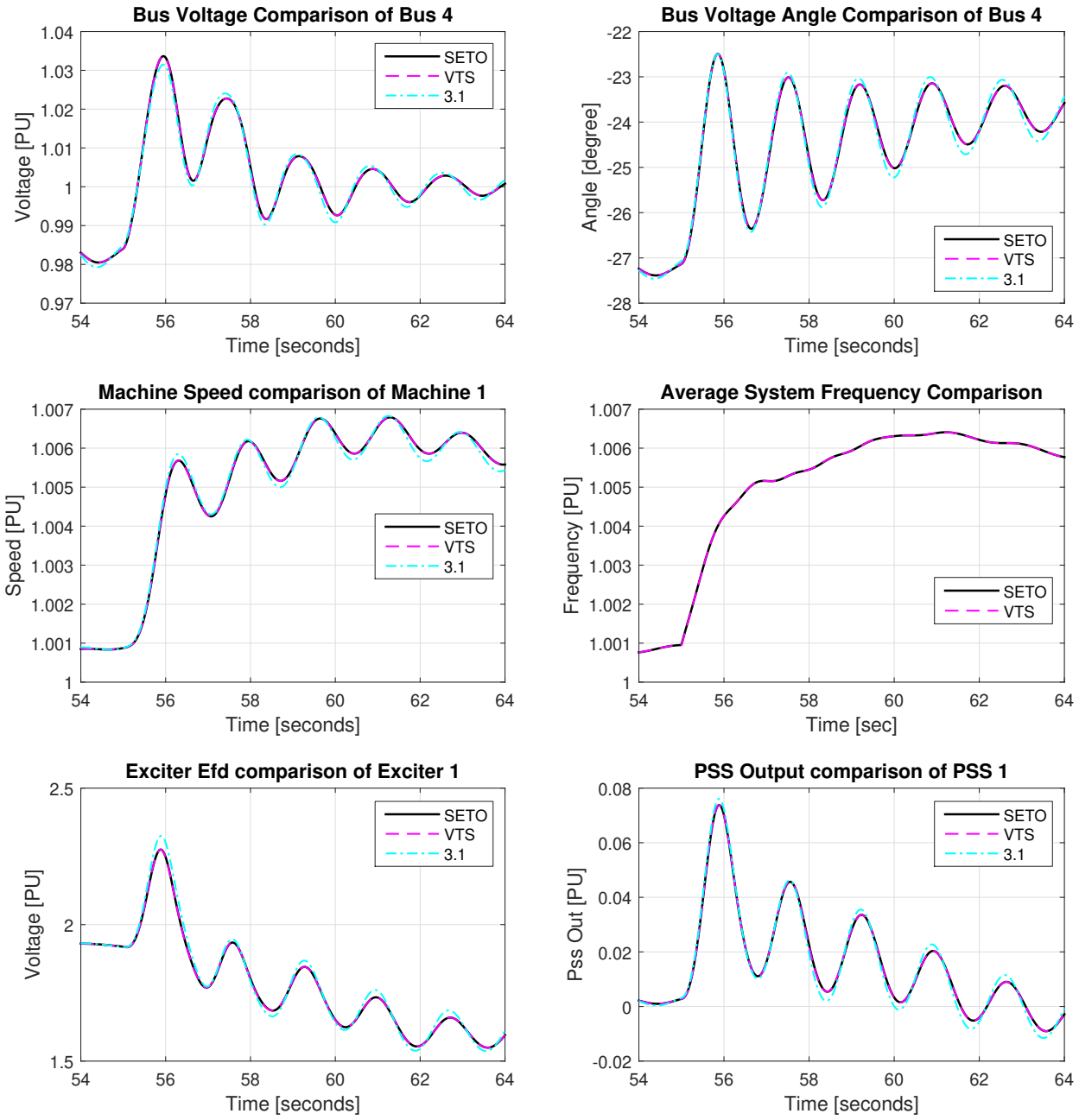


Figure 4.17: Various Comparisons - Detail 1 extended.

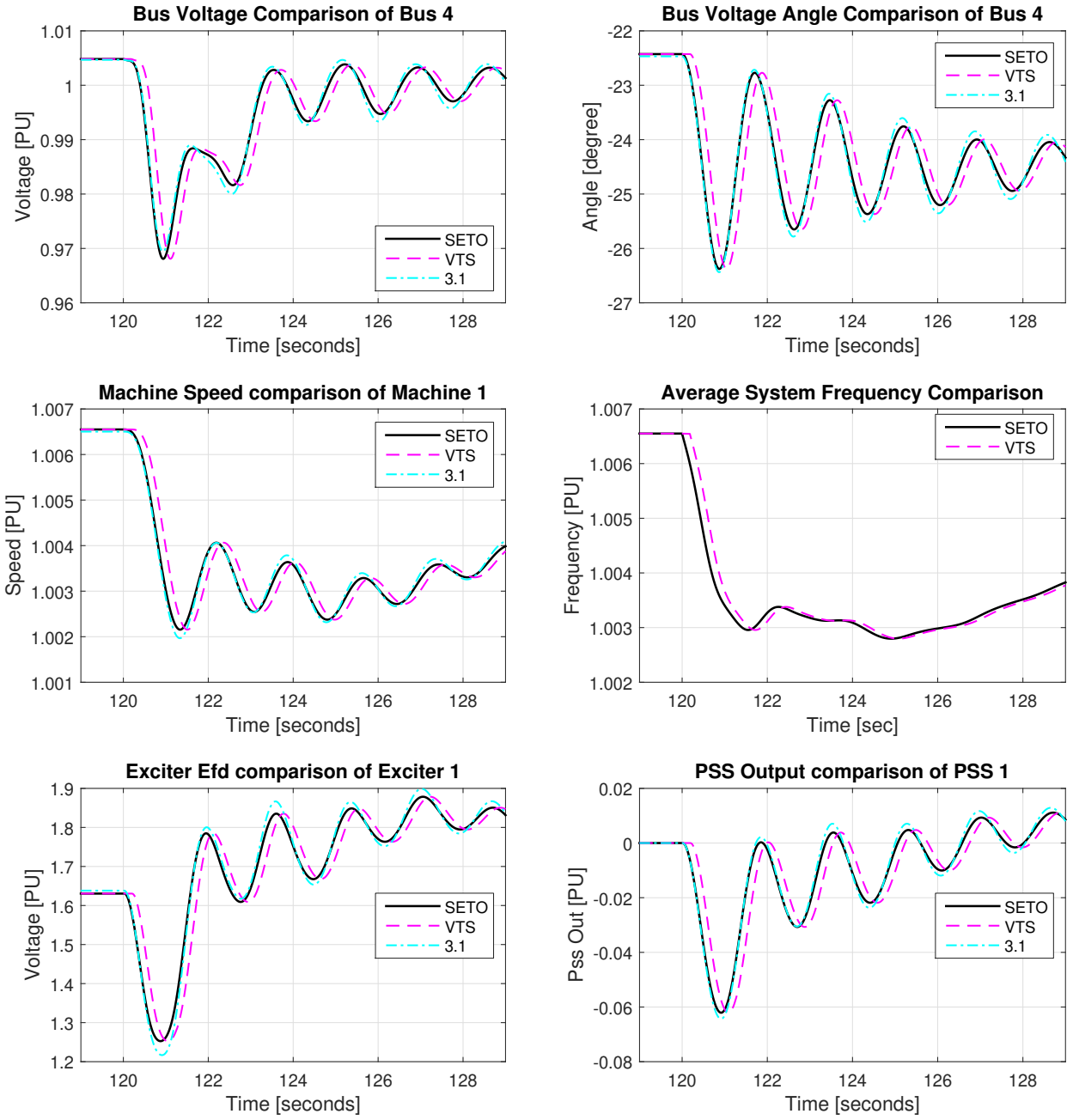


Figure 4.18: Various Comparisons - Detail 2 extended.

Figure 4.18 NOTE: VTS events may not occur at exact specified time due to the nature of variable time steps. Additional entries in the `sw_con` can be created to account for this. For example, creating an entry for every known perturbance can ensure the exact execution time.

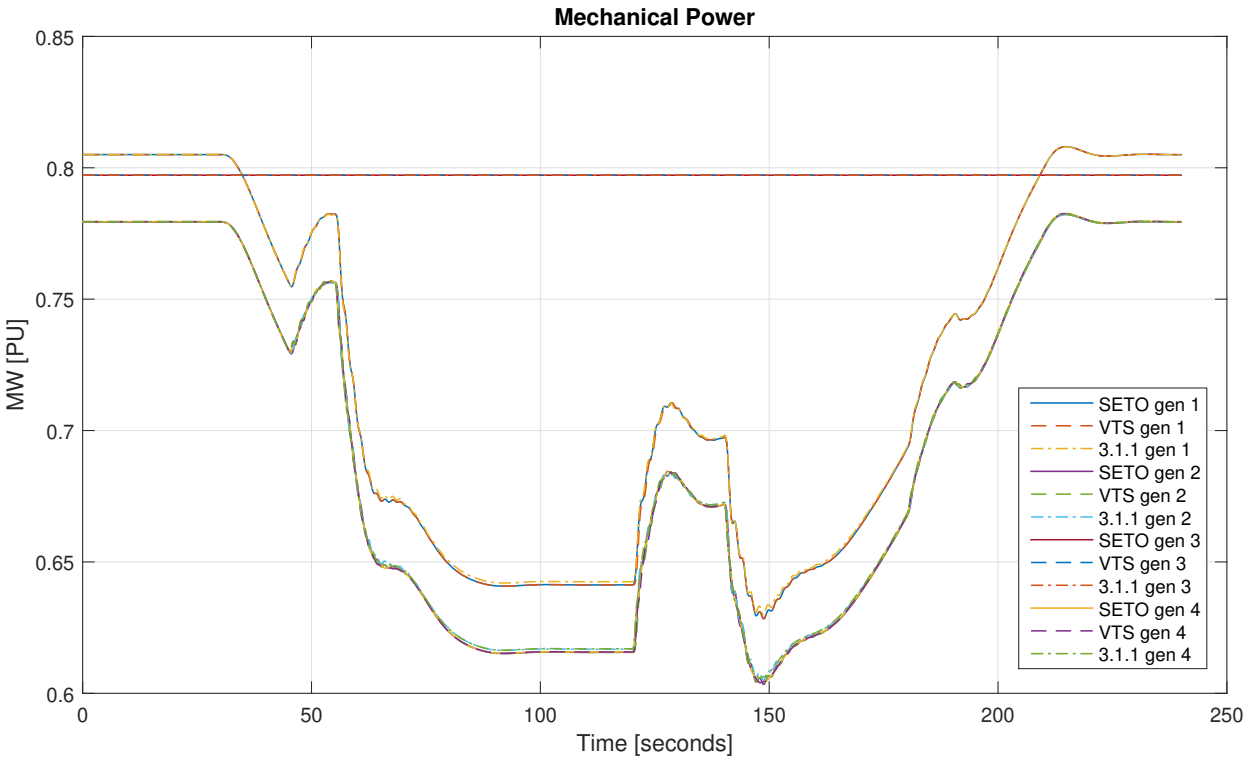


Figure 4.19: Mechanical Power extended.

PST does not add the modulated mechanical power signal the recorded mechanical power. Instead, the modulation signal is added to the derivative of machine speed.

4.6 ivmmod - WIP

Adapted from Dr. Trudnowski example. Shows how angle change 'moves' entire system. VTS option shows how variable time step method will adjust to capture dynamics.

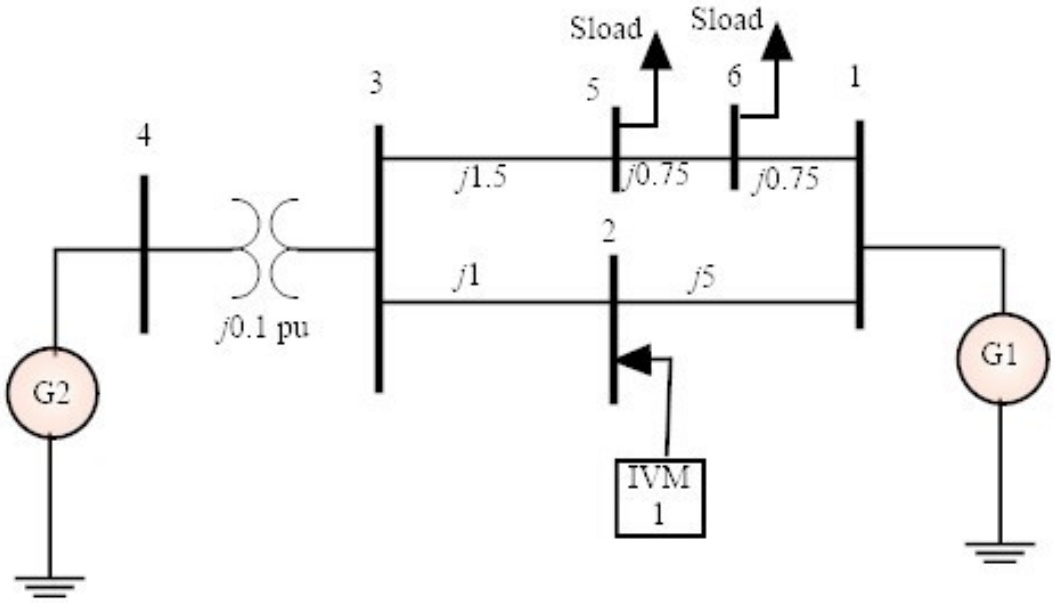


Figure 4.20: System one-line diagram from run_IVM.

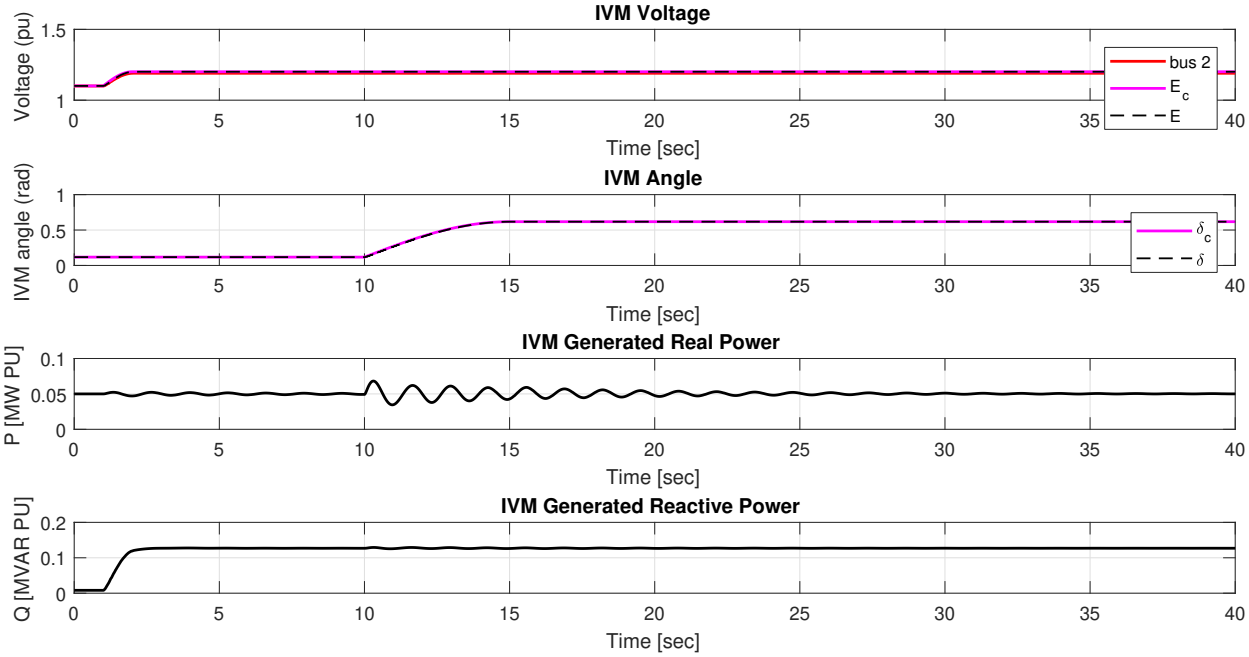


Figure 4.21: IVM action and resulting generator power output from run_IVM.

All system angles change when the angle is perturbed at $t = 10$. The desired outcome would be for the controlled machine to generate more real power, however, this is not the case.

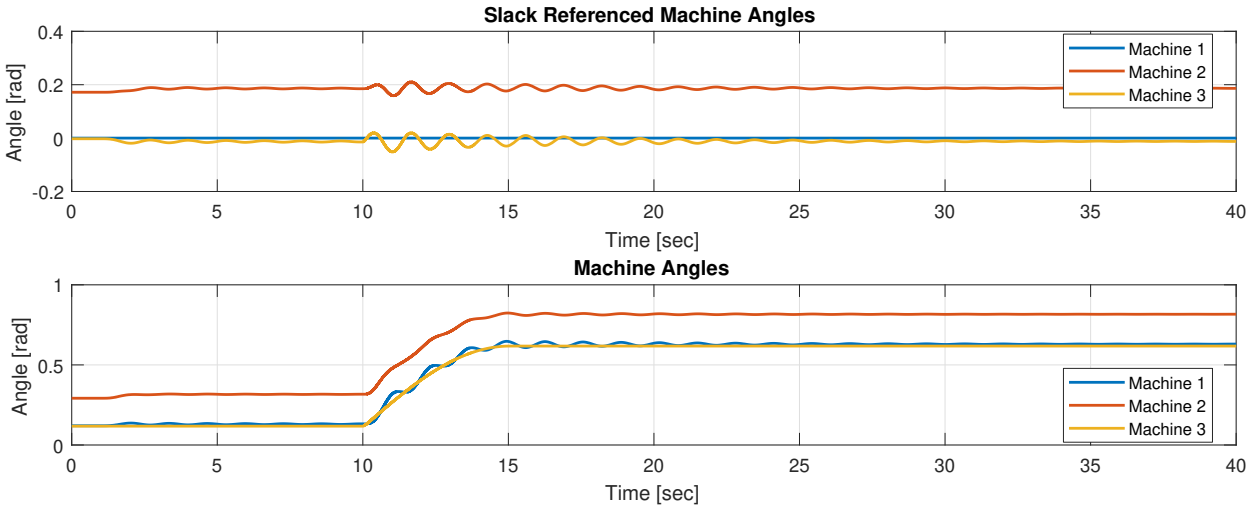


Figure 4.22: Machine angle comparisons from run_IVM.

4.7 miniWECC - WIP

The miniWECC is a 120 bus 34 generator system designed to simulate the western US interconnect. The original miniWECC was created without areas however, two slightly different multi-area models have been created. One mirrors the divisions used by [5], while the other attempts to recreate the EIA area boundaries. One-line diagrams of both configurations are presented in Appendix B.

4.7.1 miniWECC-genTrip - WIP

Example shows multiple generator trips in one simulation and the use of the miniWECC system in all versions. Main differences between 3.1 and 4 is overall simulation speed (PST 4 is ≈2x faster), the zeroing of tripped machine derivative, and system inertia calculations.

Table 4.3: PST Version Comparisons of MiniWECC Generator Trip Example.

	Step Size [seconds]				Solutions Per Step				
PST Version	Max.	Min.	Ave.	Total Steps	Ave.	Max.	Total Slns.	Sim. Time	Speed Up
3.1	8.33E-03	8.33E-03	8.33E-03	2401	2	2	4802	42.87	1.00
4.0	8.33E-03	8.33E-03	8.33E-03	2401	2	2	4802	21.89	1.96

PST 3 does not zero derivatives of tripped machine speeds. As a result, tripped machine speeds increase greatly despite not having any effect on the rest of the connected system.

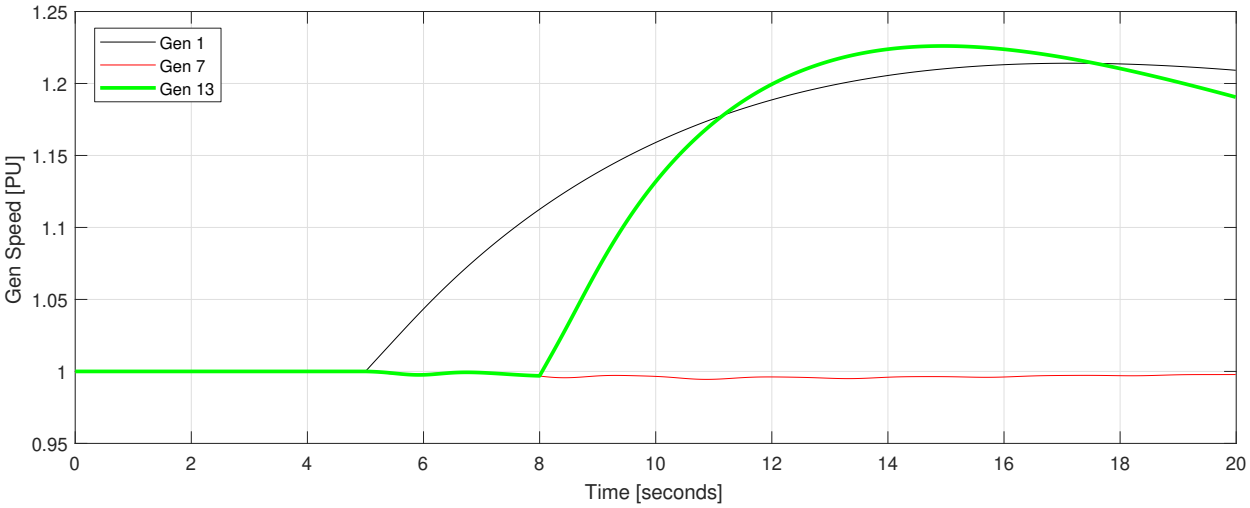


Figure 4.23: Select generator speed during PST 3.1 run_mwGenTrip.

PST 4 zeros derivatives of tripped machine speeds so that the machine will stay at whatever speed it was tripped at. This zeroing of derivatives is helpful when using VTS as time step size is determined by all derivatives in a system.

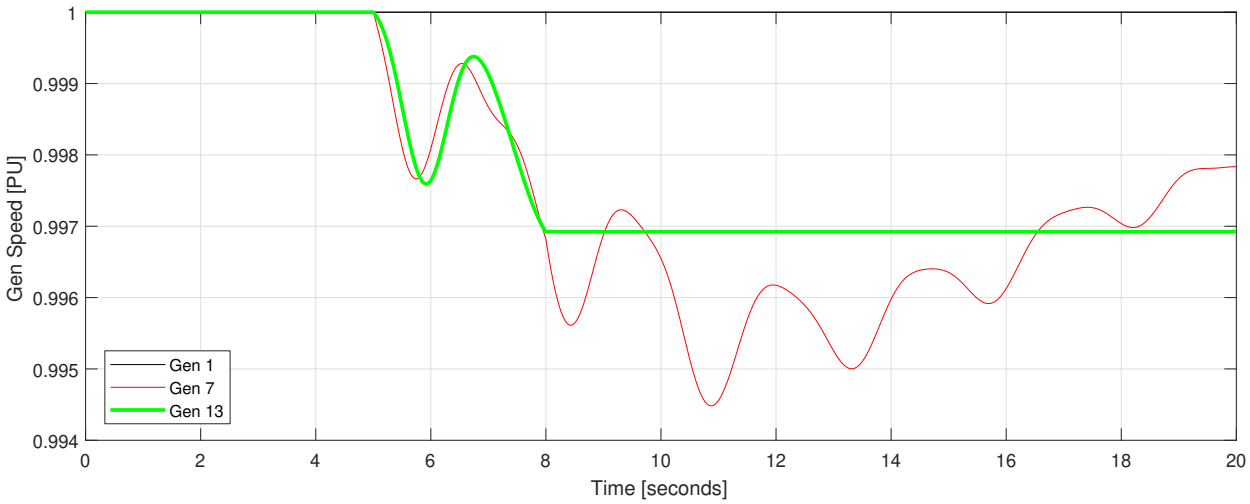


Figure 4.24: Select generator speed during PST 4.0 run_mwGenTrip.

PST 4 calculates system total inertia and accounts for tripped generators.

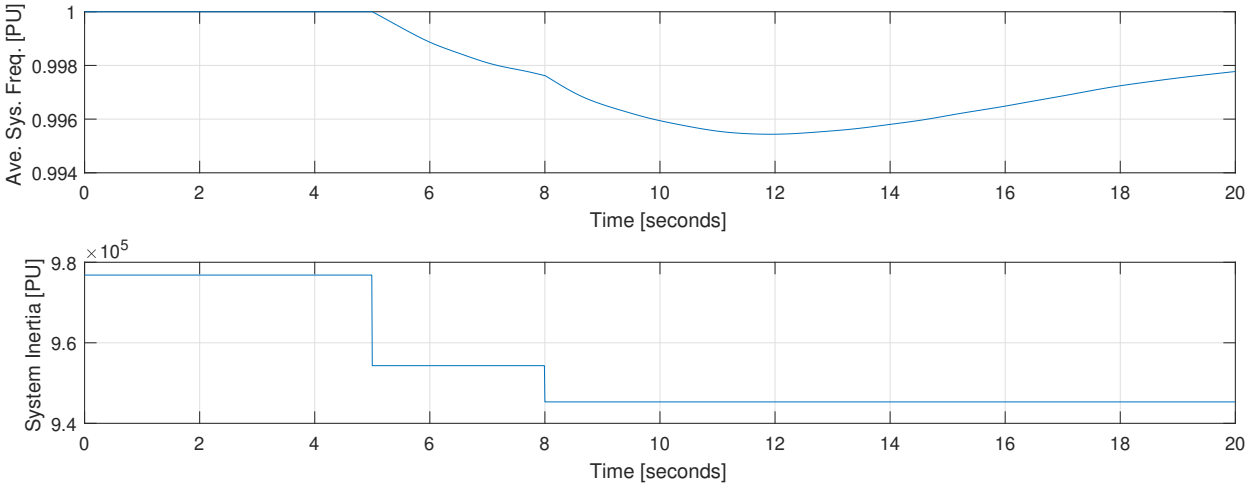


Figure 4.25: Calculated system inertia during PST 4.0 run_mwGenTrip.

4.7.2 miniWECC-AGC - WIP

Ten minute multi-area AGC recovery using optional VTS in PSTv4. Two different area configurations: One is the same as in [5], and one made to mimic EIA regions (see Appendix B). Example allows for comparing VTS to FTS simulation methods. DC lines are modeled as power injection via load settings. This power import/export is **not** included in AGC calculations.

Table 4.4: PST Version Comparisons of MiniWECC AGC VTS Example.

	Step Size [seconds]				Solutions Per Step				
PST Version	Max.	Min.	Ave.	Total Steps	Ave.	Max.	Total Slns.	Sim. Time	Speed Up
4.0 - FTS	8.33E-3	8.33E-3	8.33E-3	72,001	2	2	144,002	782.18	1.00
4.0 - VTS	3.32	4.07E-06	1.18E-01	5,094	3	787	15,332	151.54	5.16

VTS is ≈5 times faster than FTS and saves ≈13x less data.

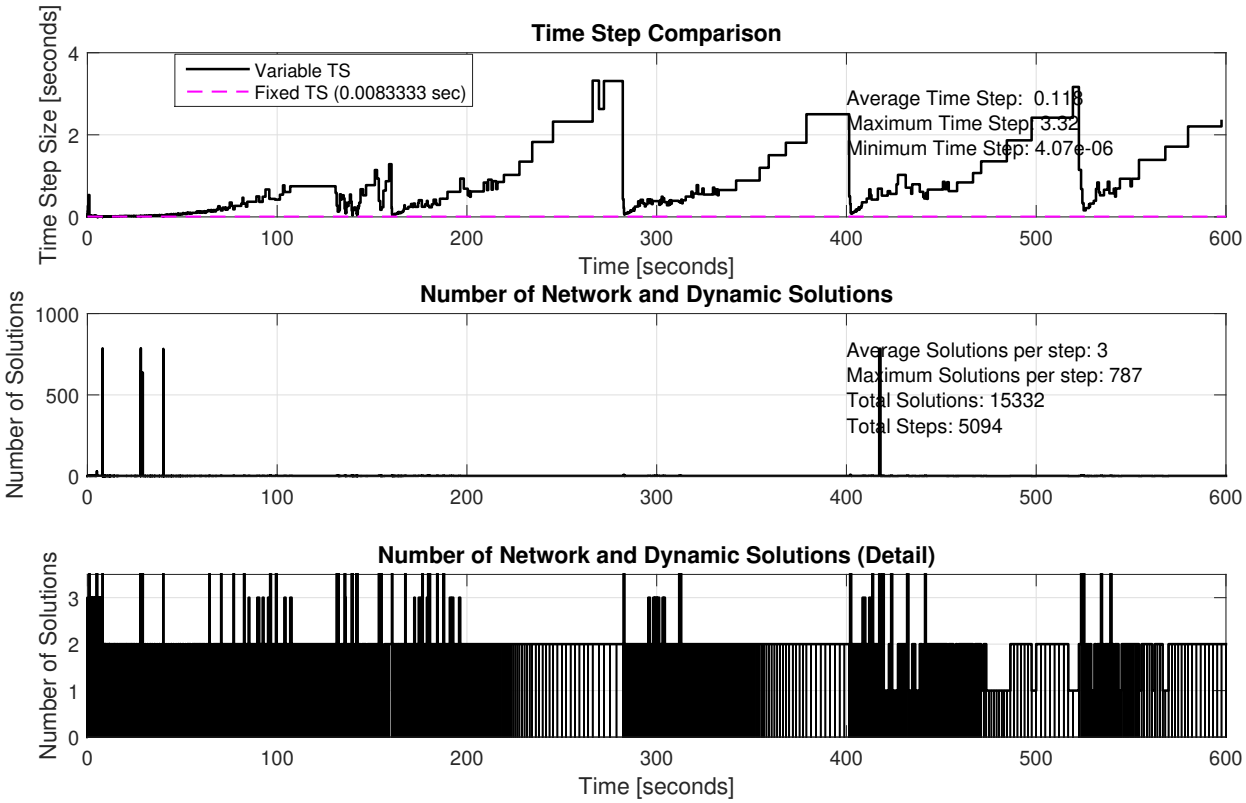


Figure 4.26: MiniWECC AGC recovery FTS vs VTS time step size and solution count.

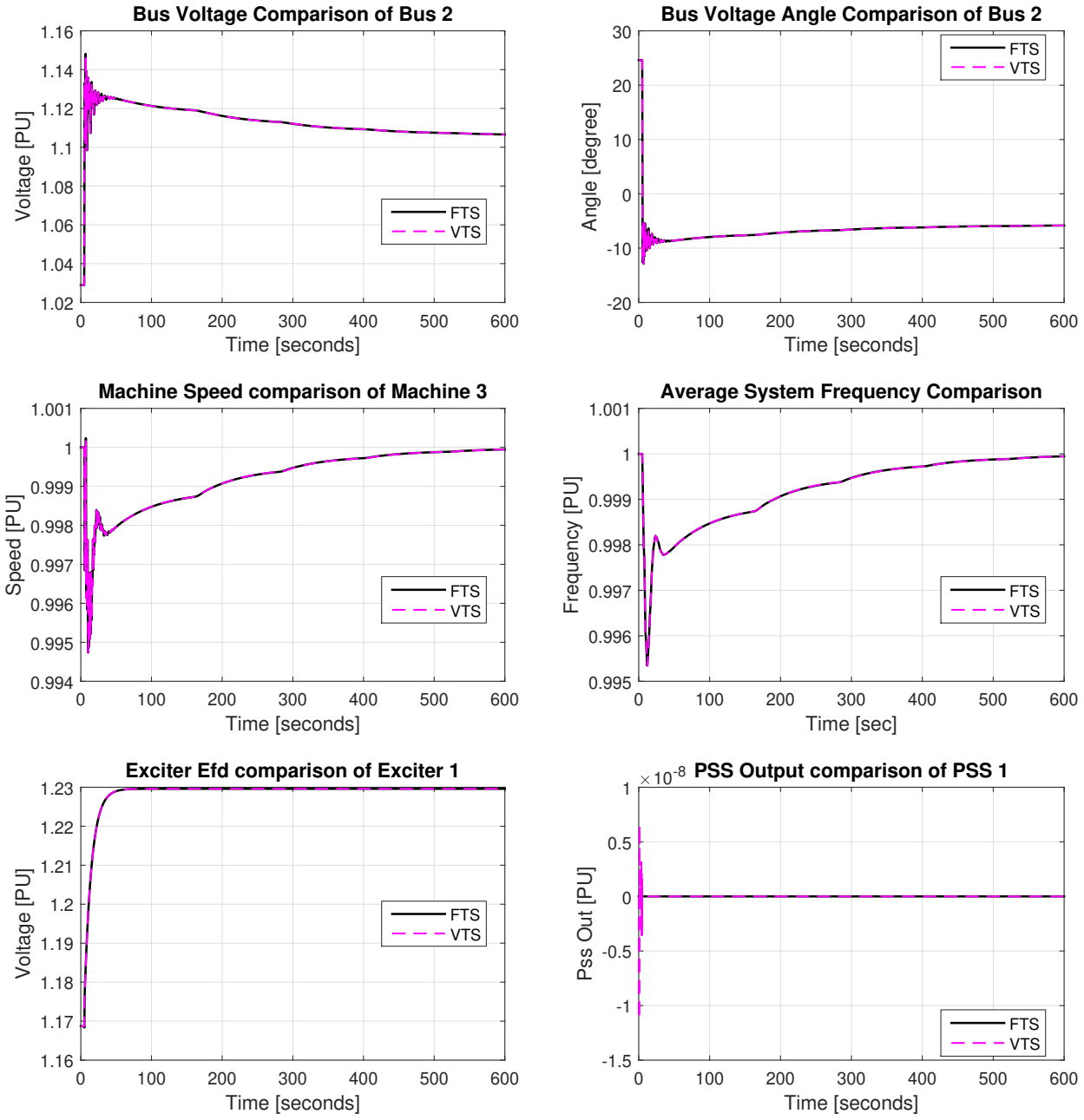


Figure 4.27: MiniWECC AGC recovery FTS vs VTS select comparisons.

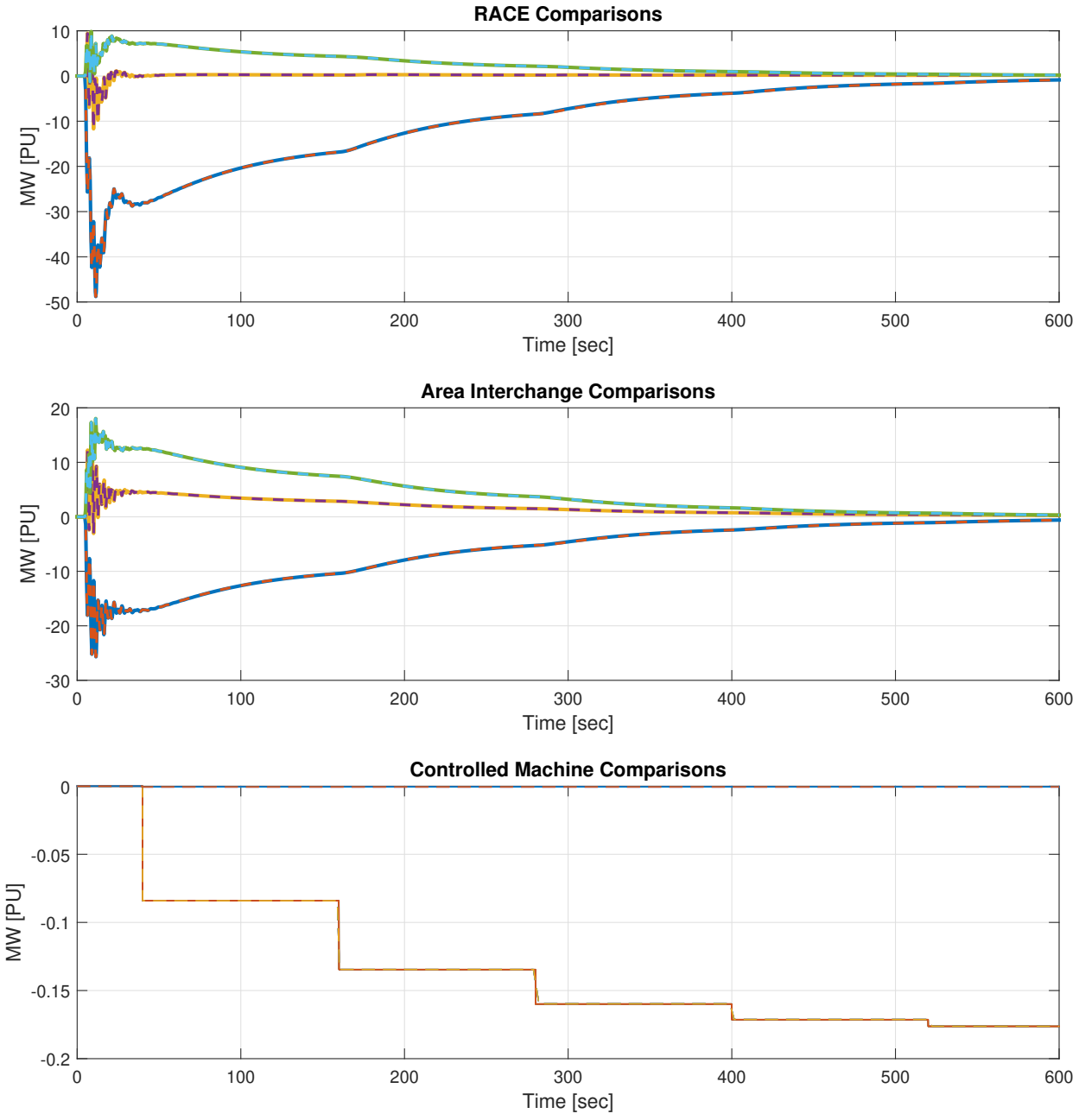


Figure 4.28: MiniWECC AGC recovery FTS vs VTS AGC values.

4.8 pwrmod -WIP

Adapted from Dr. Trudnowski example. Power injection example contain VTS run script. Linear comparison for both examples.

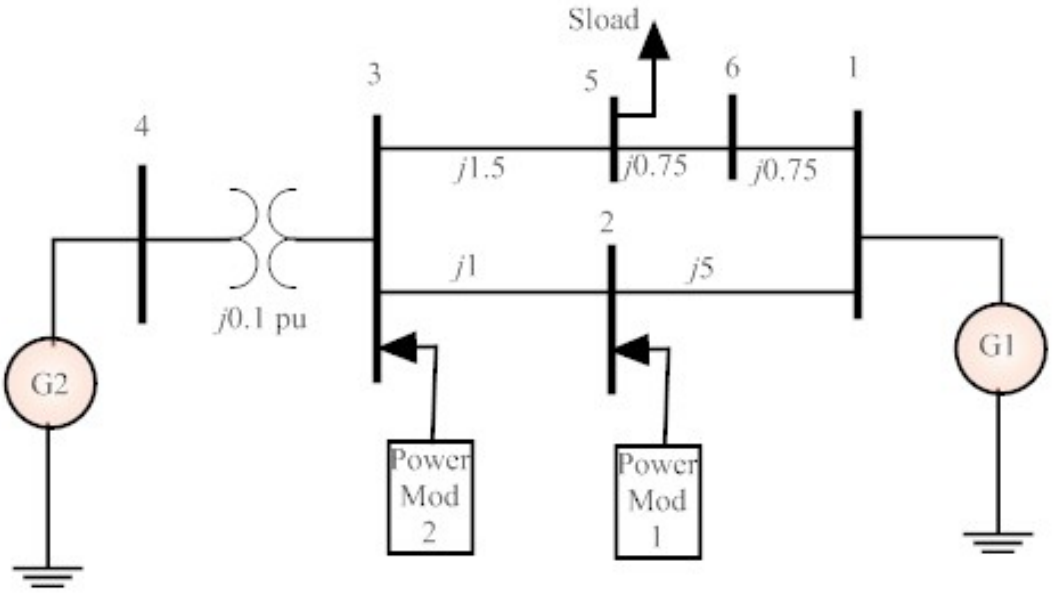


Figure 4.29: System one-line diagram for pwrmod examples.

4.8.1 P-injection - WIP

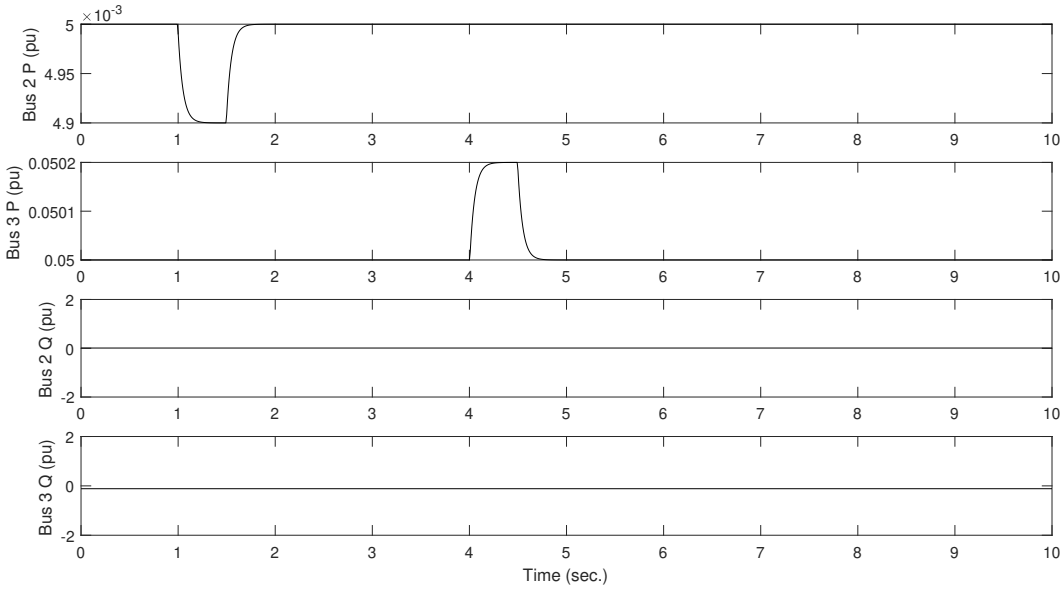


Figure 4.30: Power injection from example case.

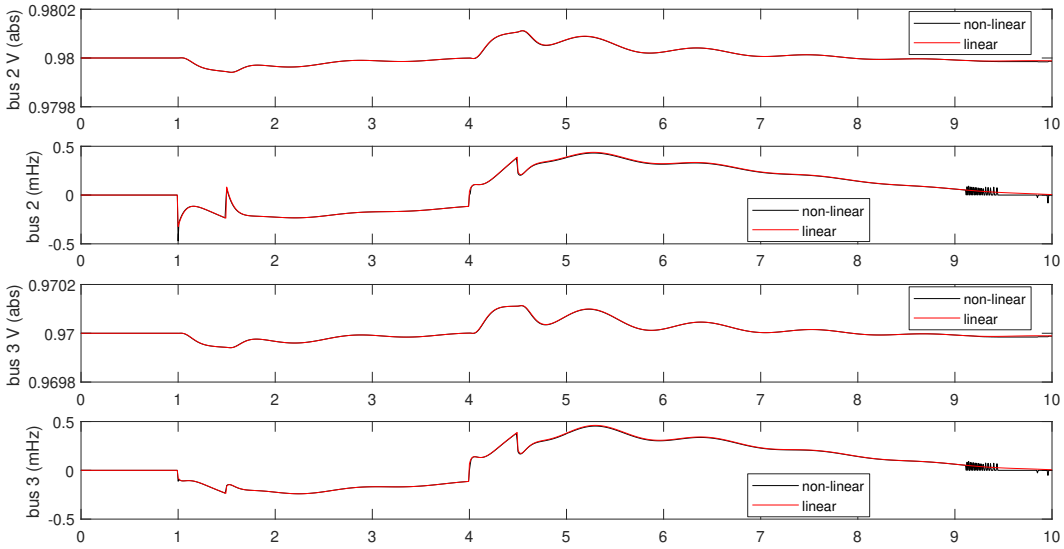


Figure 4.31: Linear comparison of power injection example case.

4.8.2 I-injection - WIP

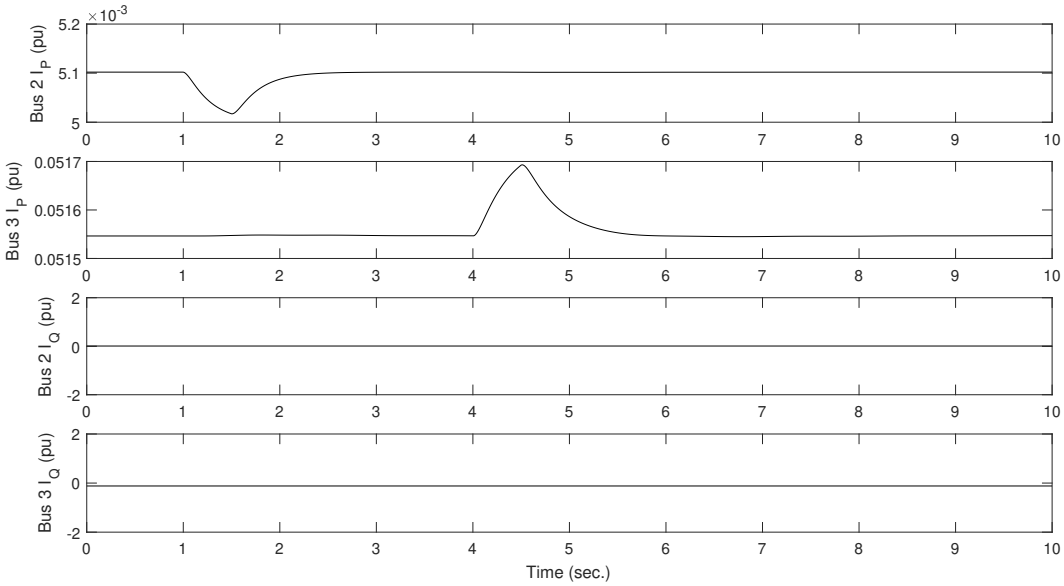


Figure 4.32: Current injection from example case.

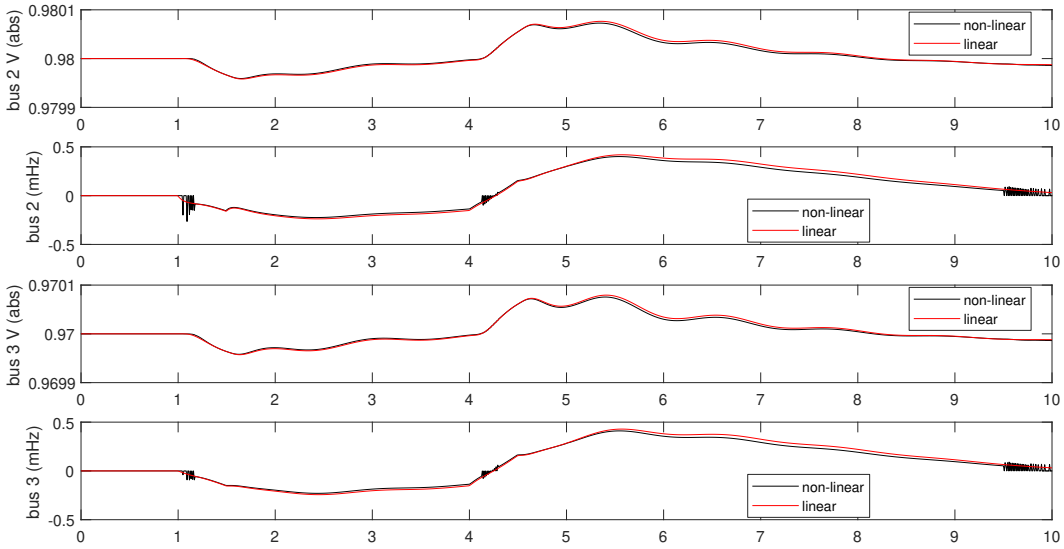


Figure 4.33: Linear comparison of current injection example case.

4.9 untrip - WIP

Extended term simulations may require the addition of more generation sources. To experiment with bringing generation online, a simple 3 machine system was created. All machines were modeled with governors, exciters, and PSS. Most model parameters are the same, with the exception of MVA base. Generators 1, 2, and 3, have an M_{base} of 500, 200, and 100 MVA respectively. The experimental goal was to trip Generator 3 off-line, and then 'nicely' re-connect it.

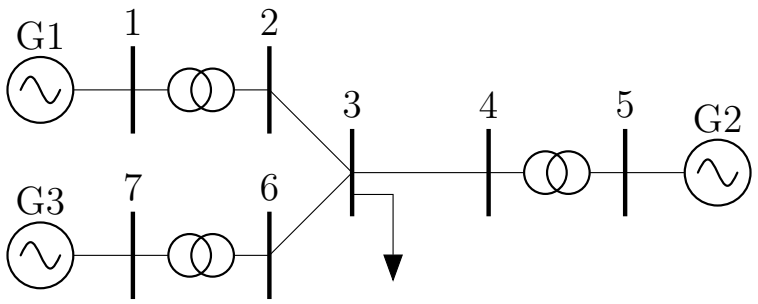


Figure 4.34: System used in untrip example.

4.9.1 Test Event Time Line - WIP

- $t = 0$ - System initialized
- $t = 5$ - Generator 3 trips off. Associated derivatives, P_{mech}, and governor P_{ref} set to zero.
- $t = 15$ - Generator 3 re-synced to system and infinite reactance reset to original value.
- $t = 20 - 25$ - The governor attached to Generator 3 is reinitialized and the ω_{ref} value is ramped to its original value. This causes mechanical power to be generated by Generator 3 which causes minor transients in system machine speed.
- $t = 35$ - The exciter and PSS on generator 3 is re-initialized and the exciter bypass is removed.
- $t = 45 - 65$ - Ramping the governor P_{ref} to the original value increases system speed and real power flow from Generator 3 while the ramping of exciter reference voltage to its original value decreases system speed and increases reactive power flow from generator 3.
- $t = 150$ - Simulation End

4.9.2 Observations of Note - WIP

- Nicely 'un-tripping' a generator appears possible.
- System seems to return to original state.
- Scenario development used FTS as VTS will likely present additional re-initialization issues.

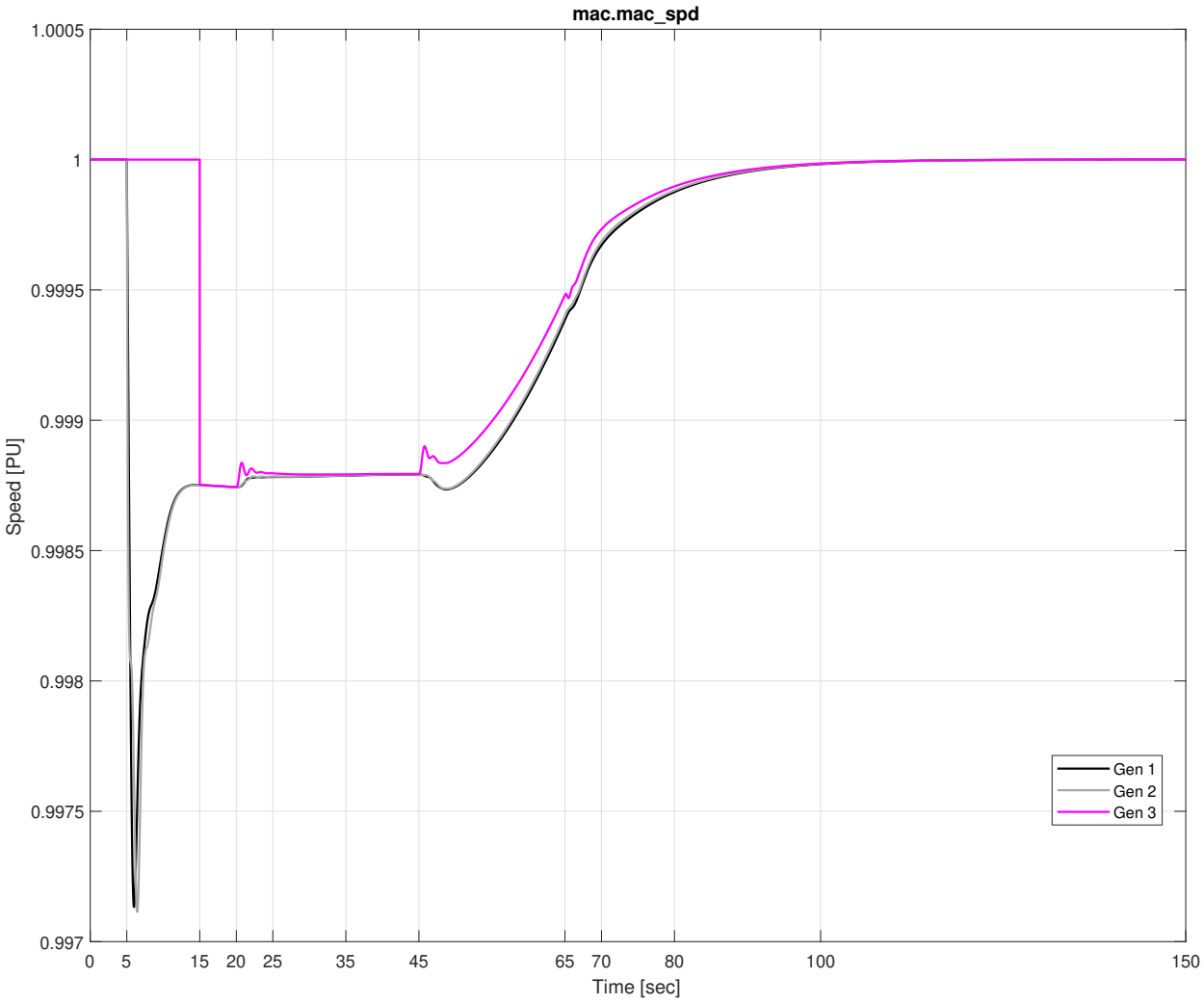


Figure 4.35: Generator speeds from untrip example.

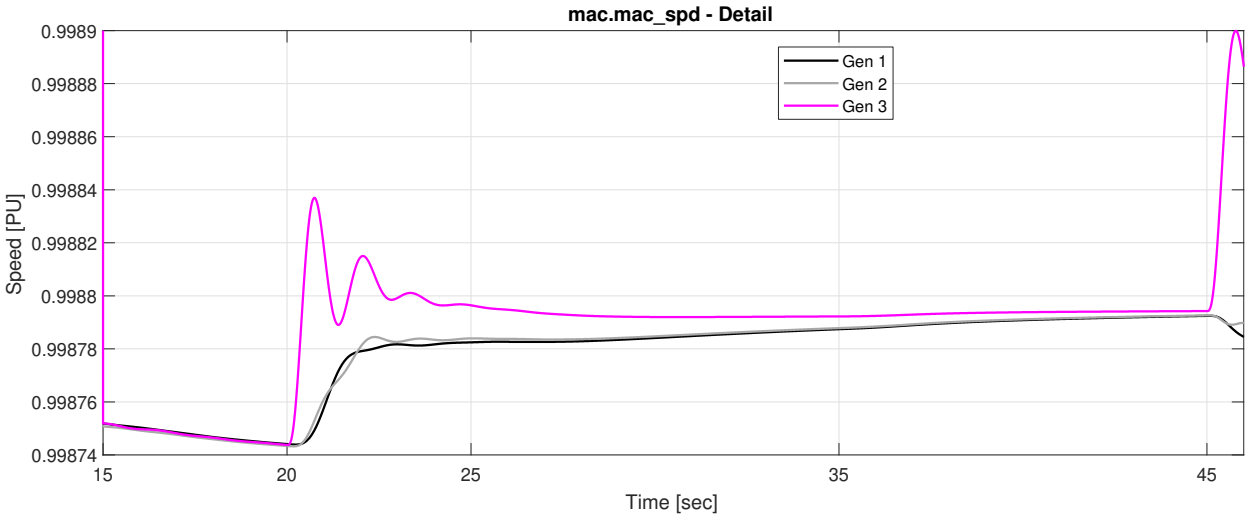


Figure 4.36: Detail generator speed during governor re-initialization.

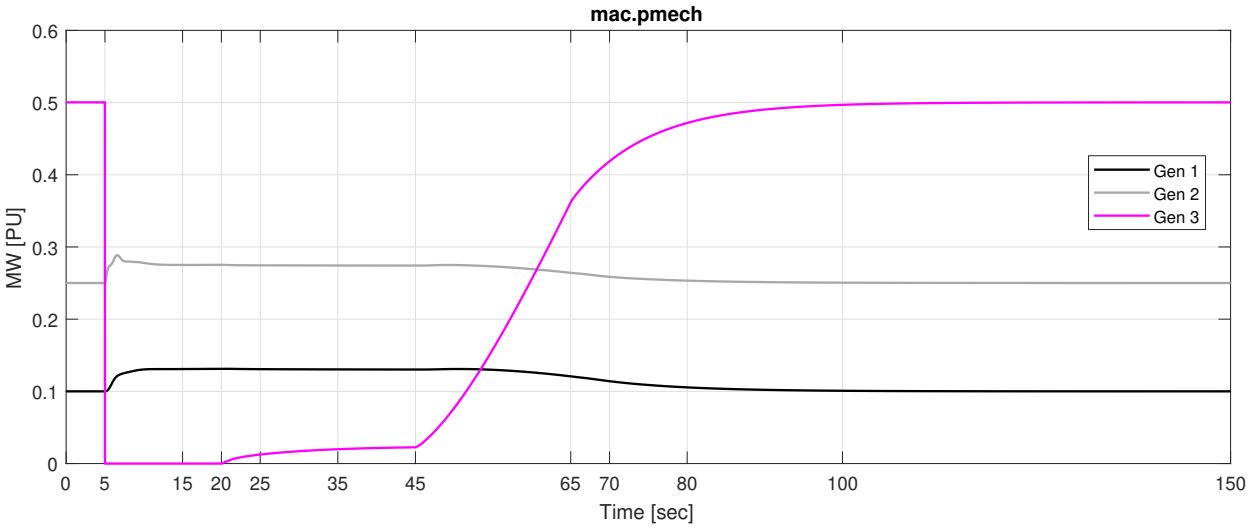


Figure 4.37: Mechanical power during untrip example.

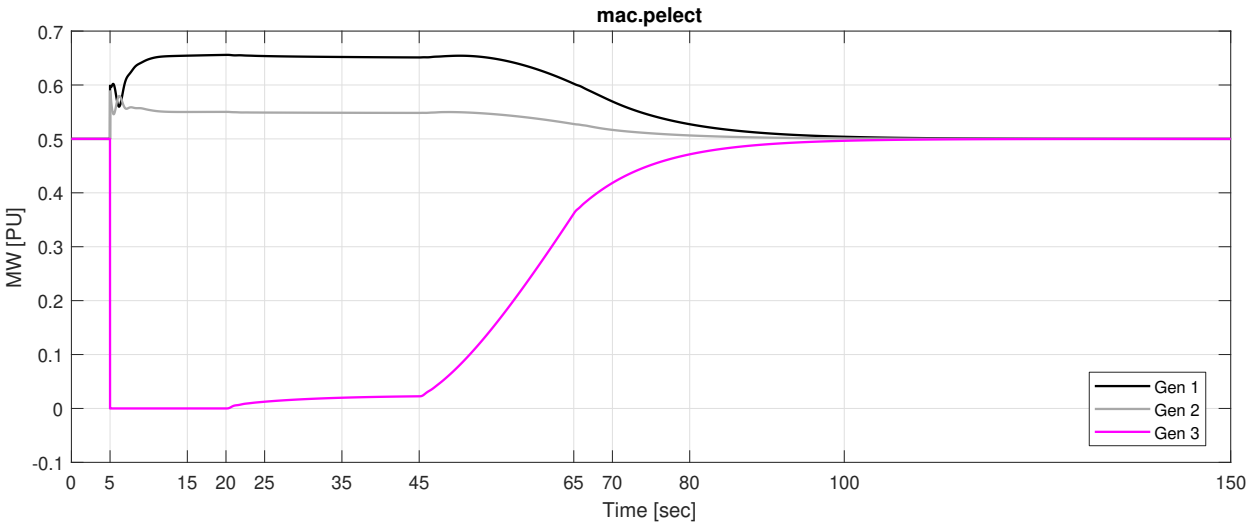


Figure 4.38: Real electric power generated during untrip example.

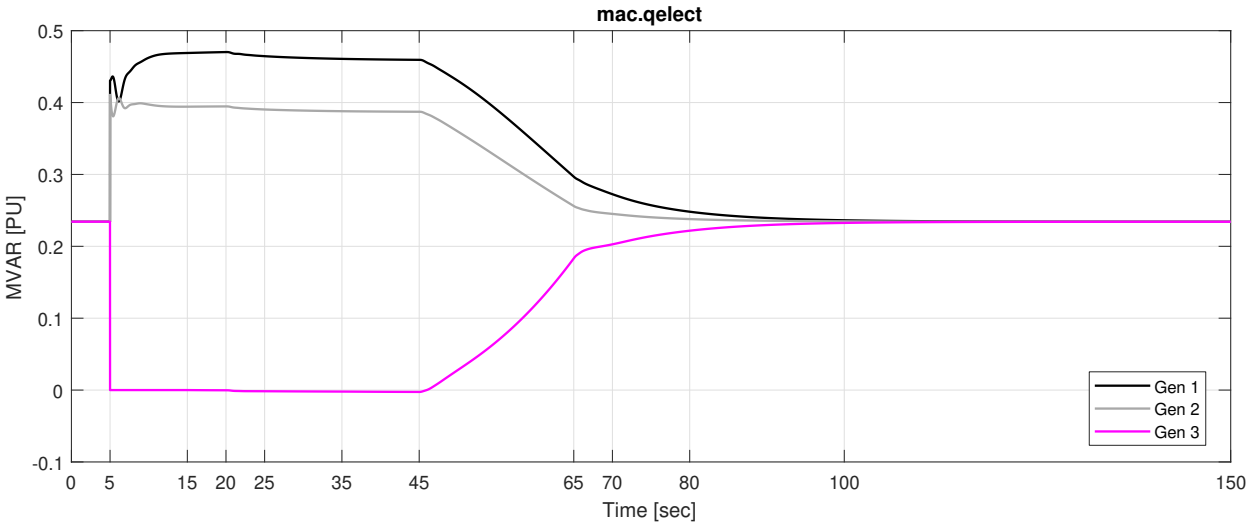


Figure 4.39: Reactive power generated during untrip example.

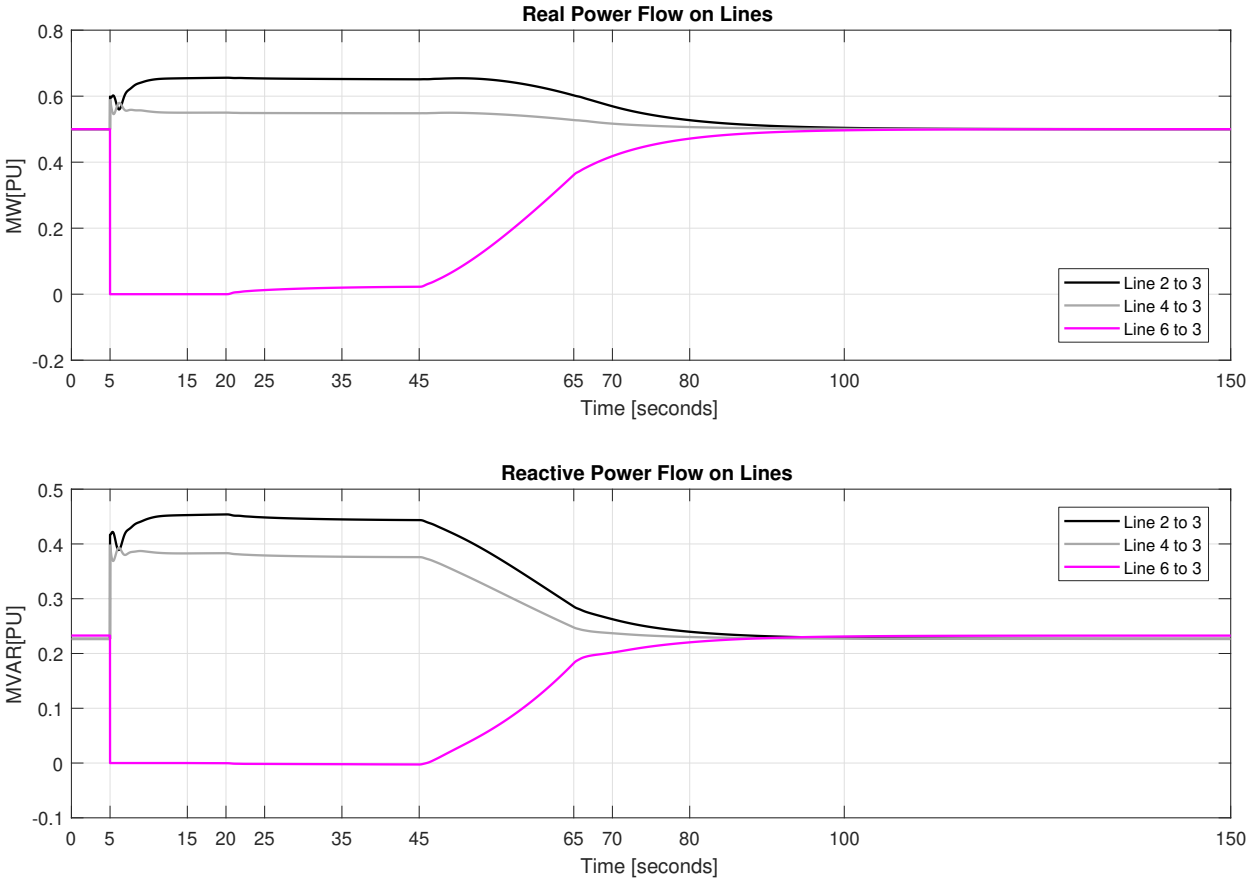


Figure 4.40: Line power flow from generators during untrip example.

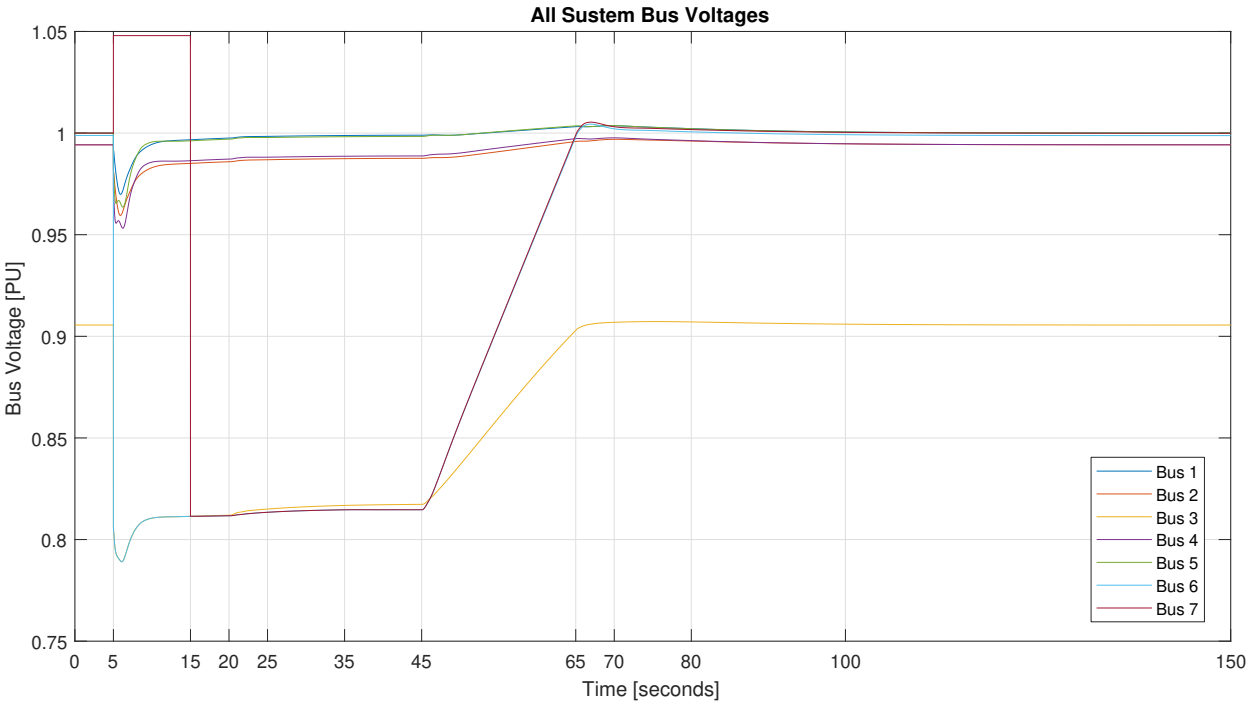


Figure 4.41: System bus voltages during untrip example.

4.9.3 Machine Trip Logic Code - WIP

Most 'un-trip' action takes place in the `mac_trip_logic` file. Such actions include:

- Trip generator 3
- Set mechanical power to zero and bypass governor
- Bypass exciter
- Un-trip generator 3
- Re-initialize machine
- Re-init governor
- Ramp governor ω_{ref}
- Re-init and remove bypass on exciter and PSS
- Ramp exciter reference

It should be noted that the `mac_trip_logic` routine usage was created 'pre-global g', and as a result, passes variables in and out that are essentially globals. Realistically, only a data index would need to be passed into the function, and any action can take place directly on the associated `g.mac.mac_trip_states` vector or other required global.

Variables to note in associated examples (where `x` is the internal model number):

- exciter V_{ref} = `g.exc.exc_pot(x,3)`
- governor P_{ref} = `g.tg.tg_pot(x,5)`
- governor ω_{ref} = `g.tg.tg_con(x,3)`

Listing 4.2: Machine Trip Logic for Untripping a Generator

```
function [tripOut,mac_trip_states] = mac_trip_logic(tripStatus,mac_trip_states,t,kT)
% Purpose: trip generators.
%
% Inputs:
%   tripStatus = n_mac x 1 bool vector of current trip status.  If
%       tripStatus(n) is true, then the generator corresponding to the nth
%       row of mac_con is already tripped.  Else, it is false.
%   mac_trip_states = storage matrix defined by user.
%   t = vector of simulation time (sec.).
%   kT = current integer time (sample).  Corresponds to t(kT)
%
% Output:
%   tripOut = n_mac x 1 bool vector of desired trips.  If
%       tripOut(n)==1, then the generator corresponding to the nth
%       row of mac_con is will be tripped.  Note that each element of
%       tripOut must be either 0 or 1.

% Version 1.0
% Author:   Dan Trudnowski
% Date:   Jan 2017

% 08/28/20  12:35   Thad Haines     Trip a generator, then bring it back online
% exciter and governor ramp at same time

%% define global variables
global g

persistent excVrefNEW excVrefOLD % variables for exciter ramping
persistent wRef0 wRef1 wDelta r0 % variables for governor ramping

if kT<2
    tripOut = false(g.mac.n_mac,1);
    mac_trip_states = [0 0;0 0]; % to store two generators trip data...
else
    tripOut = tripStatus;

    %% Trip generator
    if abs(t(kT)-5)<1e-5
        tripOut(3) = true; %trip gen 1 at t=5 sec.
        mac_trip_states(3,:) = [3; t(kT)]; %keep track of when things trip
        disp(['MAC_TRIP_LOGIC:  Tripping gen 3 at t = ' num2str(t(kT))])
        for n=0:1
            g.mac.pmech(3,kT+n) = 0; % set pmech to zero
```

```
44          end
45
46          % bypass governor
47          g.tg.tg_pot(3,5) = 0.0;     % set Pref to zero
48          r0 = g.tg.tg_con(3,4);      % store orginal 1/R
49          g.tg.tg_con(3,4) = 0.0;     % set 1/R = 0
50          reInitGov(3,kT)             % reset governor states
51      end
52
53      %% untrip gen
54      if abs(t(kT)-15.0)<1e-5 %
55          disp(['MAC_TRIP_LOGIC:  "Un-Tripping" gen 3 at t = ' num2str(t(kT))])
56          tripOut(3) = false;
57          mac_trip_states(3,:) = [3; t(kT)];  % keep track of when things trip
58          g.mac.mac_trip_flags(3) = 0;        % set global flag to zero.
59
60          % bypass exciter (and pss)
61          g.exc.exc_bypass(3) = 1;            % set bypass flag
62          excVrefOLD = g.exc.exc_pot(3,3);          % save initial voltage reference
63          reInitSub(3,kT)                     % init machine states and voltage to
            ↪  connected bus at index kT
64      end
65
66      %% ramp wref to wref0
67      if abs(g.sys.t(kT)-20) < 1e-5
68          disp(['MAC_TRIP_LOGIC:  reinit gov, start ramping wref at t = ',
            ↪  num2str(t(kT))])
69          g.tg.tg_con(3,4) = r0;       % restore original 1/R value
70          reInitGov(3,kT)              % re-init gov states
71          wRef0 = g.tg.tg_con(3,3);    % wref0
72          wRef1 = g.mac.mac_spd(3,kT); % current machine speed
73          g.tg.tg_con(3,3) = wRef1;    % set reference to current speed
74          wDelta = wRef0 - wRef1;      % amount to ramp in
75      end
76
77      if g.sys.t(kT)>= 20 && g.sys.t(kT)< 35      % ramp w ref to original value
78          g.tg.tg_con(3,3) = wRef1+ wDelta*(1 - exp( 20-g.sys.t(kT) ) ); % concave down
79      end
80
81      if abs(t(kT)-35.0)<1e-5 % Reset governor w ref
82          g.tg.tg_con(3,3) = wRef0;
83          disp(['MAC_TRIP_LOGIC:  wref ramp in complete at t = ', num2str(t(kT)) ])
84      end
85
```

```
    %% Re-connect exciter
    if abs(t(kT)-35.0)<1e-5     % remove bypass on exciter
        disp(['MAC_TRIP_LOGIC:  connecting exciter at t = ', num2str(t(kT))])
        reInitSmpExc(3,kT)      % re-init single exciter
        pss(3,kT,0)             % re-init pss
        g.exc.exc_bypass(3) = 0;% remove exciter bypass
    end

    %% Ramp exciter
    if abs(t(kT)-45.0)<1e-5 % ramp exciter reference voltage
        disp(['MAC_TRIP_LOGIC:  ramping exciter to original Vref at t = ',
        ↪  num2str(t(kT))])
        excVrefNEW = excVrefOLD - g.exc.exc_pot(3,3); % calculate difference to make up
        excVrefOLD = g.exc.exc_pot(3,3);
    end
    if t(kT)>=45 && t(kT) <65
        g.exc.exc_pot(3,3) =  excVrefOLD + (t(kT)-45)*excVrefNEW/20;
    end

end
end
```

4.9.4 Turbine Governor Modulation Code - WIP

The `mtg_sig` file was used to ramp the governors P_{ref} back to the original value.

Listing 4.3: Mechanical Power Modulation Signal Code for Untripping a Generator

```
function mtg_sig(k)
% MTG_SIG Defines modulation signal for turbine power reference
% Syntax: mtg_sig(k)
%
global g
% actions to return a generator back on line
% ramp pref instead of tg sig

%% ramp Pref near to original value
if abs(g.sys.t(k)-45) < 1e-6
    disp(['MTG_SIG:  ramping gov Pref at t = ', num2str(g.sys.t(k))])
end
if g.sys.t(k)>= 45 && g.sys.t(k)< 65 %
    g.tg.tg_pot(3,5) = (g.sys.t(k)-45)*(0.5003)/20; % ramp reference
end

%% set signal near to pref,
if abs(g.sys.t(k)-65) < 1e-6
    disp(['MTG_SIG:  Pref ramp done, setting Pref at t = ', num2str(g.sys.t(k))])
    g.tg.tg_pot(3,5) = (0.5003);
end

end% end function
```

4.10 Lightly Introduced Examples - WIP

These examples exist on github, but don't necessarily require a much more than a casual mention.

4.10.1 DC - WIP

Non-linear simulation seems to work in all versions. Linear analysis doesn't seem to work correctly - possibly due to user error

4.10.2 exciterBatchTests - WIP

PST 4 only script. Used to compare all 4 exciter models linear/non-linear response to load step. Was useful in ensuring exciter models were cast to globals correctly.

4.10.3 inductive - WIP

Meant to verify functionality of inductive generators and loads (motors) using a global g. Multiple cases run from example script: fault example and load pulse with linear/non-linear comparison. Works in all PST versions.

4.10.4 tg - WIP

Examples of a load step with and without governor action. Meant to clearly show how governors act to arrest frequency change.

5 Loose Ends

As software development is never actually 'done', this chapter is meant to contain any loose ends that felt relevant.

1. As infinite buses don't seem to be used in dynamic simulation, they were not converted to use the golbal g.
2. `tgh` model was not converted for use with global g. (no examples of tgh gov)
3. In original (and current) `s_simu`, the global `tap` value associated with HVDC is overwritten with a value used to compute line current multiple times.
   Ryan Elliot has reported that this `tap` variable is locally re-created before every use. Thus, it has no real need to be global.
4. Constant Power or Current loads seemed to require a portion of constant Impedance - possibly user error.
5. PSS design functionality not explored.
6. No examples of the delta P omega filter or user defined damping controls for SVC and TCSC models appear to exist.
7. Differences in `mac_ind` between PST 2 and 3 seem backward compatible, but this are untested.
8. DC is not implemented in VTS. If desired, DC models should be combined into the main routine for VTS as it seems counter intuitive/ confusing / inefficient to do *multi-rate variable time step integration.*
9. AGC capacity should consider defined machine limits instead of assuming 1 PU max.
10. AGC should allow for a 'center of inertia' frequency option instead of the weighted average frequency.
11. A method to initialize the power system with tripped generators should be devised and occur before the first power flow solution.
12. A method to zero derivatives of any model attached to a tripped generator should be created to enable VTS to optimize time steps.
13. Re-initializing a tripped generator in VTS will likely require indexing the `g.vts.stVec`. This could be aided by adding indices to the `g.vts.fsdn` cell.
14. miniWECC DC lines (modeled as power injection) are not included in AGC calculations as the power does not travel over any simulated lines.
15. If a machine has been tripped, the Y matrix is adjusted and reduced every time step. This repeated action could be made more efficient.
16. Odd IVM behavior may have to do with how PST doesn't really use the synchronous reference frame (`g.sys.syn_ref`) or assumes `g.sys.sys_freq` is always 1.
17. Variables passed into the generator trip function may be cleaned up as they are globals.

6 Bibliography

[1] J. H. Chow. (2015). Power system toolbox, Rensselaer Polytechnic Institute, [Online]. Available: `http://web.eecs.utk.edu/~dcostine/ECE620/Fall2015/lectures/PST-tutorial.pdf`.

[2] J. H. Chow and K. W. Cheung, "A Toolbox for Power System Dynamics and Control Engineering Education and Research," Rensselaer Polytechnic Institute, DOI: 10.1109/59.207380, 1992.

[3] J. Chow and G. Rogers, *Power System Toolbox Version 2.0 Load Flow Tutorial and Functions*, 1999.

[4] J. Chow and G. Rogers, *Power System Toolbox Version 3.0 User Manual*, 2008.

[5] T. Haines, "Long-term dynamic simulation of power systems using python, agent based modeling, and time-sequenced power flows," Master's thesis, Montana Technological University, 2020.

[6] I. Hiskens, "IEEE PES Task Force on Benchmark Systems for Stability Controls," 2013.

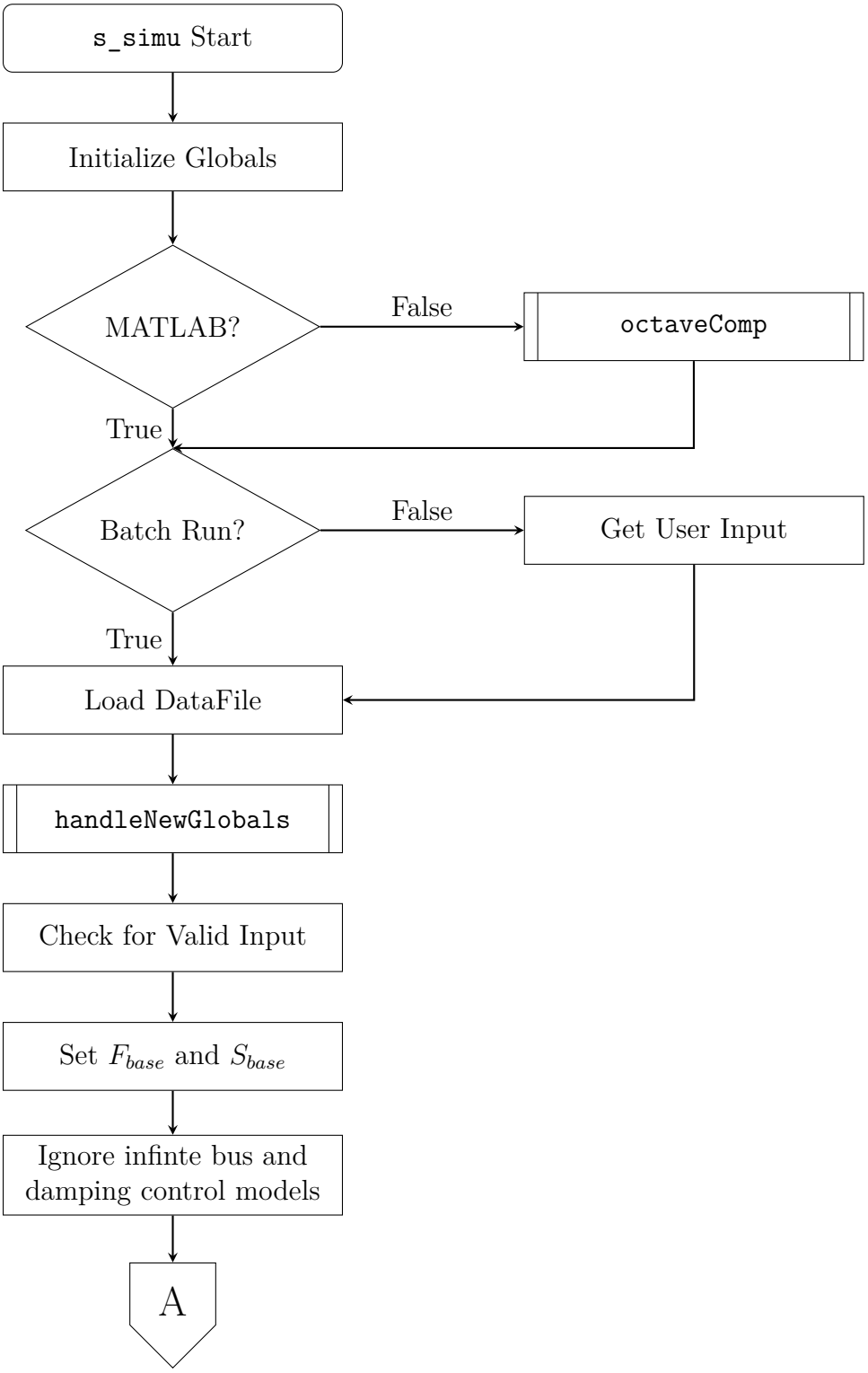
[7] G. Rogers, *Power System Oscillations*. Springer, 1999.

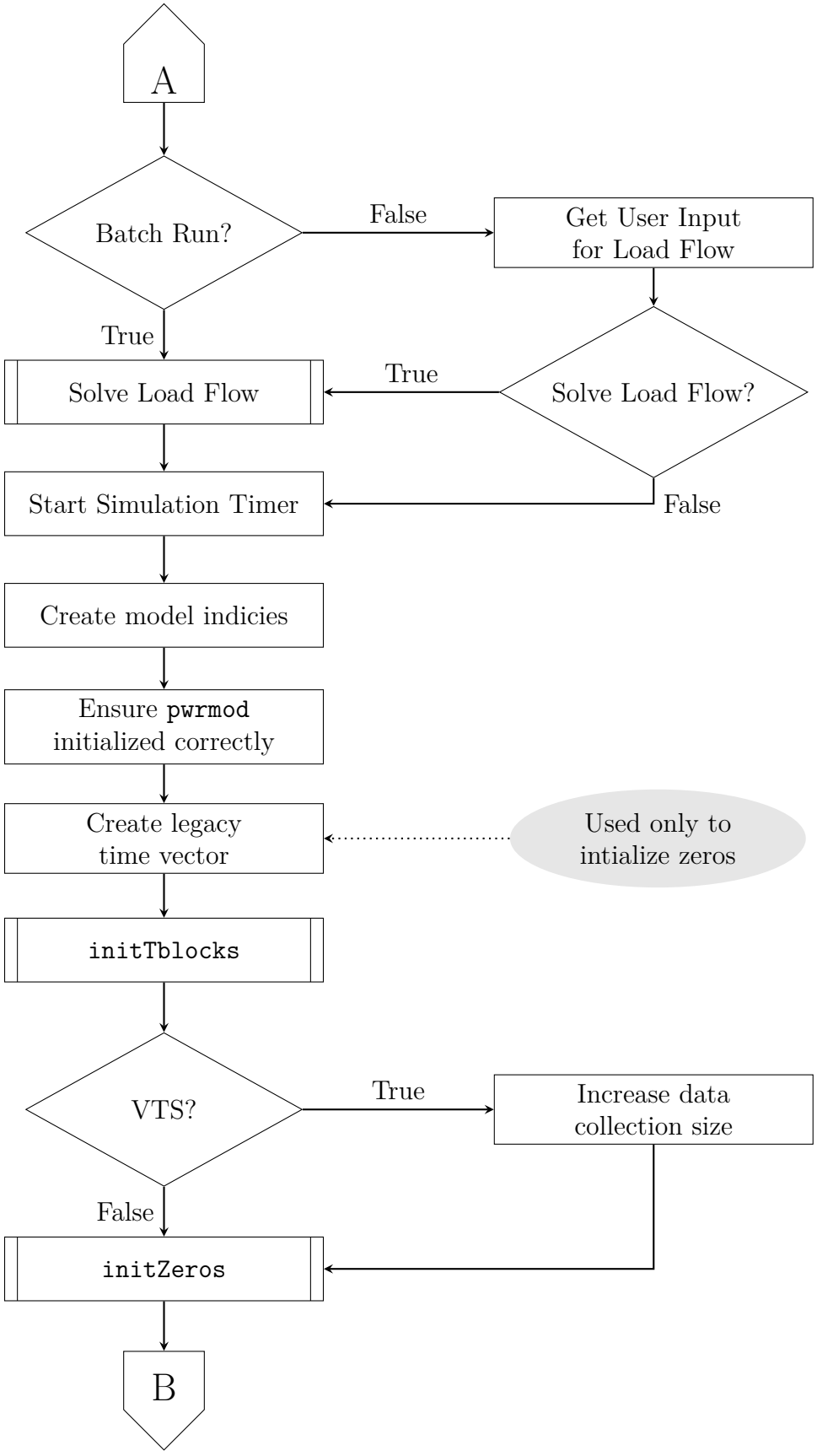
[8] P. W. Sauer, M. A. Pai, and J. H. Chow, *Power System Dynamics and Stability: With Synchrophasor Measurement and Power System Toolbox*. Wiley-IEEE Press, 2017.

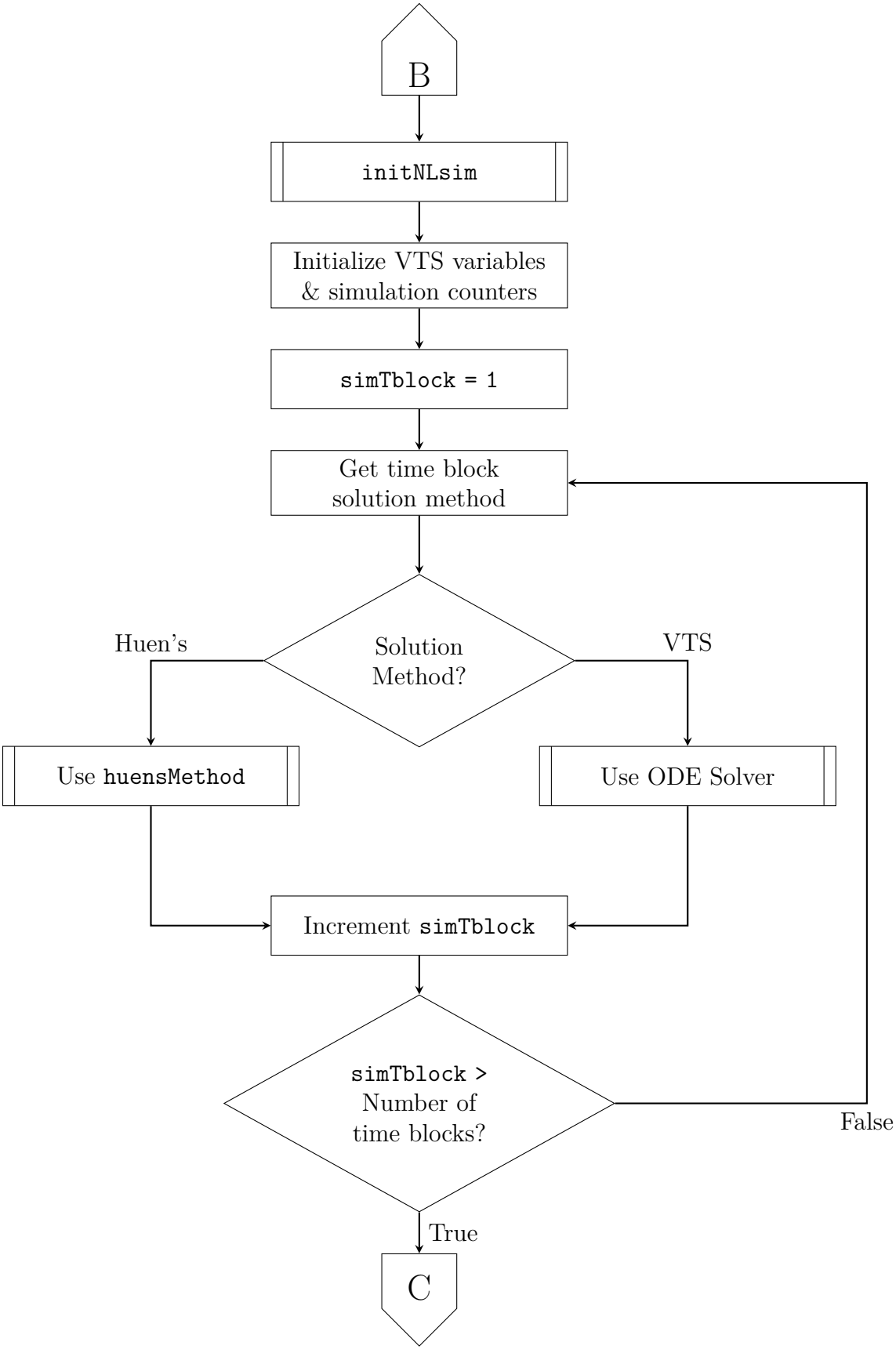
7 Document History

- 09/02/20 - 0.0.0-a.1 - Document initialization.
- 09/03/20 - 0.0.0-a.2 - Population with sections and some previously generated content, addition of glossary
- 09/07/20 - 0.0.0-a.3 - Added and alphabetized more sections, collected raw material from research documents
- 09/08/20 - 0.0.0-a.4 - Figure and equation formatting in AGC, more logical sectioning, minor editing
- 09/09/20 - 0.0.0-a.5 - Introduction work, AGC section editing, restructure of various sections, readability edits to a majority of sections, WIP added to sections requiring more work.
- 09/10/20 - 0.0.0-a.6 - Split of introductions, global variables, lmon, liveplot, and VTS sections draft complete
- 09/12/20 - 0.0.0-a.7 - Addition of `huensMethod`, clean up of glossary, additional bibliography and loose ends entries.
- 09/13/20 - 0.0.0-a.8 - Edits to huensMethod and addition of block diagram
- 09/14/20 - 0.0.0-a.9 - Creation of `s_simu` block diagram in appendix, added content to `mac_trip_logic` section
- 09/15/20 - 0.0.0-a.10 - Addition of `svm_mgen_Batch` section, work on pwrmod section, hiskens example and citation, Addition of Trudnowski PST intro
- 09/16/20 - 0.0.0-b.1 - Addition of example case sections, expanded bibliography, other edits
- 09/17/20 - 0.0.0-b.2 - Update of code format in global variable section, addition of some 'appropriate' page breaks, corrected naming of flow charts
- 09/21/20 - 0.0.0-b.3 - Edits after 1st read through, page breaks added, addition of listing classification to code examples, sloppy copy of extended and agc ic mod example
- 09/28/20 - 0.0.0-b.4 - Glossary clean up, minor editing of lists, correction of PSS listing reference, update of pst version history, work on IVM example and description, clean up of sloppy copy: AGC, extendedTerm, and untrip examples.
- 09/30/20 - 0.0.0-rc.1 - Removed acknowledgments, added miniWECC, pwr, and ivm example plots.
- 10/01/20 - 1.0.0-dub - Final edits and formatting... Part 1.
- 10/13/20 - 1.0.0 - Final edits and formatting... Part 2.

A PST 4 s_simu Flow Chart




A
Batch Run?
False
Get User Input for Load Flow
True
Solve Load Flow
True
Solve Load Flow?
Start Simulation Timer
False
Create model indicies
Ensure pwrmod initialized correctly
Create legacy time vector
Used only to intialize zeros
initTblocks
VTS?
True
Increase data collection size
False
initZeros
B


B
initNLsim
Initialize VTS variables & simulation counters
simTblock = 1
Get time block solution method
Solution Method?
Huen's
VTS
Use huensMethod
Use ODE Solver
Increment simTblock
simTblock > Number of time blocks?
False
True
C

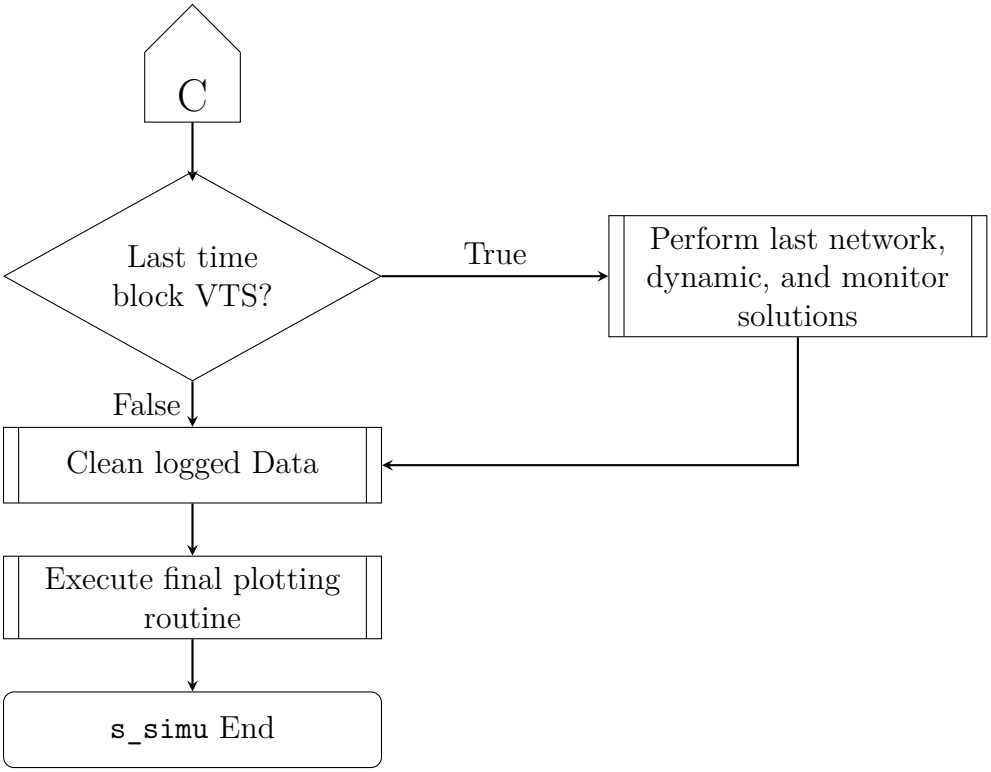


Figure A.1: PST 4 s_simu Flow Chart.

B MiniWECC One-Line Diagrams

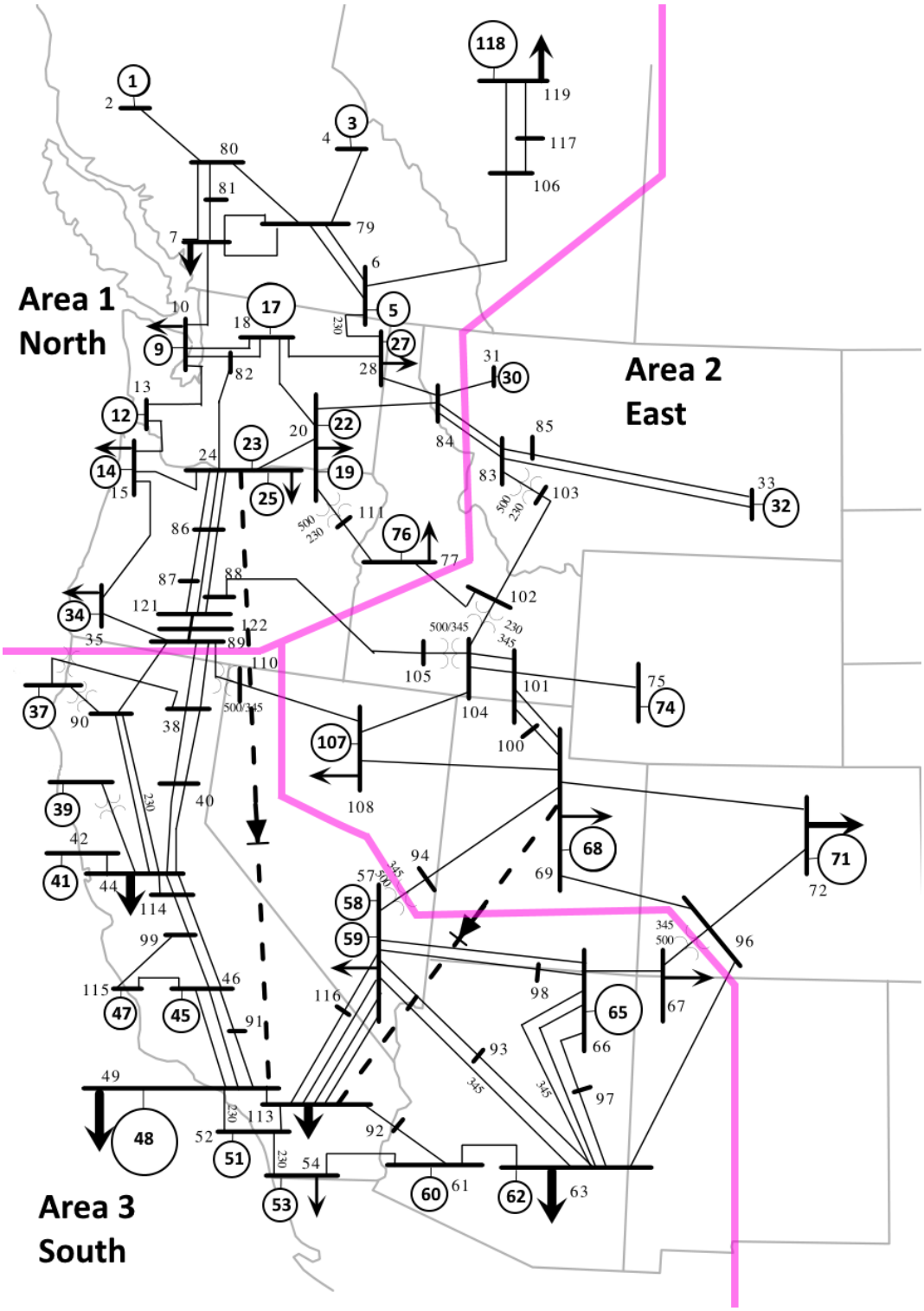


Figure B.1: MiniWECC using PSLTDSim Areas.

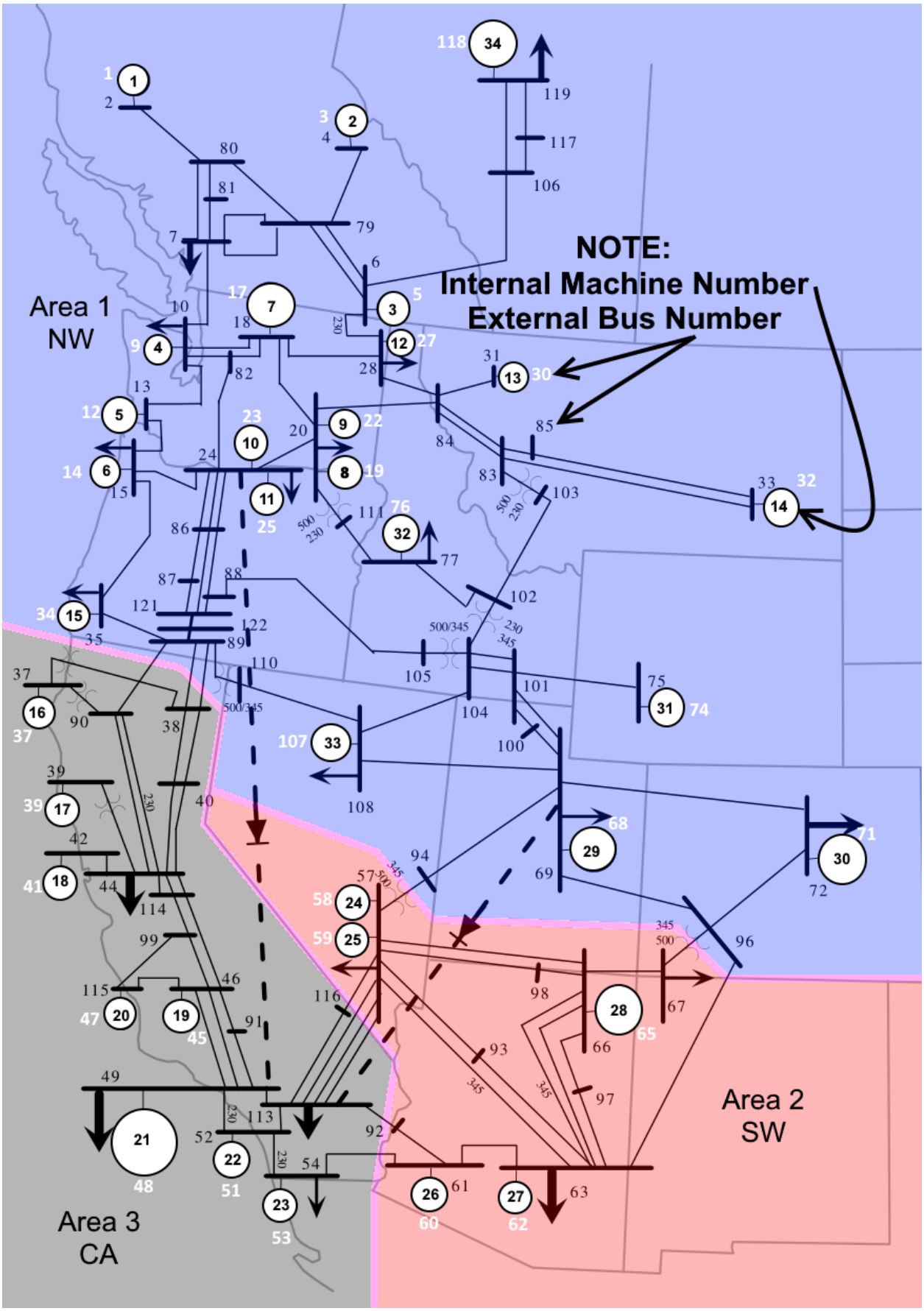


Figure B.2: MiniWECC using EIA Area Approximation.