Power System Toolbox 4

–User Manual–

Documentation Version 0.0.0-a.8

by

Thad Haines

Last Update: September 13, 2020

Power System Toolbox Introduction - WIP

Possibly written/drafted by Trudnowski...

This intro should contain a bit of first hand history about how Power System Toolbox (PST) has developed into its current form-
Why/How PST has been useful to solve/analyze engineering problems.
Previous work using PST: RJ, Sam, etc.
Mention of renewed/ongoing interest in PST via SETO/Sandia and mention of the new MIT license may be appropriate.

User Manual Introduction - WIP

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 manual provided with PST 3 [1]. 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 explanations of examples are intended to be included. Unfortunately, due to time constraints, details may be rather lacking and intentions may only matter so much.

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.

Acknowledgments - WIP

- Original Creators (graham and joe)
- Known contributors (trudnowski, ...)
- funding that made this work possible.

Table of Contents

List of Tables

List of Figures

List of Equations

Glossary of Terms

Term	**Definition**
AC	Alternating Current
ACE	Area Control Error
AGC	Automatic Generation Control
DACE	Distributed ACE
DC	Direct Current
EIA	United States Energy Information Administration
FERC	Federal Energy Regulatory Commission
FTS	Fixed Time Step
Hz	Hertz, cycles per second
J	Joule, Neton meters, Watt seconds
NERC	North American Electric Reliability Corporation
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 - WIP

- 2 - Original PST from 199x?
- 2.1 - Dan Trudnowski added batch running capability and machine tripping.
- 2.2 - Dan Trudnowski added power and current injection via the pwrmod model.
- 2.3 - Dan Trudnowski added a voltage behind a reactance model as ivmmod.
- 3.0 - From Joe Chow's website. Includes fixes and alterations. Notably multiple DC lines and PSS modifications.
- 3.1 - Based on 3.0, incorporates Dan Trudnowski's pwrmod and ivmmod models in non-linear simulation and machine tripping functionality. pwrmod included linear simulation and various model patches. Multiple generator tripping code is included, but untested.
- 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.
- SETO - Based on 3.1. New global structure introduced. Includes automatic generation control models and variable time step options.
- 4.0.0 - Based on SETO version. Includes a refined VTS routine, confirmed multi-generator tripping, improved AGC action/modulation, code cleanup, example library, and documentation. Represents the end result of work by Thad Haines.

2 Code Fixes

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 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

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 below.

```
%% 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.

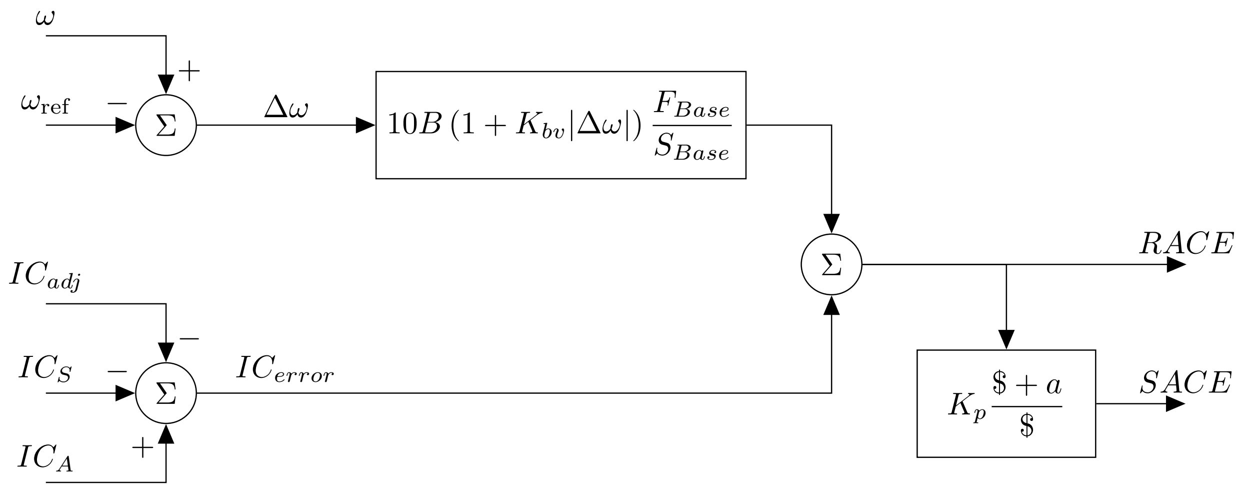

Figure 3.1: AGC calculation of $RACE$ and $SACE$.

$RACE$ 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`.

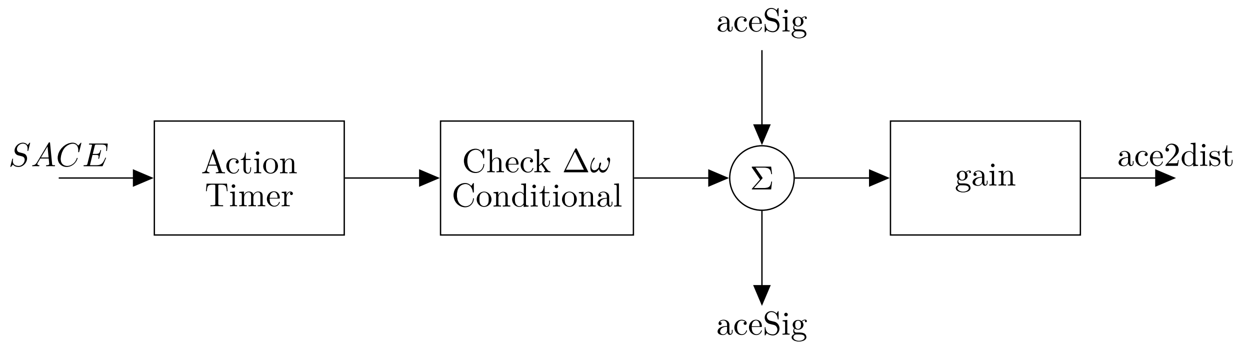

Figure 3.2: AGC calculation of *ace2dist*.

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.

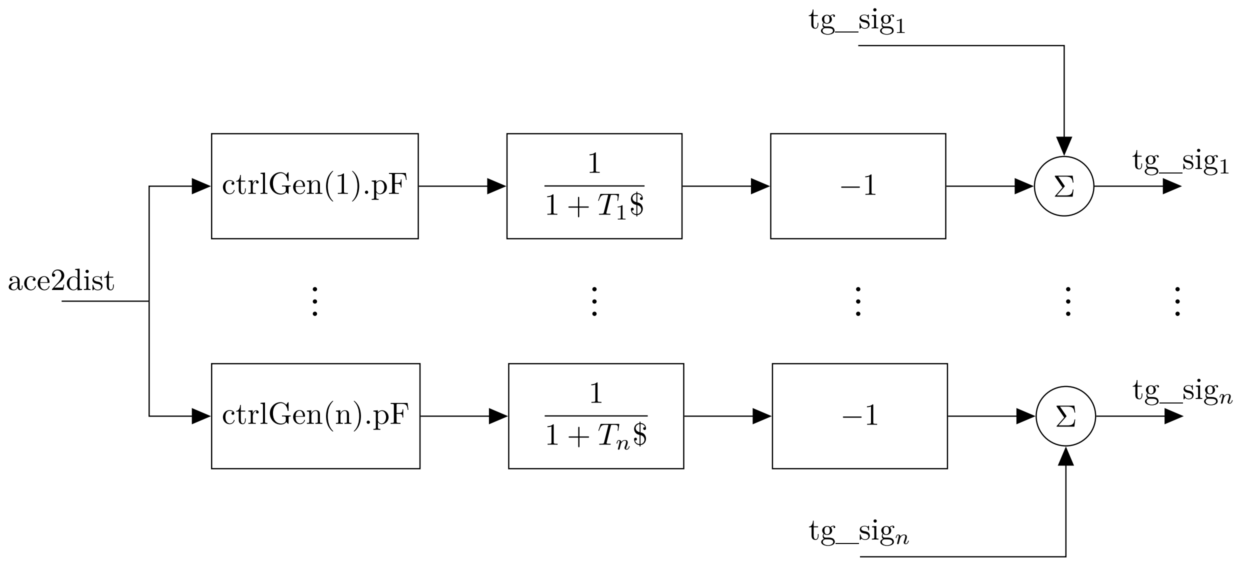

Figure 3.3: AGC handling of *ace2dist* to individual governor signals.

3.2.2 AGC Definition Example

The following AGC settings 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.

```
%% 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 (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 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. In other words, it could be assumed previous versions of PST spent half of their computation time handling globals.

Inside the global variable `g` are fields that corresponds to models, or groups, of other globals. Essentially, globals defined in the `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 [1] for a description of what most 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 previously existing 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 listed below.

```
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 similarly to AGC globals. 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 listed below.

```
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 faults 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.

```
%% 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. More specifically, the `tap` variable is re-written numerous times during a simulation when calculating line flows.

3.3.5 exc

This field contains collected global variables for exciter models, calculations, and operations.

```
%% 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.

```
%% 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.

```
%% 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.

```
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

```
global k_inc h ks h_sol
golbal 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 faults created in `y_switch`.

3.3.11 lmod

This field contains collected global variables for real load modulation models, calculations, and operations.

```
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 previously existing 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 listed below.

```
iFrom  % Comlex current injection at from bus
iTo    % Comlex 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.

```
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.

```
global  load_con load_pot nload
```

3.3.15 pss

This field contains collected global variables for power system stabilizer models, calculations, and operations.

```
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
```

Despite the renaming of the `pss_noT4_idx`, it doesn't seem to actually be used anywhere.

3.3.16 pwr

This field contains collected global variables for power or current injection models, calculations, and operations that use the `pwrmod` model.

```
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.

```
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.

```
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 seems 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.

```
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.

```
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.

```
%% 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. A description of the included variables is shown below.

```
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 matrcies, 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 handleNewGlobals Function

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) - WIP

This is the ideal voltage behind an impedance model Dan created. It's meant to model a 'grid forming' inverter where voltage and angle can be set. While there are questions about the reality of such operations and how PST simulates them, the model exists and appears to 'work' in the non-linear simulation of PST 4. There is pre-existing documentation and an example that should be condensed into this document.

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 definition of the `lmon_con` is the same as previous PST versions and an example is shown below.

```
%% 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 below.

```
% 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) - WIP

This section should cover how machines are tripped and mention later examples?

3.9 Octave Compatibility

To more fully support free and open-source software, PST 4 has been developed to be compatible with Octave. It should be noted that more experimental features, like VTS, hasn't 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

Additionally, MATLAB simulations run are generally faster than Octave. It should be noted that when using Octave, calls to save and load data must include explicit file endings (i.e. `save someData.mat`).

3.10 Power Injection Model (pwrmod) - WIP

This is the power/current injection model Dan created for version 2.3. It's meant to model the 'grid-following' type of inverters. It is included in both the non-linear and linear simulation modes of PST 4. There is pre-existing documentation and an example that should be condensed into this document.

3.11 Power System Stabilizer Model

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 below where `PSTpath` is a variable containing the full path to the root directory of PST 4.

```
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

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`.

3.13.1 Functionalization

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

3.13.1.1 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`.

3.13.1.2 correctorIntegration

As shown in the code except below, 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`.

```
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.

3.13.1.3 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.

3.13.1.4 dynamicSolution

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

3.13.1.5 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 description 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 block diagram of these actions is shown in Figure 3.4.

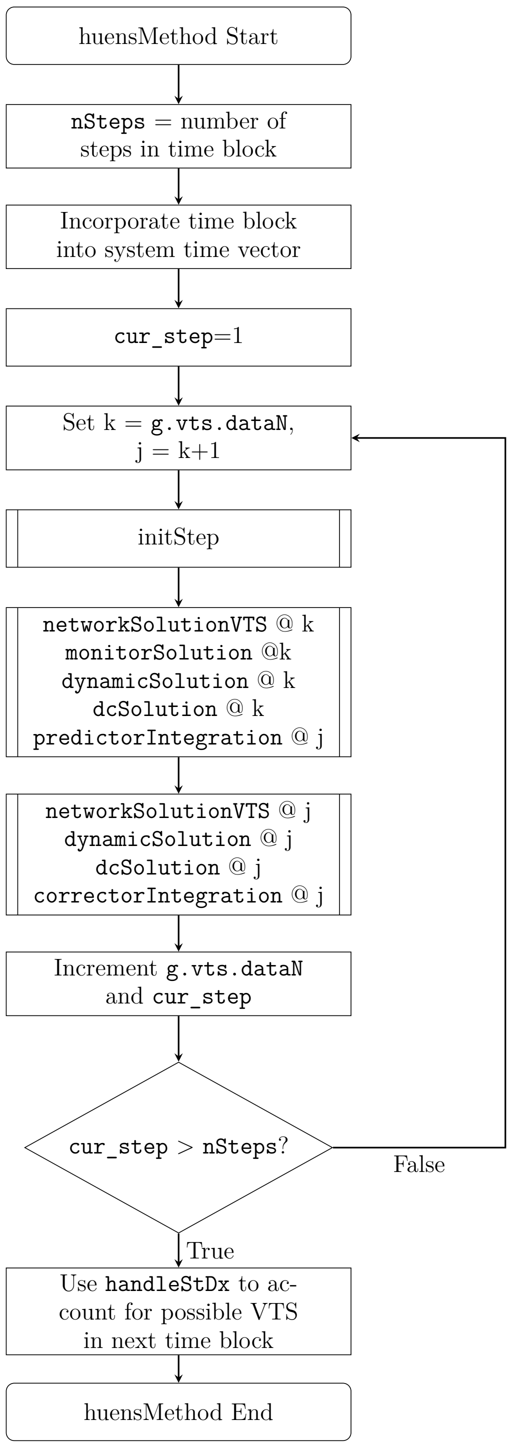

Figure 3.4: Huen's Method Block Diagram.

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.

3.13.1.6 initNLsim

The `initNLsim` function is a collection of code from `s_simu` that performs initialization operations before a non-linear simulation. This is essentially 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.

3.13.1.7 initStep

Code from `s_simu` that was performed 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, as well as handling machine trip flags.

3.13.1.8 initTblocks

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

3.13.1.9 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.

```
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
```

3.13.1.10 monitorSolution

The `monitorSolution` function takes a single input that defines the data index used to calculate any user defined line monitoring values, average system and/or area 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.

3.13.1.11 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 `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 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.

3.13.1.12 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`.

```matlab
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
```

3.13.1.13 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, it can be run independently from the simulation to analyze a pre-existing global `g` by being invoked as `standAlonePlot(1)`.

3.13.1.14 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 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 deemed fitting. 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`.

```
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 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 John Undrill. The mac_sub_NEW2 model is the same as the _NEW model with minor bug fixes and alterations by Dan Trudnowski. Any model may be copied over the mac_sub file for simulation use as shown below.

```
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
```

3.15 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

3.16 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 theme behind VTS is that when the system is moving 'slowly', larger time steps could accurately capture system dynamics. Obviously, as time step increases, fewer solutions are required to simulate a set period of time.

VTS simulations are made possible by using standard MATLAB ODE solvers. This decision was made to explore the viability of applying VTS methods to a power system. As the time step of these solvers is dependent upon analysis of system derivatives, 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 still experimental and should be used with caution.

3.16.1 Solver Control Array (solver_con)

To use VTS integration methods, a user will have to 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 below.

```
%% 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 hundreds 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)
    };
```

3.16.2 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 3.5.

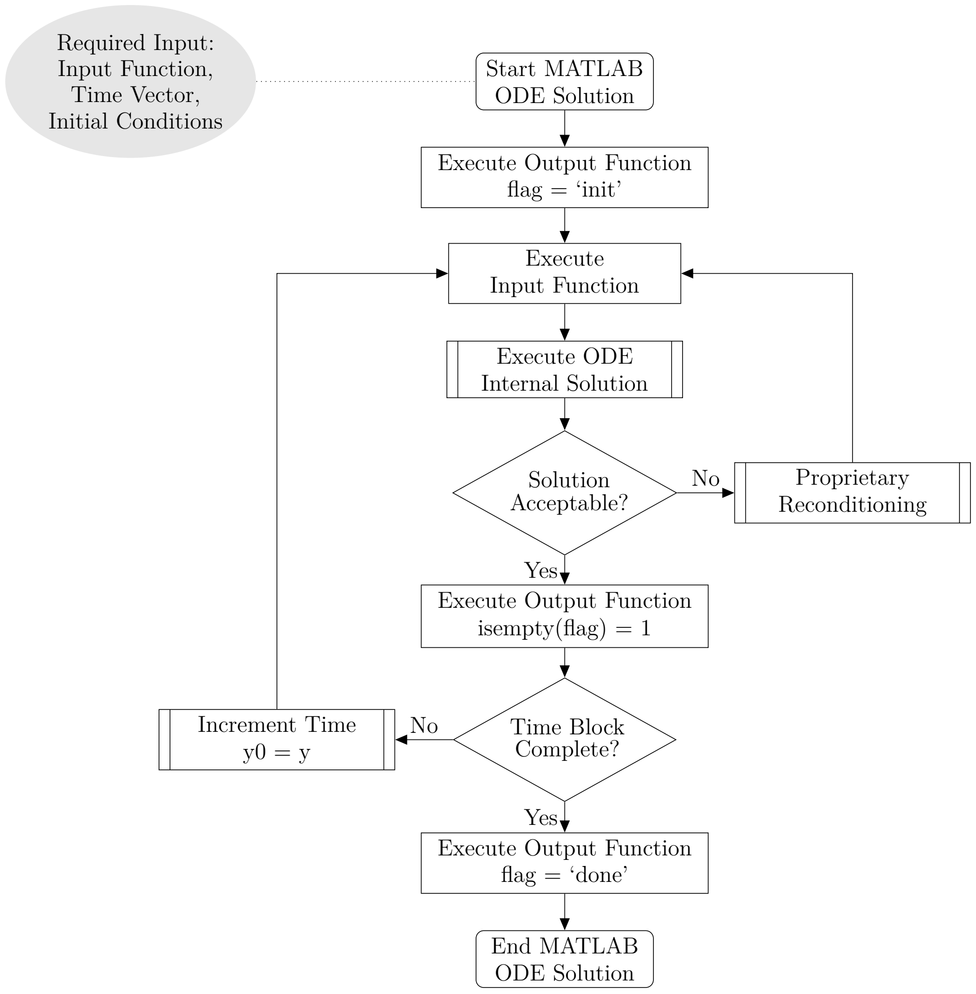
Required Input:
Input Function,
Time Vector,
Initial Conditions
Start MATLAB ODE Solution
Execute Output Function flag = 'init'
Execute Input Function
Execute ODE Internal Solution
Solution Acceptable?
No
Proprietary Reconditioning
Yes
Execute Output Function isempty(flag) = 1
Time Block Complete?
No
Increment Time y0 = y
Yes
Execute Output Function flag = 'done'
End MATLAB ODE Solution

Figure 3.5: MATLAB ODE block diagram.

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 are shown below and deal with error tolerance levels, initial step size, max step size, and an Output function.

```
% 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
```

3.16.3 Functions Specific to VTS

A number of new functions were created to enable VTS to be integrated into PST. Most functions were created to handle data or perform other tasks specifically related to VTS. The following sections provide information about such functions.

3.16.3.1 vtsInputFcn

To use the MATLAB ODE solvers, a function must be passed in that returns a vector of derivatives associated with the states to integrate. The `vtsInputFcn` was created to perform this critical task. The slightly abbreviated input function is shown below.

```
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

%% call handleStDx with flag==2 to update global states with newest passed in soln.
% write slnVec vector of values to associated states at index k
% i.e. update states at g.vts.dataN with newest solution
handleStDx(g.vts.dataN, y, 2)

%% Start initStep action ==================================================
initStep(g.vts.dataN)

%% Start of Network Solution ==============================================
networkSolutionVTS(g.vts.dataN, t)
```

```
%% Start Dynamic Solution ===================================================
dynamicSolution(g.vts.dataN )

%% Start of DC solution =====================================================
dcSolution(g.vts.dataN )

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

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

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

3.16.3.2 vtsOutputFcn

After each acceptable solution, the ODE solver calls a passed in ‘output function’. The `vtsOutputFcn` was created to handle restoring the network solution and performing the monitor solution. Additionally, the `vtsOutputFcn` handles required indexing and iteration accumulation and logging. The slightly abbreviated output function is shown below.

```
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
    % restore network to initial solution
    handleNetworkSln(g.vts.dataN ,2) % may cause issues with DC.
    monitorSolution(g.vts.dataN); % Perform Line Monitoring and Area Calculations

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

    % after each successful integration step by ODE solver:
    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
```

3.16.3.3 handleNetworkSln

The `handleNetworkSln` function was created to store, and restore, calculated values set to globals 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 may occur during the MATLAB ODE solver's repeated calls to the input function. As shown below, `handlNetworkSln` takes a data index `k` and an operation `flag` as inputs.

```
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 a global vector
%       2 - write stored network solution vector to network globals data at index k
```

3.16.3.4 handleStDx

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

```
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 this function 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 globals. 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 required (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 and pwrmod/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` and it seems very possible to add more models that require integration as they arise.

4 PST Operation Overview - WIP

This is likely to include a flow chart/ multiple flow charts.

Mention partitioned explicit method via Huen's, partitioned implicit via VTS.

Overview of running `s_simu` in batch or stand alone mode.

copying of modulation files

4.1 Simulation Loop - WIP

The complete simulation loop code is shown below. This code was copied from `s_simu_BatchVTS` with corresponding line numbers.

```
362 %% Simulation loop start
363 warning('*** Simulation Loop Start')
364 for simTblock = 1:size(g.vts.t_block)
365 
366     g.vts.t_blockN = simTblock;
367     g.k.ks = simTblock; % required for huen's solution method.
368 
369     if ~isempty(g.vts.solver_con)
370         odeName = g.vts.solver_con{g.vts.t_blockN};
371     else
372         odeName = 'huens'; % default PST solver
373     end
374 
375     if strcmp( odeName, 'huens')
376         % use standard PST huens method
377         fprintf('*** Using Huen''s integration method for time block %d\n*** t=[%7.4f,
        ↪  %7.4f]\n', ...
378             simTblock, g.vts.fts{simTblock}(1), g.vts.fts{simTblock}(end))
379 
380         % add fixed time vector to system time vector
381         nSteps = length(g.vts.fts{simTblock});
382         g.sys.t(g.vts.dataN:g.vts.dataN+nSteps-1) = g.vts.fts{simTblock};
383 
384         % account for pretictor last step time check
385         g.sys.t(g.vts.dataN+nSteps) = g.sys.t(g.vts.dataN+nSteps-1)+
        ↪  g.sys.sw_con(simTblock,7);
386 
387         for cur_Step = 1:nSteps
```

```
            k = g.vts.dataN;
            j = k+1;

            % display k and t at every first, last, and 50th step
            if ( mod(k,50)==0 ) || cur_Step == 1 || cur_Step == nSteps
                fprintf('*** k = %5d, \tt(k) = %7.4f\n',k,g.sys.t(k)) % DEBUG
            end

            %% Time step start
            initStep(k)

            %% Predictor Solution =========================================
            networkSolutionVTS(k, g.sys.t(k))
            monitorSolution(k);
            dynamicSolution(k)
            dcSolution(k)
            predictorIntegration(k, j, g.k.h_sol)   % g.k.h_sol updated i_simu

            %% Corrector Solution =========================================
            networkSolutionVTS(j, g.sys.t(j))
            dynamicSolution(j)
            dcSolution(j)
            correctorIntegration(k, j, g.k.h_sol)

            % most recent network solution based on completely calculated states is k
            monitorSolution(k);
            %% Live plot call
            if g.sys.livePlotFlag
                livePlot(k)
            end

            g.vts.dataN = j;                        % increment data counter
            g.vts.tot_iter = g.vts.tot_iter  + 2;   % increment total solution counter
            g.vts.slns(g.vts.dataN) = 2;            % track step solution
        end
        % Account for next time block using VTS
        handleStDx(j, [], 3) % update g.vts.stVec to initial conditions of states
        handleStDx(k, [], 1) % update g.vts.dxVec to initial conditions of derivatives

    else % use given variable method
        fprintf('*** Using %s integration method for time block %d\n*** t=[%7.4f,
        ↪ %7.4f]\n', ...
            odeName, simTblock, g.vts.t_block(simTblock, 1), g.vts.t_block(simTblock,
            ↪ 2))

        % feval used for ODE call - could be replaced with if statements.
        feval(odeName, inputFcn, g.vts.t_block(simTblock,:), g.vts.stVec , options);

        % Alternative example of using actual function name:
        %ode113(inputFcn, g.vts.t_block(simTblock,:), g.vts.stVec , options);
    end

end% end simulation loop
```

5 Examples Explained - WIP

As github as many examples, this is a chapter to explain what the purpose of them is and detail non-linear vs linear operation of examples that do such things.

- Experimental VTS
- AGC
- AGC interchange modulation
- Extended term with VTS
- Hiskens NE model via Ryan
- MiniWECC via Dan
- Experimental Un-tripping
- Modulation via `_mod` files
- Linear simulation where applicable...
- Combine examples?

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)`

5.1 Automatic Generation Control - WIP

A variety of examples were created for AGC testing. While some may use VTS, this section covers information only related to AGC. The VTS example section contains VTS specific information.

The `0-examples/AGC` folder contains all system and modulation files associated with the AGC examples. The `run***` files are designed to run PST in using the batch mode.

5.2 Variable Time Step - WIP

6 Loose Ends - WIP

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.
   It probably shouldn’t be.
4. Constant Power or Current loads seem to require a portion of constant Impedance.
5. PSS design functionality not explored.
6. No examples of of delta P omega filter or user defined damping controls for SVC and TCSC models
7. Differences in `mac_ind` between PST 2 and 3 seem backward compatible, but this is untested.
8. DC is not implemented in VTS. It seems like DC models should be combined into the main routine if so desired. Seems counter intuitive / (not very possible) 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.

7 Bibliography

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

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

[3] 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.

8 Document History - WIP

- 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

A Appendix A: Formatting Examples

This appendix is included to show how appendices work, blowing up of numbering, and to also serve as an easy LaTeX formatting template. Despite this being an appendix, it is still numbered like a chapter.

A.1 Numberings in Equations

Additionally, a reminder of Ohm's law

$$V = IR \tag{A.42}$$

shows that equation numbering has blown up. This is to show spacing on the table of contents.

A.2 Numberings in Tables

Table A.14 is full of nonsense data and is numbered artificially large.

Table A.14: Another Table.

One	Two	Three	Four	Five
2.35	45.87	9.00	1.00	0.33
5.88	48.01	7.85	2.35	0.45

A.3 Numberings in Figures

Finally, Figure A.67 shows how figure numbers look when double digit.

Figure A.67: A boxcat in its natural environment.

A.4 Code using Minted

Code can be added using the `minted` package. The example below is for a Python example, but can be configured other ways. Note that a `--shell-escape` command had to be added to the TEXStudio build command due to peculiarities with the package. Additionally, the `minted` LATEX package requires python to be installed with the `pygments` package. This can be done via the '`py -m pip install -U pygments`' console command.

```python
def sumPe(mirror):
    """Returns sum of all electrical power from active machines"""
    sysPe = 0.0

    # for each area
    for area in mirror.Area:
        # reset current sum
        area.cv['Pe'] = 0.0

        # sum each active machine Pe to area agent
        for mach in area.Machines:
            if mach.cv['St'] == 1:
                area.cv['Pe'] += mach.cv['Pe']

        # sum area agent totals to system
        sysPe += area.cv['Pe']

    return sysPe
```

Figure A.68: Code listing as a figure.

Other packages exist for code insertion, but they may or may not be as pretty. Remember, as with anything, this is totally optional and voluntary...