Power System Toolbox 4

–User Manual–
Documentation Version 0.0.0-a.3

by
Thad Haines

Last Update: September 8, 2020

Introduction

This section should a bit of history about how PST has developed into its current form. Mention of new MIT license seems appropriate.

A brief explanation of the user manual purpose and major content would make sense.

Highlight/foreshadow major changes and additions that are explained in more detail later.

Acknowledgments

- 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
BA	Balancing Authority
BES	Bulk Electrical System
CTS	Classical Transient Stability
DACE	Distributed ACE
EIA	United States Energy Information Administration
FERC	Federal Energy Regulatory Commission
FTS	Fixed Time Step
Hz	Hertz, cycles per second
IC	Interchange
ISO	Independent Service Operator
J	Joule, Neton meters, Watt seconds
LFC	Load Frequency Control
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
RTO	Regional Transmission Organization
SACE	Smoothed ACE
SI	International System of Units
US	United States of America

Term	Definition
VAR	Volt Amps Reactive
VTS	Variable Time Step
W	Watt, Joules per second
WECC	Western Electricity Coordinating Council

1 PST Version History

- 2.3 - Based on PST 2, includes Dan Trudnowski's pwrmod, ivmmod, and generator tripping modifications.
- 3.0 - From Joe Chow's website. Includes fixes and alterations. Notably multiple DC lines and PSS modifications.
- 3.1 - Based on 3.0, includes Dan Trudnowski's pwrmod and ivmmod models in non-linear simulation. pwrmod included linear simulation and various model patches. Multiple generator tripping code is included, but untested.
- 3.1.1 - Based on 3.1. Ryan Elliott's version with energy storage and updated linear simulation along with various other fixes and code cleanup alterations.
- SETO - Based on 3.1. New global structure. Includes automatic generation (AGC) models and variable time step (VTS) options. Will become 4.0 after clean up actions taken.
- 4.0.0-aX - Alpha version of PST 4 based on SETO version. Includes a refined VTS routine, confirmed multi-generator tripping, and improved AGC action/modulation.
- 4.0.0 - Finished VTS, AGC, code cleanup, example library, and documentation work by Thad Haines.

2 Code Fixes

The purpose of this section is to record fixes to PST code from version 3 to version 4.

2.1 exc_dc12

In 2015 there were 'errors' corrected in the saturation block that create differences between version 2 and 3 of this model. Effects are noticeable, but a solution hasn't been investigated yet.

2.2 exc_st3

Corrected theta index to `n_bus` from `n` per Ryan Elliott. Corrected simple `*` to `.*` int the `if ~isempty(nst3_sub)` section.

2.3 lmod

Fixed state limiting. While if over-limit, the derivative was set to zero, the state was not set to the maximum/minimum value correctly.

2.4 mac_tra

Commented out code that prevented the setting equal of the transient reactances.

2.5 rlmod

Fixed state limiting. While if over-limit, the derivative was set to zero, the state was not set to the maximum/minimum value correctly.

3 Changes and Additions

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

3.1 Area Definitions

Created area functionality via the `area_def`. The `area_indx` function creates the required data structures and indices.

To enable allow for AGC and the required area calculations, each bus in the `bus` array must be assigned to an area in the `area_def` array. An example `area_def` array is shown below.

```
%% area_def data format
% 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];
```

It should be noted that rows may not have to be in the same order as the bus array (untested). The `area_def` array is automatically placed into the global g structure.

```
>> g.area
ans =
    area_def: [13x2 double]
      n_area: 2
```

```
        area: [1x2 struct]
```

Each area currently contains values that may be relevant to AGC calculations. The `calcAreaVals` function is used to calculate and store such values. An example of what is stored in the `g.area.area(x)` structure is shown below.

```
>> g.area.area(2)
ans =
         number: 2
      areaBuses: [6x1 double]
         macBus: [2x1 double]
      macBusNdx: [3 4]
        loadBus: [4x1 double]
     loadBusNdx: [8 9 12 13]
         genBus: [2x1 double]
      genBusNdx: [6 7]
           totH: [1x4063 double]
           aveF: [1x4063 double]
         totGen: [1x4063 double]
        totLoad: []
            icA: [1x4063 double]
            icS: []
  exportLineNdx: [11 12]
  importLineNdx: []
       n_export: 2
       n_import: []
```

It should be noted that `icS` represents a placeholder for a scheduled interchange value and the `totLoad` is a field for collected total load. The collection of actual running load values may prove more complicated as the bus array does not seem to be updated every step, only a reduced Y matrix used in the `nc_load` function called from `i_simu`.

3.2 Automatic Generation Control (AGC)

Automatic generation control model that calculates RACE and distributes signal to defined controlled generators according to participation factor. The ACE signal is filtered through a PI before being distributed to each generator through a unique low pass filter that adds the negative of the value to the governor `tg_sig` value. (NOTE: the `tg_sig` value is equivalent to an addition to the governors P_{ref} value) The `agc_indx` function creates the required data structures and indices.

3.2.1 AGC Block Diagrams

The AGC process is shown in Figures 3.1 -3.3. RACE and SACE 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 $RACE$ value. The scheduled interchange may be altered by the user via a `mAGC_sig` file that controls the behavior of the IC_{adj} input.

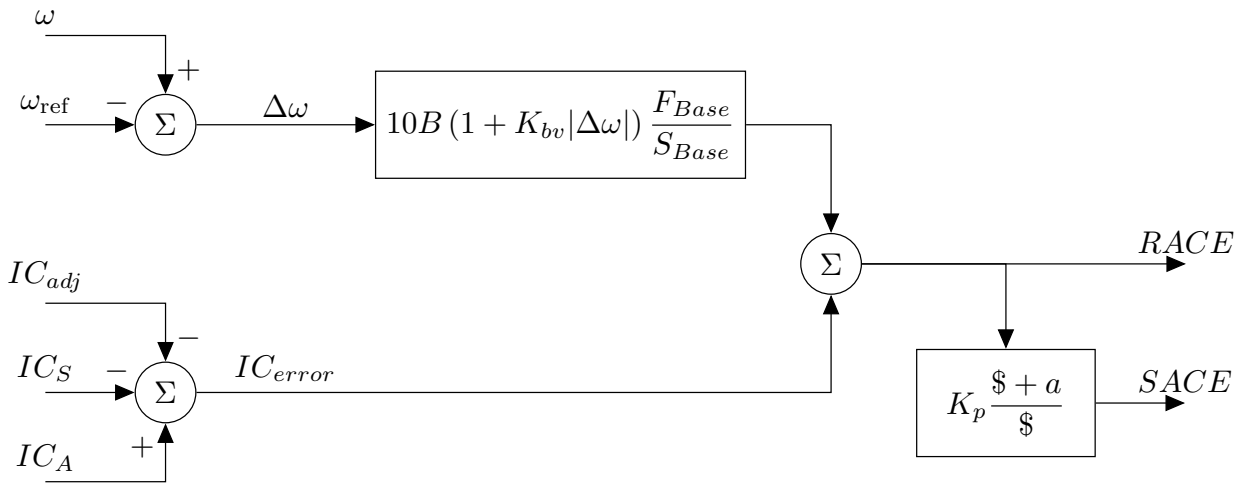

Figure 3.1: AGC calculation of $RACE$ and $SACE$.

RACE and *SACE* are calculated every simulation time step, however distribution of *SACE* is determined by the `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`.

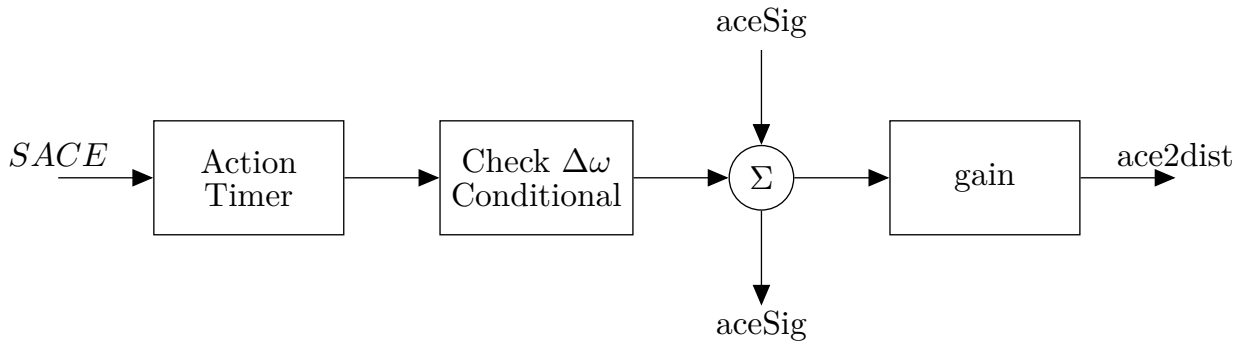

Figure 3.2: AGC calculation of *ace2dist*.

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 that processes the signal that is then gained by -1 and added to the existing associated governor `tg_sig` value.

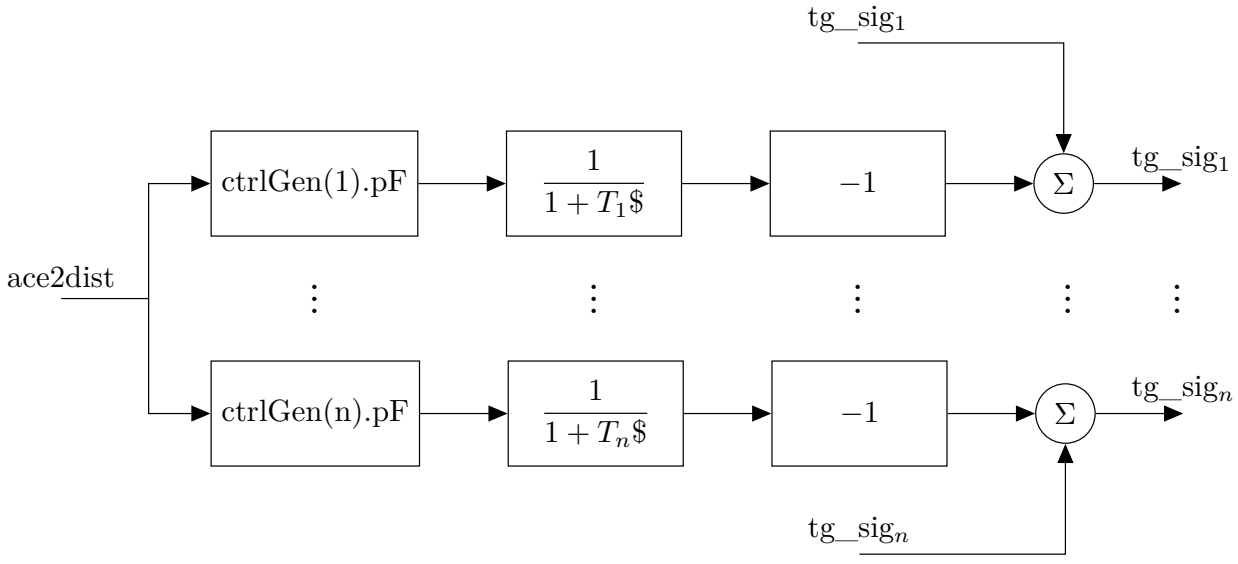

Figure 3.3: AGC handling of *ace2dist* to individual governor signals.

3.2.2 Weighted Average Frequency (calcAveF)

Calculates system and area weighted average frequency. System values stored in `g.sys.aveF` and area values stored in `g.area.area(x).aveF`.

An average weighted frequency is calculated for the total system and for each area if there are areas detected. The calculation involves a sum of system inertias that may change with generator trips. The current algorithm does not account for tripped generators, but was designed to incorporate this feature in the future.

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:

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

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

Then 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

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

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

3.2.3 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

To enable easier manipulation of PST - it was decided to create a global structure that contains all system globals. While this may or may not have been a good idea - it happened. Initial results show a speed up of over 2 times. In other words, it could be assumed previous versions of PST spent half of their computation time loading globals…

Inside the global variable `g` are fields that corresponds to models, or groups, of other globals. For example, the `g.mac.mac_spd` global contains a all machine speeds while the `g.bus.bus_v` contains all bus voltages, etc. As of this writing, compiled on September 8, 2020, the following subsections describe the globals contained in each field of the global `g`.

3.3.1 agc

This contains variables for agc calculation and operation. An example of AGC variables are shown below

```
>> g.agc
ans =
      agc: [1x2 struct]
    n_agc: 2
do better
```

3.3.2 area

This contains variables for area calculation and operation. An example of area variables are shown below.

```
>> g.area
ans =
    area_def: [13x2 double]
      n_area: 2
        area: [1x2 struct]
```

```
>> g.area.area(1)
do better
```

3.3.3 bus

Contains `bus` and all altered bus arrays associated with faults created in `y_switch`. Also contains the `bus_v` and `theta` information related to buses.

3.3.4 dc

```
%% 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 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.

3.3.5 exc

```
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 % pm_sig n_pm % not related to exciters?
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

```
%% induction genertaor variables - 19
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

```
%% induction motor variables - 21
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

voltage model...

```
get the tstuff
```

3.3.9 k

To allow for functionalized running, various index values were placed into the global structure

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

3.3.10 line

Contains `line` and all altered line arrays associated with faults created in `y_switch`.

3.3.11 lmod

```
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

This contains variables for line monitoring not previously collected during simulation. An example of lmon contents is shown below.

```
>> g.lmon
ans =
     lmon_con: [3 10]
       n_lmon: 2
```

```
    busFromTo: [2x2 double]
         line: [1x2 struct]
>> g.lmon.line(1)
ans =
     busArrayNdx: 3
         FromBus: 3
           ToBus: 4
           iFrom: [1x8751 double]
             iTo: [1x8751 double]
           sFrom: [1x8751 double]
             sTo: [1x8751 double]
```

3.3.13 mac

```
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

```
global  load_con load_pot nload
```

3.3.15 pss

```
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

```
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
```

There are some cells that contain user defined derivatives that aren't included yet.

3.3.17 rlmod

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

3.3.18 svc

```
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 some code related to user defined damping control of SVC, but it is not described in the user manual. (Added by Graham around 98/99)

3.3.19 sys

```
global  basmva basrad syn_ref mach_ref sys_freq
```

3.3.20 tcsc

```
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, there seems to be some added functionality for controlled damping, but no examples exist? (Added by Graham around 98/99)

3.3.21 tg

```
%% 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 seemed to use it.

3.3.22 vts

Globals associated with variable time step simulation runs were placed in the `g.vts` field.

```
dataN % used as a the data index for logging values
fsdn % a cell of fields, states, derivatives, and number of states
n_states % total system state count
dxVec % vector used to pass around current dataN derivatives
stVec % vector used to pass around current dataN states
t_block % a list of time blocks collected from sw_con
t_blockN % current time block being executed
iter % counter to monitor number of solutions per step
tot_iter % total number of solutions
slns % a running history of solution iterations
```

3.3.23 y

Contains reduced Y matrices, voltage recovery matrcies, and bus order variables created in `y_switch` associated with faults. Example variables are shown below.

```
>> g.int
ans =
      Y_gprf: [4x4 double]
do better....
```

3.4 ivmmod

This is the voltage behind an impedance model Dan created. It's meant to model a 'grid forming' converter where voltage and angle are manipulatable. While there are

questions about the reality of such operations, the model exists and appears to work in the non-linear simulation of PST 4. There is documentation that should probably be moved into this document.

3.5 livePlot Function

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

There are currently XXX 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.

3.6 Line Monitoring Model (lmon)

Coded to actually monitor line current and power flow during non-linear simulation. The `lmon_indx` function creates the required data structures and indices.

Power flow on a line must be calculated each step as AGC requires actual area interchange for the ACE calculation. The previously existing `line_pq` function performed this task, but was not fully incorporated into the simulation to allow calculation during execution. This minor oversight has been resolved and the `lmon_con` array is still used to define monitored lines as in previous version of PST. It should be noted that areas with AGC automatically track area interchange flows and do not need to be user defined.

```
%% 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 = [5, 6, 13]; % lines between bus 3 and 101, and line between 13 and 120

lmon_con = [3,10]; % lines to loads
```

Line monitoring data is collected in the `g.lmon` field of the global variable.

```
>> g.lmon
ans =
     lmon_con: [3 10]
       n_lmon: 2
    busFromTo: [2x2 double]
         line: [1x2 struct]
```

Each `g.lmon.line` entry contains the following fields and logged data:

```
>> g.lmon.line(2)
ans =
    busArrayNdx: 10
        FromBus: 13
```

```
        ToBus: 14
        iFrom: [1x4063 double]
          iTo: [1x4063 double]
        sFrom: [1x4063 double]
          sTo: [1x4063 double]
```

3.7 Sub-Transient Machine Models

There are three versions of the sub-transient machine model (`mac_sub`) included with PST 4. The `_ORIG` model is the standard PST model based on the R. P. Schulz, "Synchronous machine modeling" algorithm. The `_NEW` model is based on the PSLF 'genrou' model by John Undrill. The `_NEW2` model is the same as the `_NEW` model with minor alterations. Any model may be copied over the `mac_sub` file for use.

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

3.8 Machine Trip Logic

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

3.9 Octave Compatibility

To more fully support free and open-source code, PST 4 has been developed to be compatible with Octave.

Mention VTS maybe not so super compatible. Most other things are.

3.10 Power System Stabilizer (PSS) Model

There was a modification to the washout time constant in the PSS model between PST version 2 and 3 that affects the output of the model in fairly drastic ways. 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.

```
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
```

The default PSS model used in PST 4 is `pss2`.

While this works, it's kind of confusing and may be removed.

3.11 pwrmod

This is the power or current injection model Dan created for version 2.3. It's meant to model the 'grid following' type of converters. It is included in both the non-linear and linear simulation modes of PST 4. There is documentation that should probably be moved into this document.

3.12 s_simu Changes

This file was huge. It has been functionalized and cleaned up. still has stand alone mode.

3.13 Variable Time Step (VTS) Simulation

Variable time step simulations are possible using standard MATLAB ODE solvers. A semi-complete and partially-detailed explanation of the functions and code used to make this happen will *probably* be written in a separate document 'later'…

Explain FTS and VTS

Cover speed up possibility

explain zeroing derivatives.

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

3.13.1 Solver Control Array

Theoretically, a user will only have to add a `solver_con` to a valid data file to use variable time step methods. If a `solver_con` is not specified, Huen's method is used for all

time blocks (i.e. default PST 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 (with code comments) 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 hundereds of iterations, sometimes not... decent
% 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
    'huens'; % sw_con row 5 - sw_con row 6
    'ode23t'; % sw_con row 6  - sw_con row 7  (end)
    };
```

As of this writing, the pstSETO version uses the `s_simu_BatchVTS` script to perform variable time stepping methods. This script will likely replace the `s_simu` script in the main PST4 folder after versioning is complete.

3.13.2 MATLAB ODE solver

The variable time step implementation in PST revolves around using the built in MATLAB ODE solvers. All these methods perform actions depicted in the following block

diagram.

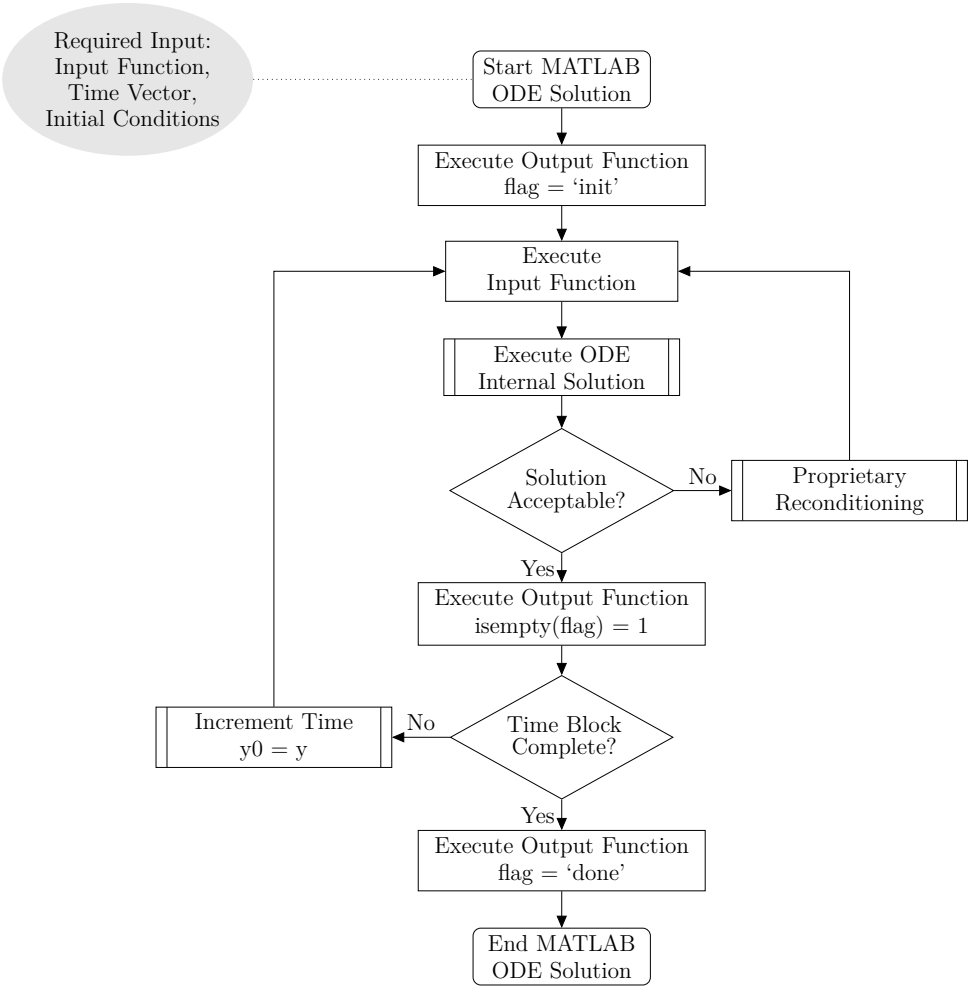

Figure 3.4: MATLAB ODE block diagram.

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
```

3.13.2.1 vtsInputFcn

The slightly abbreviated (mostly complete) 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
```

3.13.2.2 vtsOutputFcn

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
```

3.13.3 Functions for Variable Time Step Integration

A number of new functions were created to allow for VTS to be integrated into PST and collected in the `test` folder of the main SETO version directory. Some functions were simply portions of code previously located in `s_simu` and placed into a function for ease of use and clarity of code flow, while others were created to handle data or perform other tasks specifically related to VTS. The following sub paragraphs provide some information about such functions that have not been previously introduced.

3.13.3.1 cleanZeros

Function that cleans all zero variables from the global g. Executed at the end of `s_simu_Batch`. Stores cleared variable names in the `clearedVars` cell that is stored in `g.sys`

3.13.3.2 trimLogs

Function that trims all logged values in the global g to a given length `k`. Executed near the end of `s_simu_BatchXXX` before cleanZeros.

3.13.3.3 s_simu

The `s_simu_BatchTestF` script is a modified version of `s_simu_Batch` that was used to test the new functions used in non-linear simulation outside of the variable time step process. As the VTS method appears to work, this script will most likely go away as it has served its purpose.

3.13.3.4 s_simu_BatchVTS

The `s_simu_BatchVTS` script is a functionalized `s_simu_Batch` with elements from `s_simu` that prompt user input re-introduced. To enter *stand alone mode* (where the user is

prompted for input), simply run this script after issuing the `clear all; close all` commands. While being able to operate in stand alone mode, it is also able to run in *batch mode* where it assumes the data file to run is the `DataFile.m` in the root PST directory. This script performs optional VTS simulation and is slated to replace `s_simu` once PST SETO becomes PST 4.0.

3.13.3.5 initZeros

A large amount of code (≈400 lines) in `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
```

3.13.3.6 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.

3.13.3.7 initTblocks

The `initiTblocks` function analyzes the global `sw_con` and `solver_con` to create appropriate *time blocks* that are used in VTS 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.

3.13.3.8 initStep

Code from `s_simu` that was performed at the begining 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.

3.13.3.9 networkSolution

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 located in this function. The input to this function is the data index on which to operate.

3.13.3.10 networkSolutionVTS

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.

3.13.3.11 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`.

3.13.3.12 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 developement. It *should* work as normal when using Huen's method, but is untested as of this writing.

3.13.3.13 monitorSolution

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

3.13.3.14 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.

```
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
```

3.13.3.15 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`.

```
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.

3.13.3.16 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 the SETO version of PST 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 and pwrmod functionality has been added to `handleStDx` and it seems very possible to add more models that require integration as they arise.

3.13.3.17 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 solvers repeated call 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
```

3.13.3.18 trimLogs

As there is no way to accurately predict the amount of (length of) data to be logged during a variable time step simulation, more space is allocated (20x the amount from a fixed step simulation) and then all logged values are trimmed to the proper length post simulation. It should be noted that this 20x size allocation was arbitrary and will probably be altered in the future as actual extended term simulation using VTS typically requires fewer steps than a fixed step method. However, if not enough space is allocated the simulation will crash.

```
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
```

3.13.3.19 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_BatchVTS` if stand alone mode is detected.

4 PST Operation Overview

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

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
```

```
388             k = g.vts.dataN;
389             j = k+1;
390 
391             % display k and t at every first, last, and 50th step
392             if ( mod(k,50)==0 ) || cur_Step == 1 || cur_Step == nSteps
393                 fprintf('*** k = %5d, \tt(k) = %7.4f\n',k,g.sys.t(k)) % DEBUG
394             end
395 
396             %% Time step start
397             initStep(k)
398 
399             %% Predictor Solution ========================================
400             networkSolutionVTS(k, g.sys.t(k))
401             monitorSolution(k);
402             dynamicSolution(k)
403             dcSolution(k)
404             predictorIntegration(k, j, g.k.h_sol)   % g.k.h_sol updated i_simu
405 
406             %% Corrector Solution ========================================
407             networkSolutionVTS(j, g.sys.t(j))
408             dynamicSolution(j)
409             dcSolution(j)
410             correctorIntegration(k, j, g.k.h_sol)
411 
412             % most recent network solution based on completely calculated states is k
413             monitorSolution(k);
414             %% Live plot call
415             if g.sys.livePlotFlag
416                 livePlot(k)
417             end
418 
419             g.vts.dataN = j;                        % increment data counter
420             g.vts.tot_iter = g.vts.tot_iter  + 2;   % increment total solution counter
421             g.vts.slns(g.vts.dataN) = 2;            % track step solution
422         end
423         % Account for next time block using VTS
424         handleStDx(j, [], 3) % update g.vts.stVec to initial conditions of states
425         handleStDx(k, [], 1) % update g.vts.dxVec to initial conditions of derivatives
426 
427     else % use given variable method
428         fprintf('*** Using %s integration method for time block %d\n*** t=[%7.4f,
        ↪  %7.4f]\n', ...
429             odeName, simTblock, g.vts.t_block(simTblock, 1), g.vts.t_block(simTblock,
            ↪  2))
```

```
430
431         % feval used for ODE call - could be replaced with if statements.
432         feval(odeName, inputFcn, g.vts.t_block(simTblock,:), g.vts.stVec , options);
433
434         % Alternative example of using actual function name:
435         %ode113(inputFcn, g.vts.t_block(simTblock,:), g.vts.stVec , options);
436     end
437
438 end% end simulation loop
```

5 Examples Explained

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

6 Loose Ends

As software development is never actually over, this chapter is meant to contain any loose ends that felt relevant. The below is copied from the most recent weekly (09/02/20) and some minor additions

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 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 - untested.
8. DC is not implemented in VTS - Just combine into main routine? Seems counter intuitive 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.

7 Bibliography

[1] Joe Chow, Graham Rogers, *Power System Toolbox Version 3.0 User Manual*, 2008.

8 Document History

It'd make sense if this turned into a table...

- 09/02/20 - 0.0.0-a.1 - document creation...
- 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,

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.

```
1  def sumPe(mirror):
2      """Returns sum of all electrical power from active machines"""
3      sysPe = 0.0
4
5      # for each area
6      for area in mirror.Area:
7          # reset current sum
8          area.cv['Pe'] = 0.0
9
10         # sum each active machine Pe to area agent
11         for mach in area.Machines:
12             if mach.cv['St'] == 1:
13                 area.cv['Pe'] += mach.cv['Pe']
14
15         # sum area agent totals to system
16         sysPe += area.cv['Pe']
17
18     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...