Purpose

This document is meant to explain PST additions and alterations created to accommodate VTS (variable time stepping). While the current method works, it may change in the future.

Solution Control Array

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

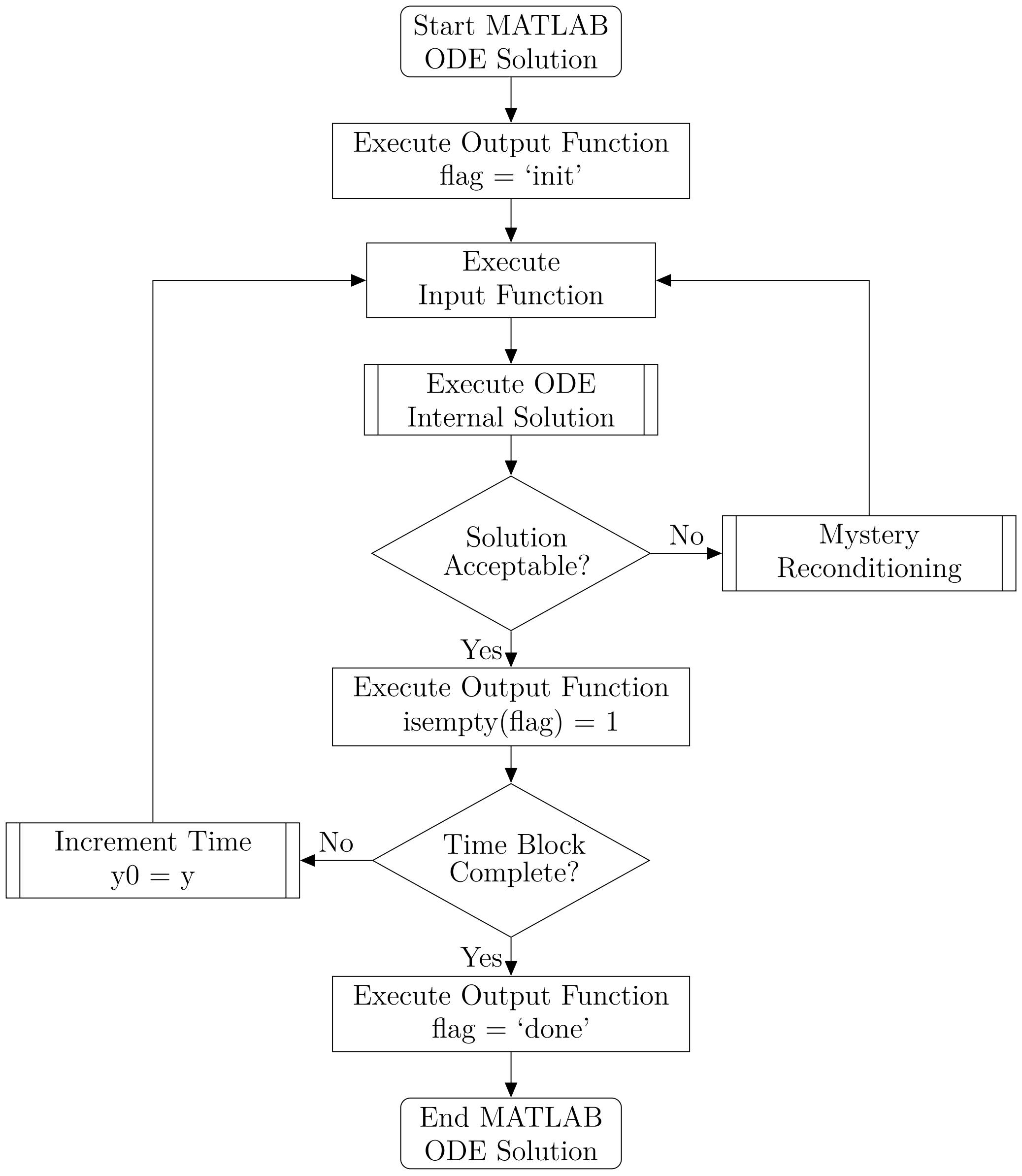
```
%% solver_con format
% A cell with a solver method in each row corresponding to the specified
% 'time blocks' defined in sw_con
%
% Valid solver names:
% huens - Fixed time step default to PST
% ode113 - works well during transients, consistent # of slns, time step stays relatively
↪  small
% ode15s - large number of slns during init, time step increases to reasonable size
% ode23 - realtively consisten # 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.

Theoretically, a user would only have to add a `solver_con` to a data file to use variable time step methods. If one is not specified, Huen's method is used for all time blocks (i.e. default PST behavior).

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.


Start MATLAB ODE Solution
Execute Output Function flag = 'init'
Execute Input Function
Execute ODE Internal Solution
Solution Acceptable?
No
Mystery 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

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

vtsInputFcn

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 g.vts.dxVec, and returns values
%
%   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 )

%% call handleStDx with flag==1 to update global dxVec
handleStDx(g.vts.dataN , [], 1) % update g.vts.dxVec (solution vector not needed)

dxVec = g.vts.dxVec; % return for ODE fcn requirements

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

g.vts.iter = g.vts.iter + 1; % increment iteration number
end % end vtsInputFcn
```

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)   % log initial conditions

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

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             g.vts.t_blockN, ...
379             g.vts.fts{g.vts.t_blockN}(1), g.vts.fts{g.vts.t_blockN}(end))
380 
381         % add fixed time vector to system time vector
382         nSteps = length(g.vts.fts{g.vts.t_blockN});
383         g.sys.t(g.vts.dataN:g.vts.dataN+nSteps-1) = g.vts.fts{g.vts.t_blockN};
384 
385         % account for pretictor last step time check
386         g.sys.t(g.vts.dataN+nSteps) = g.sys.t(g.vts.dataN+nSteps-1)+
            ↪  g.sys.sw_con(g.vts.t_blockN,7);
387 
388         for fStep = 1:nSteps
389             k = g.vts.dataN;
390             j = k+1;
391 
392             % display k and t at every first, last, and 50th step
393             if ( mod(k,50)==0 ) || fStep == 1 || fStep == nSteps
394                 fprintf('*** k = %5d, \tt(k) = %7.4f\n',k,g.sys.t(k)) % DEBUG
395             end
396 
397             %% Time step start
398             initStep(k)
399 
400             %% Predictor Solution =========================================
401             networkSolutionVTS(k, g.sys.t(g.vts.dataN))
402             monitorSolution(k);
403             dynamicSolution(k)
404             dcSolution(k)
```

```
            predictorIntegration(k, j, g.k.h_sol)

            %% Corrector Solution =========================================
            networkSolutionVTS(j, g.sys.t(g.vts.dataN+1))
            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

            % index handling
            g.vts.dataN = g.vts.dataN + 1;
            g.vts.tot_iter = g.vts.tot_iter  + 2;
            g.vts.slns(g.vts.dataN) = 2;
        end
        handleStDx(j, [], 3) % update g.vts.stVec to initial conditions of states
        handleStDx(k, [], 1) % update g.vts.dxVec to initial conditions of derivatives
        handleNetworkSln(k, 1) % update saved network solution

    else % use given variable method
        fprintf('*** Using %s integration method for time block %d\n*** t=[%7.4f, %7.4f]\n',
        ↪  ...
            odeName, g.vts.t_blockN, ...
            g.vts.t_block(g.vts.t_blockN, 1), g.vts.t_block(g.vts.t_blockN, 2))
        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(g.vts.t_blockN,:), g.vts.stVec , options);
        % feval used for now, could be replaced with if statements.
    end

end% end simulation loop
```

Functions that Enable 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 collected 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 these functions

correctorIntegration

As shown in the code except below, the `correctorIntegration` function performs the corrector integration step of the simulation loop to calculate the next value of integrated states. The executed code was taken directly from `s_simu`. The inputs to the function are the same variables used in `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
```

dcSolution

The portion of `s_simu` that integrates DC values at 10 times the rate of the normal time step were 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.

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`.

```
function dynamicSolution(k)
% DYNAMICSOLUTION Performs the dynamic solution for index k
```

handleNetworkSln

The `handleNetworkSln` function was created to store and restore the calculated values that are then 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. This function takes a data index `k` and an operation `flag` as inputs. The operation of the function is described in the code excerpt below.

```
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 solution vector to network globals data index k
```

handleStDx

The `handleStDx` function was created to perform the required state and derivative handling to enable the use internal MATLAB ODE solvers. The general function 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 populated 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).

While these operations were predicted during conceptual modeling of the function, a third operation that collects states into vector to use as initial conditions was somehow over looked. However, as all these operations have a similar form, the operation flag 3 was added to perform the state collection task without much fuss.

As most original PST globals follow the same structure, new models (such as AGC and pwrmod/ivmmmod) use a slightly different structure and must be handled in a slightly different way. As of this writing AGC and pwrmod has been added and it seems that adding new functionality to `handleStDx` is very possible and fairly straight forward.

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-matricies used for fault simulation and the calling of the dynamic models with the input flag set to 0.

initStep
Code from `s_simu` that was performed at the begining of each solution step was collected into `initStep`. It seems mostly 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.

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 have to be modified slightly.

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
```

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 caluculations are mostly based on complex voltages that are calculated during the network solution.

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

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 events and that is no longer a reality.

predictorIntegration

The `predictorIntegration` function is very similar to the `correctorIntegration` function, but performs the first step in Huen's method (instead of the 2nd step).

```
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
```

It should be noted that the two 'Integration' functions write to the same `j` state value index. Additionally, the `h_sol` value is generated in `i_simu` from the index of `ks` referencing a `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.

s_simu_BatchTestF

This 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 seems to work, this script will probably go away as it's usefulness seems minor.

s_simu_BatchVTS

This 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. This script performs optional VTS simulation and is slated to replace `s_simu` once PST SETO becomes PST 4.0.

standAlonePlot

The `standAlonePlot` function is the cleaned up plotting routine based on user input from the end `s_simu`. It is called from `s_simu_BatchVTS` if stand alone mode is detected.

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.

```
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)
%
%   Input:
%   k - data index
```

vtsInputFcn and vtsOutputFcn

These functions were described earlier.