

## 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 - relatively consistent # of required slns, timestep doesn't get very large
% ode23s - many iterations per step - not efficient...
% ode23t - occasionally hundreds of iterations, sometimes not... decent
% ode23tb - similar to 23t, sometimes more large solution counts

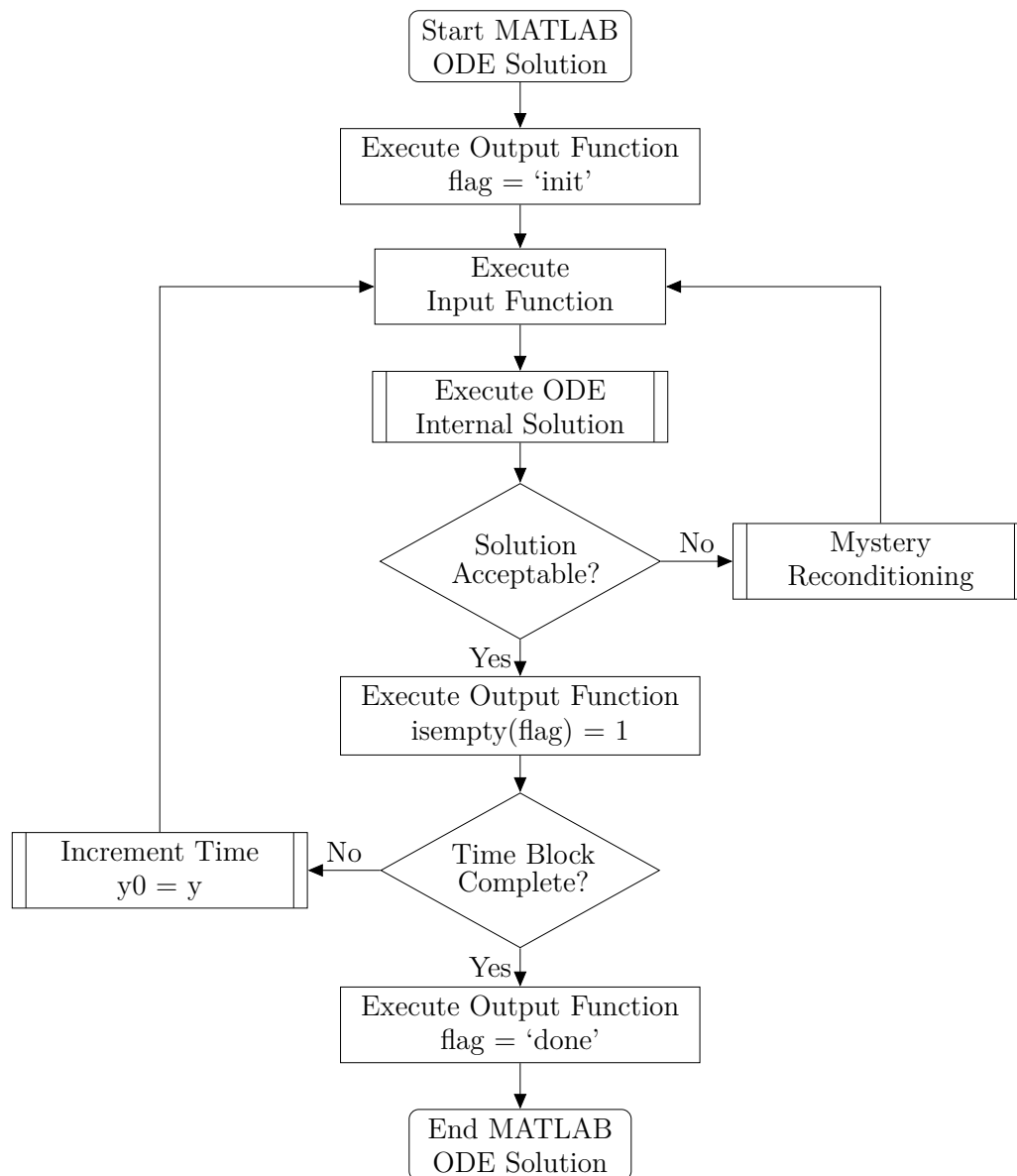
solver_con = { ...
    'huens'; % pre fault - fault
    'huens'; % fault - post fault 1
    'huens'; % post fault 1 - post fault 2
    'huens'; % post fault 2 - sw_con row 5
    'huens'; % sw_con row 5 - sw_con row 6
    'ode23t'; % sw_con row 6 - sw_con row 7 (end)
};
```

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.



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
  
```

## vtInputFcn

The slightly abbreviated input function is shown below.

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

## mtsOutputFcn

The slightly abbreviated output function is shown below.

```
function status = mtsOutputFcn(t,y,flag)
% MTSOUTPUTFCN 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.mts.dataN ,2) % may cause issues with DC.

    monitorSolution(g.mts.dataN); % Perform Line Monitoring and Area Calculations

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

    % after each successful integration step by ODE solver:
    g.mts.dataN = g.mts.dataN+1; % increment logged data index 'dataN'
    g.sys.t(g.mts.dataN) = t; % log step time
    g.mts.stVec = y; % update state vector
    handleStDx(g.mts.dataN, y, 2) % place new solution results into associated globals

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

elseif flag(1) == 'i'
    % init solver for new time block
    g.sys.t(g.mts.dataN) = t(1); % log step time
    handleStDx(g.mts.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,
378             ↪ %7.4f]\n', ...
379             g.vts.t_blockN, ...
380             g.vts.fts{g.vts.t_blockN}(1), g.vts.fts{g.vts.t_blockN}(end))
381
382         % add fixed time vector to system time vector
383         nSteps = length(g.vts.fts{g.vts.t_blockN});
384         g.sys.t(g.vts.dataN:g.vts.dataN+nSteps-1) = g.vts.fts{g.vts.t_blockN};
385
386         % account for predictor last step time check
387         g.sys.t(g.vts.dataN+nSteps) = g.sys.t(g.vts.dataN+nSteps-1)+
388             ↪ g.sys.sw_con(g.vts.t_blockN,7);
389
390     for fStep = 1:nSteps
391         k = g.vts.dataN;
392         j = k+1;
393
394         % display k and t at every first, last, and 50th step
395         if ( mod(k,50)==0 ) || fStep == 1 || fStep == nSteps
396             fprintf('*** k = %5d, \tt(k) = %7.4f\n',k,g.sys.t(k)) % DEBUG
397         end
398
399         %% Time step start
400         initStep(k)
401
402         %% Predictor Solution =====
403         networkSolutionVTS(k, g.sys.t(g.vts.dataN))
404         monitorSolution(k);
405         dynamicSolution(k)
406         dcSolution(k)
```

```
405     predictorIntegration(k, j, g.k.h_sol)
406
407     %% Corrector Solution =====
408     networkSolutionVTS(j, g.sys.t(g.vts.dataN+1))
409     dynamicSolution(j)
410     dcSolution(j)
411     correctorIntegration(k, j, g.k.h_sol)
412
413     % most recent network solution based on completely calculated states is k
414     monitorSolution(k);
415     %% Live plot call
416     if g.sys.livePlotFlag
417         livePlot(k)
418     end
419
420     % index handling
421     g.vts.dataN = g.vts.dataN + 1;
422     g.vts.tot_iter = g.vts.tot_iter + 2;
423     g.vts.slns(g.vts.dataN) = 2;
424 end
425 handleStDx(j, [], 3) % update g.vts.stVec to initial conditions of states
426 handleStDx(k, [], 1) % update g.vts.dxdVec to initial conditions of derivatives
427 handleNetworkSln(k, 1) % update saved network solution
428
429 else % use given variable method
430     fprintf('*** Using %s integration method for time block %d\n*** t=[%7.4f, %7.4f]\n',
431         ↪ ...
432         odeName, g.vts.t_blockN, ...
433         g.vts.t_block(g.vts.t_blockN, 1), g.vts.t_block(g.vts.t_blockN, 2))
434     feval(odeName, inputFcn, g.vts.t_block(simTblock,:), g.vts.stVec, options);
435
436     % Alternative example of using actual function name:
437     %ode113(inputFcn, g.vts.t_block(g.vts.t_blockN,:), g.vts.stVec, options);
438     % feval used for now, could be replaced with if statements.
439 end
440 end% end simulation loop
```

Place holders for function explanations / real basic explanations of *some* VTS functions.

### **initTblocks**

Analyzes `sw_con` and `solver_con` to create appropriate *time blocks* that are used in simulation.

### **handleStDx**

Function to handle the collecting, indexing, and updating ALL states and derivatives. It uses dynamic field names and seems like a pretty slick solution to a very annoying problem.

### **handleNetworkSln**

Saves the initial network solution and ensures it is restored after a variable step solution where **many** network solutions may overwrite the first (and correct) solution.

### **networkSolutionVTS**

A function that solves the system network at the passed in data index. 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.