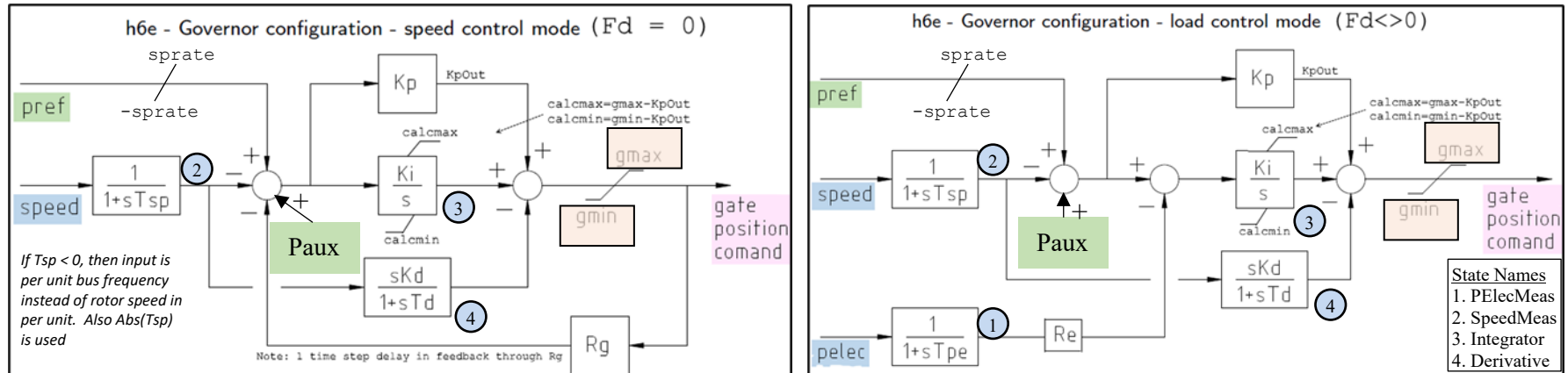


Governor H6E

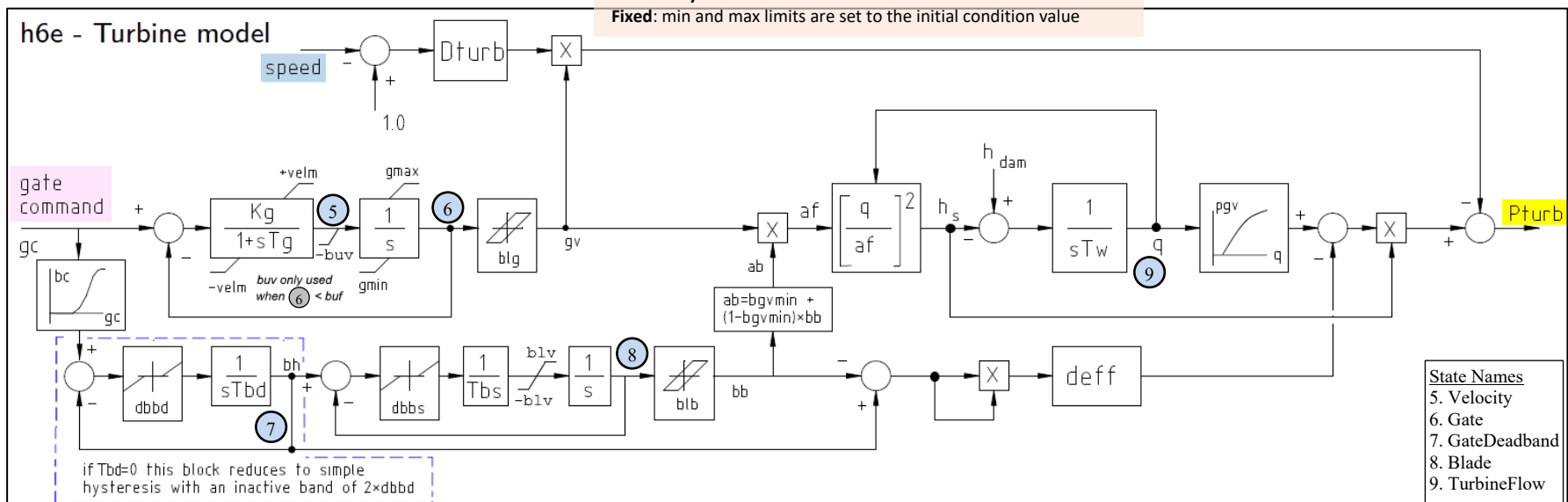
Governor H6E
Hydro Turbine with American Governor Company Controller

Governor Configuration is determined by the integer flag parameter Fd which is either 0 or not zero.



Governor Response Limits setting modifies these limits

Down Only:	max limit is set to initial condition value
Fixed:	min and max limits are set to the initial condition value



Model Notes (Notes provided by John Undrill in memo on June 13, 2016)

- (a) Per unit parameters are on the base of turbine MW capability as given by the parameter T_{rate} . Because of this, the input to the model of pelec must be converted from the machine base to the turbine base by multiplying by $\frac{MachineMVABase}{T_{rate}}$. Similarly when used with other model the output of this model must be converted by the the MachineMVABase by multiplying by $\frac{T_{rate}}{MachineMVABase}$.
- (b) The gates travel over a range of 1.0 per unit from fully closed to fully opened. The gates are at a position greater than zero when the turbine real power output is zero. Gmax and Gmin are operating limits.
- (c) Tpe, Td, Tg must be greater than or equal to zero. Absolute value of Tsp must be greater than zero.
- (d) Dturb has the dimensions power/speed
- (e) Details of the deadband and backlash functions are shown in the Turbine Model block diagram on on previous page.
- (f) The turbine is described by two curves as follows:
 pgv -- Turbine power in per unit of tbase versus gate servomotor stroke.
 bgv -- Blade servo stroke versus gate servomotor stroke.
- (g) Each curve is specified by ten points (numbered 0-9) corresponding to ten values of gate servomotor stroke which are given as gv0 - gv9. The same set of servo stroke values is used for both curves. Figure 4 shows the form of the turbine curves. The value of gv0 should normally be greater than zero and the value of gv9 should normally be 1.0. The value of pgv9 need not be equal to unity.
- (h) For a Kaplan turbine the blade flow area curve, bgv, must pass through (0,0) and (1,1). Figure 5 shows typical curves for a Kaplan turbine.
- (i) For a Francis turbine all the points on the blade flow area curve, bgv, curve must be at be (g,1). Figure 6 shows typical curves for a Francis turbine
- (j) The parameter bgvmin specifies the blade flow area when the blade servo is at zero stroke. The value of bgvmin is in per unit of the blade flow area when the blades are fully open.
- (k) The power absorbed by the turbine when running at rated speed with the gates fully closed is given by the parameter, pgc.
- (l) The adjustment factor, deff, can be used to account for reduction of Kaplan turbine power when the blade angle is at an off-nominal angle. The adjustment is illustrated by Figure 7 and is given by
 (adjusted pwr) = (on-angle pwr) - deff * (ideal blade stroke - actual blade stroke)**2
- (m) The speed-load reference of the governor is initialized to correspond to the initial turbine output. For initialization, linkages with backlash and deadband are taken to be at the midpoint of the backlash/deadband.
- (n) When the curves describing the relationship between gate servo stroke and power are set up from test data, they must be corrected to account for generator losses and for the difference between the operating head and rated head, as follows (where M_{base} is the generator MVA base and T_{rate} is the turbine MW base).
- Increase the measured generator power by the generator stator losses

$$P_g = P_{meas} + R_a \left(\frac{P_{meas}}{M_{base}} \right)^2$$
 - State adjusted measured power in per unit of turbine power base, T_{rate}

$$P_e = \frac{P_g}{T_{rate}}$$
 - Correct power to account for the head in the test condition

$$P_{gv} = \frac{P_e}{h_{dam}^{1.5}}$$
- (o) The 'speed' input to the governor is the turbine shaft speed if the time constant, tsp>=0, or frequency of the generator terminal voltage if the time constant, tsp<0. The speed/frequency transducer time constant is the absolute value of tsp.
- (p) The parameters *buf* and *buv* describe the effect of the closing rate buffer on the gate servomotor. The buffer takes effect when the gate servomotor stroke is less than *buf* and limits the gate closing rate to *buv* per unit per second.
- (q) All adjustments of the speed-load reference are applied at the fixed rate specified by the parameter, *sprate*. The reference used by the governor is always ramped to a newly specified value of pref at the fixed rate of $\pm sprate$.

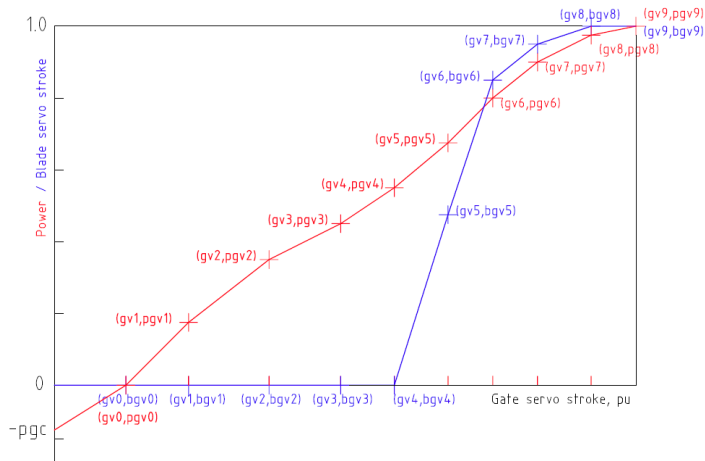


Figure 4: Turbine power-gate and blade-gate curves
Red - Power/gate (pgv/ggv)
Blue - Blade/gate (bgv/ggv)

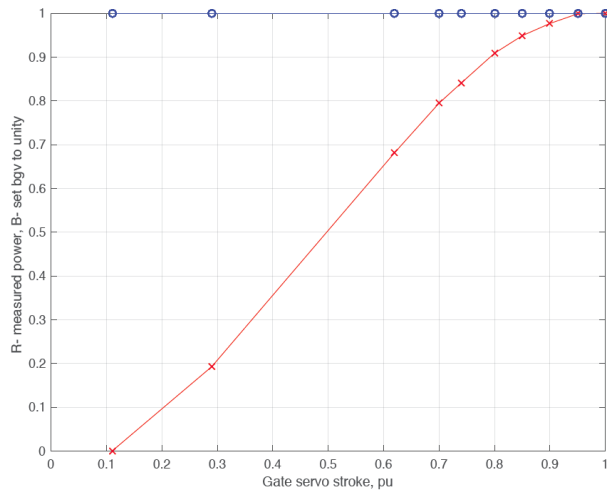


Figure 6: Typical power-gate and blade-gate curves for Francis turbine
Red - Power/gate (pgv/gv)
Blue - Blade/gate (bgv/gv)

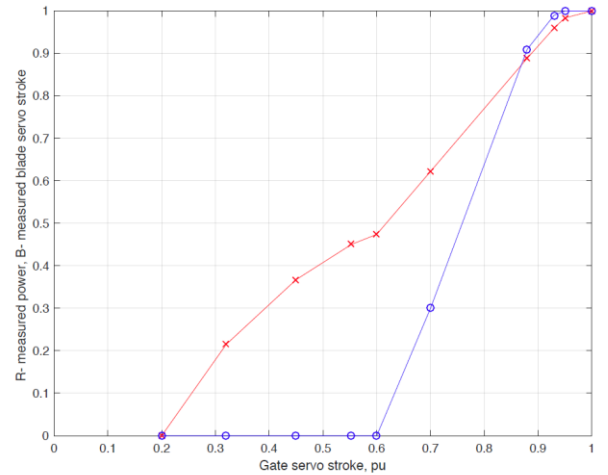


Figure 5: Typical power-gate and blade-gate curves for Kaplan turbine
Red - Power/gate (pgv/gv)
Blue - Blade/gate (bgv/gv)

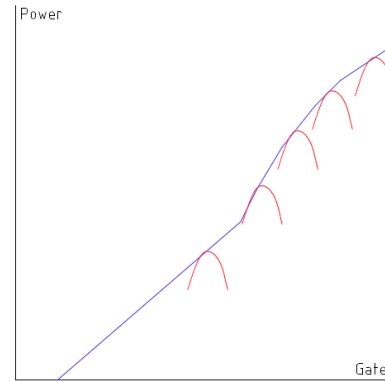


Figure 7: Variation of turbine efficiency with off-cam blade angle
Blue - Power with blade servo at programmed position
Red - Power deviation as blade servo position deviates from programmed position

PowerWorld Implementation in Pseudo-Code

User Input Parameters for Model

```
ffd, ftrate, fre, frg, finvtpe, finvtsp, fkp, fki, fkd,  
finvtd, fvelm, fgmax, fgmin, fbuf, fbuv, fkg, finvtg,  
fblg, fdbbd, finvtbd, fblb, fdbbs, finvtbs, fbgvmin,  
fblv, fdturb, fpgc, fdeff, fhdam, finvtw, fsprate : single;  
// Note various parameters such as finvtpe are storing 1/tpe  
fBackLash_blb, fBackLash_blg : BackLash Object; // see below  
fP_Q, fB_GV : Nonlinear Lookup Tables
```

These tables are created from the input parameters GV0...GV9, PGV0...PGV9, and BGV0...BGV9.
See section below for details of how these are calculated.

Cached Values with model

These variables are available inside all functions.

```
cache_Spref, cache_LastTimeStepGateCommand, cache_GateCommand, cache_Hscroll,  
cache_BC, cache_GV, cache_BB, cache_BH, cache_AB, cache_AF : double;
```

State and Derivative Value Syntax

```
Derivative[INDEX] : Derivative of state INDEX for model  
StateVector[INDEX] : Value of state INDEX for model  
IgnoreStateVector[INDEX] : boolean indicating if state INDEX is ignored
```

Nonlinear Lookup Curves (GV0...GV9, PGV0...PGV9, BGV0...BGV9)

After reading these input parameters three lookup tables are created

```
// force fbgvmin to between 0 and 1, but not equal to those numbers  
if fbgvmin < 0.00001 then fbgvmin := 0.00001  
else if fbgvmin > 0.99999 then fbgvmin := 0.99999;  
for i := 0 to 9 do begin  
    // Calculate steady state Q for particular gate value  
    QGV[i] := GV[i]*( fbgvmin + (1-fbgvmin) * BGV[i]);  
End;  
// Create lookup function and add first point.  
// The FALSE in constructor means to not extrapolate past the start and end point  
fP_Q := PiecewiseLinearFunctionObject.Create(QGV[0],PGV[0],FALSE);  
fB_GV := PiecewiseLinearFunctionObject.Create( GV[0],BGV[0],FALSE);  
For i := 1 to 9 Do Begin  
    if (GV[i] > 0) then begin  
        fP_Q.AddPointInOrder(QGV[i],PGV[i]); // Power vs Q  
        fB_GV.AddPointInOrder(GV[i],BGV[i]); // B vs Gate  
    end;  
End;  
// PiecewiseLinearFunctionObject just encapsulates a lookup function.  
// Method X(index) returns the X value of the X,Y part at a particular index  
// Method Y(index) returns the Y value of the X,Y part at a particular index  
// Method XtoY takes an input X value and calculates the output Y// Method YtoX takes an output Y  
value and calculates the input X
```

BackLash Object

```
fDeadband : Double;  
fLastOutput: Double; // Tells the last output value  
Public  
    Constructor CreateBackLash(tDeadband,tInitialOutput : Double);  
    Function Output(anInput : Double) : Double;  
  
BackLash.CreateBackLash(tDeadband,tInitialOutput : Double);  
Begin  
    fDeadBand := tDeadBand;  
    If fDeadband < 0 Then fDeadband := 0; // Must be non-negative  
    fLastOutput := tInitialOutput;  
End;
```

```

Function BackLash.Output(anInput : Double) : Double;
Begin
  If (anInput >= fLastOutput-fDeadband) and (anInput <= fLastOutput+fDeadBand) Then begin
    Result := fLastOutput;
  End
Else Begin
  If anInput > fLastOutput+fDeadband
  Then Result := anInput-fDeadBand // Going up, subtract off deadband
  Else Result := anInput+fDeadBand; // Going down, add deadband
  fLastOutput := Result;
End;
End;

```

Start of Each TimeStep

```

Begin
  cache_LastTimeStepGateCommand := cache_GateCommand;

  // Update the cache_Spref value using a rate limiter
  if Pref > cache_Spref then begin
    cache_spref := cache_spref + fsprate * timeStepSeconds;
    if cache_spref > Pref then cache_spref := Pref;
  end
  else if Pref < cache_Spref then begin
    cache_spref := cache_spref - fsprate * timeStepSeconds;
    if cache_spref < Pref then cache_spref := Pref;
  end;
end;

```

function to return the Pmech to feed to machine model

```

var local_Q, local_MinQ : double;
begin
  local_Q := StateVector[9];
  // hydraulic power
  if fP_Q.Count > 0 then begin
    local_MinQ := fP_Q.X(1);
    if (local_Q < local_MinQ) and (local_MinQ > 0)
    then result := cache_Hscroll * (fpgc * (local_Q - local_MinQ)/local_MinQ)
    else result := cache_Hscroll * (fP_Q.XtoY( local_Q ) - fdeff*sqr(cache_bh - cache_bb));
  end
  else begin
    result := cache_Hscroll * (local_Q - fdeff*sqr(cache_bh - cache_bb));
  end;
  result := result - deltaWPU * fDTurb * cache_GV;
  // put turbine power on appropriate per unit base
  result := result/GenObject.MVABase*fTrate;
end;

```

InitializeForIntegration Function

```

Var local_pq,
    local_gv,
    local_q,
    local_af : Double; // local values for initialization
Local_Pmech : double;
Local_PElec : double;
Begin
  Local_Pmech := GenObject.PMechObjectInitInternal;
  Local_Pmech := Local_Pmech*GenObject.MVABase/fTrate;
  Local_PElec := PElec;
  Local_PElec := Local_PElec*GenObject.MVABase/fTrate;

  local_pq := Local_Pmech/(1.0*fHdam);
  if local_pq > fP_Q.LastY then begin // added initial limit violation check in February 2022
    fHdam := local_Pmech / fP_Q.LastY; // added some log messages
    local_pq := Local_Pmech/(1.0*fHdam);
  end;
  local_q := fP_Q.YtoX(local_pq);
  local_af := local_q/sqrt(fHdam);

  local_afmax := fgmax* (fbgvmin + (1-fbgvmin)*fB_GV.XtoY(fgmax) );
  if (local_af > local_afmax)
  and LOCAL_BackSolveForQ(local_afmax, local_Pmech, local_q)
  then begin // added initial limit violation check in September 2022
    local_gv := fgmax;
    local_af := local_afmax;
    fHdam := sqr(local_q/local_afmax); // added some log messages
  end
  else begin

```

```

local_gv := LOCAL_BackSolveForGate(local_af); // see on following pages
end;
// Modified fgmin and fgmax if Governor Response Limits flag ("base load") is used.
cache_GateCommand := local_gv;
cache_LastTimeStepGateCommand := local_gv;
cache_BC := fB_GV.XtoY(local_gv);
cache_GV := local_gv;
cache_BB := cache_BC;
cache_BH := cache_BC;
cache_AB := (fbgvmin + (1-fbgvmin)*cache_BB);
cache_AF := cache_AB * local_gv;
cache_Hscroll := GOVERNOR_DivideAndSquareForHead(local_q, cache_af, fHdam);

If fblb <> 0 Then fBackLash_blb := TTxBacklash.CreateBackLash(self, fblb, cache_BB);
If fblg <> 0 Then fBackLash_blg := TTxBacklash.CreateBackLash(self, fblg, local_GV);

StateVector[1] := Local_PElec;
StateVector[2] := 1.0; // speed
StateVector[4] := 0.0; // model input to derivative state as (speed - 1)
StateVector[5] := 0.0;
StateVector[6] := local_gv;
StateVector[7] := cache_BC;
StateVector[8] := cache_BC;
StateVector[9] := local_q;
// PIDInput = Pref - Speed - Relec*PElec - Rg*Gate
// PIDInput must be zero initially because we require Ki > 0
// 1.0 here represents the initial speed
if ffd = 0 then begin
    Pref := 1.0 + local_gv*fRg;
    StateVector[3] := local_gv;
end
else begin
    Pref := 1.0 + local_PElec*fRe;
    StateVector[3] := local_gv - fKp*(Pref - 1.0);
end;
cache_Spref := Pref;

// Set ignored states (this is a boolean vector PowerWorld uses to signify ignored states)
if fInvtpe = 0 then IgnoreStateVector[1] := True;
if fInvtsp = 0 then IgnoreStateVector[2] := True;
// 23 is the Ki integrator
if fInvTd = 0 then IgnoreStateVector[4] := True;
if fInvtg = 0 then IgnoreStateVector[5] := True;
// 56 is the gate integrator
if fInvtdb = 0 then IgnoreStateVector[7] := True;
if fInvtdbs = 0 then IgnoreStateVector[8] := True;
// 89 is the water integrator
end;

function LOCAL_BackSolveForGate(local_af : double) : double;
var i : integer;
    local_gk, local_gm, local_bk, local_bm,
    local_slope,
    local_A, local_B, local_C : double;
begin
    Result := 0;
    i := fB_GV.Count;
    if i = 0 then begin // just assume Bgv = 1.0 always, which means that
        result := local_af;
        exit;
    end;
    if i = 1 then begin
        local_bk := fB_GV.Y(1);
        result := local_af / (fbgvmin + local_bk*(1-fbgvmin));
        exit;
    end;
    while i >= 0 do begin
        local_gk := fB_GV.X(i);
        local_bk := fB_GV.Y(i);
        if local_af < local_gk*(fbgvmin + local_bk*(1-fbgvmin)) then begin // just keep going
            if i = 1 then begin // However if i = 1, this is degenerate case!
                result := local_af / (fbgvmin + local_bk*(1-fbgvmin));
                exit;
            end;
        end;
    end;
    else begin
        if (i = fB_GV.Count) then begin
            result := local_af / (fbgvmin + local_bk*(1-fbgvmin)); // just use this last Bk value
            exit;
        end;
        else begin
            local_gm := fB_GV.X(i+1);
            local_bm := fB_GV.Y(i+1);
            local_slope := (local_bm - local_bk) / (local_gm - local_gk);

```

if abs(local_slope) > 1E-6 **then begin**

```
{
  we are trying to solve for gv given the value of af.
  af = gv * [ bgvmin + (1-bgvmin)*Bgv ]
  Solve this for Bgv to make it easier for next step and we have
    af/gv - bgvmin
    Bgv = -----
          1 - bgvmin
  If we're one of the line segments of the Bgv vs Gv curve then we have
  a line segment going from (gk, bk) sloping up to (gm, bm)
```

$$Bgv = bk + (gv-gk) * \frac{(bm-bk)}{(gm-gk)}$$

Now equate these two function for Bgv and solve for gv (Note | just grouping stuff)

$$0 = \frac{1}{gv} * \frac{-af}{(1-bgvmin)} + \frac{bgvmin}{(1-bgvmin)} + bk - gk * \frac{(bm-bk)}{(gm-gk)} + gv * \frac{(bm-bk)}{(gm-gk)}$$

Now, multiply through by gv and make terms for "A, B, C" for quadratic function

$$0 = C + B*gv + C*sqr(gv)$$

local_C := -local_AF/(1-fbgvmin);

local_B := fbgvmin/(1-fbgvmin) + local_bk - local_gk*local_slope;

local_A := local_Slope;

result := (- local_B + sqrt(sqr(local_B) - 4*local_A*local_C))/ (2*local_A);

exit;

end

else begin

```
{
  We're on a slope with constant Bk, so back to af = gv * [ bgvmin + (1-bgvmin)*Bgv ]
  Which can be solved as gv = af/bgvmin + bk*(1-bgvmin)
}
```

result := local_af/(fbgvmin + local_bk*(1-fbgvmin));

exit;

end;

end;

end;

i := i - 1;

end;

end;

function LOCAL_BackSolveForQ(local_afmax, local_Pmech : **double**; var local_q : **double**) : **boolean**;

{

This function was added in September 2022 to handle situation where the maximum gate Value is not enough to achieve the af value necessary to get Pmech desired.

We will again solve this by increasing the Hdam value

Solving Equations: $\text{sqr}(q/\text{afmax}) = \text{Hdam}$ and $P(q)*\text{Hdam} = \text{Pmech}$

Substitute first equation into the second one --> $P(q)*\text{sqr}(q/\text{afmax}) = \text{Pmech}$

$$P(q)*\text{sqr}(q) = \text{Pmech}*\text{sqr}(\text{afmax})$$

$$P(q)*\text{sqr}(q) - \text{Pmech}*\text{sqr}(\text{afmax}) = 0$$

If $P(q) = q$ (meaning we have piecewise linear function at all this degenerate solution is $q * \text{sqr}(q/\text{afmax}) = \text{Pmech}$ --> $q = (\text{Pmech}*\text{sqr}(\text{afmax}))^{1/3}$)

Otherwise, we have only 1 point (or we end up on a horizontal segment), then this mean that $P(q)$ is a constant

If $P(q) = Pk$ (1 point, or we have a zero slope on this segment)

$$Pk * \text{sqr}(q/\text{afmax}) = \text{Pmech} \text{ --> } q = \text{sqr}(\text{Pmech}/Pk)*\text{afmax}$$

Otherwise we're looking at somewhere between 2 points

Assume we're on line segment between (qk, Pk) and (qm, Pm), so we can write

$$P(q) = Pk + (q-qk) * \frac{(Pm-Pk)}{(qm-qk)}$$

$$P(q) = q * \frac{(Pm-Pk)}{(qm-qk)} + Pk - qk * \frac{(Pm-Pk)}{(qm-qk)}$$

Define Slope = $\frac{(Pm-Pk)}{(qm-qk)}$

$$P(q) = q*Slope + Pk - qk*Slope$$

$$P(q)*\text{sqr}(q) - \text{Pmech}*\text{sqr}(\text{afmax}) = 0$$

$$(q*slope + Pk - qk*slope)*\text{sqr}(q) - \text{Pmech}*\text{sqr}(\text{afmax}) = 0$$

$$[Slope]*q^3 + [Pk - qk*Slope]*q^2 + [- \text{Pmech}*\text{sqr}(\text{afmax})] = 0;$$

$$\text{Slope} = (Pm-Pk)/(qm-qk)$$

A = Slope // A > 0 (bad user input otherwise)

B = Pk - qk*Slope // B can be positive or negative

D = -Pmech*sqr(afmax) // D < 0 (Pmech can't be negative!)

By assuming Slope > 0 this means that we have 1 sign change when looking through the Polynomial coefficients, so by Descartes sign rule, we have exactly 1 positive real root

Degenerate situation when no P vs Q curve exists. In this situation $P(q) = q$ always (multiplier is just 1.00)

```
P(q)*sqr(q) - Pmech*sqr(afmax) = 0
q^3 = Pmech*sqr(afmax)
q = ( Pmech*sqr(afmax) ) ^ (1/3) // cube root
```

Degenerate situation when Slope = 0, then this is

```
A=0; B=Pk; C=0; D=-Pmech*sqr(afmax)
Pk*sqr(q) = Pmech*sqr(afmax)
```

Solution is then

```
q = sqrt(Pmech/Pk)*afmax
```

```
}
```

```
procedure LOCAL_HandlePkConstant(local_Pk: double);
```

```
begin
```

```
  if local_Pk < 0.0001 then local_Pk := 0.0001; // this should never happen!
```

```
  Result := true;
```

```
  Local_q := sqrt(local_Pmech/local_Pk) * local_afmax;
```

```
end;
```

```
var i : integer;
```

```
  local_qk, local_qm, local_Pk, local_Pm,
```

```
  _A, _B, _D, local_slope, _Pmech_sqrAfmax : double;
```

```
begin
```

```
  result := false;
```

```
  _Pmech_sqrAfmax := local_Pmech*sqr(local_afmax);
```

```
  i := fP_Q.Count;
```

```
  if i = 0 then begin
```

```
    result := true;
```

```
    local_q := GOVERNOR_CubeRoot(_Pmech_sqrAfmax);
```

```
    exit;
```

```
  end;
```

```
  if i = 1 then begin
```

```
    local_Pk := fP_Q.Y(1);
```

```
    LOCAL_HandlePkConstant(local_Pk);
```

```
    exit;
```

```
  end;
```

```
  Result := 0;
```

```
  while i >= 0 do begin
```

```
    local_qk := fP_Q.X(i);
```

```
    local_Pk := fP_Q.Y(i);
```

```
    if sqr(local_qk)*local_Pk > _Pmech_sqrAfmax then begin
```

```
      // just keep going normally here,
```

```
      if i = 1 then begin // However if i = 1, this is silly case!
```

```
        LOCAL_HandlePkConstant(local_Pk);
```

```
        exit;
```

```
      end;
```

```
    end
```

```
  else begin
```

```
    if (i = fP_Q.Count) then begin
```

```
      LOCAL_HandlePkConstant(local_Pk);
```

```
    end
```

```
  else begin
```

```
    local_qm := fP_Q.X(i+1);
```

```
    local_Pm := fP_Q.Y(i+1);
```

```
    local_slope := (local_Pm - local_Pk)/(local_qm - local_qk);
```

```
    if abs(local_slope) < 1E-6 then begin
```

```
      LOCAL_HandlePkConstant(local_Pk);
```

```
    end
```

```
  else if local_slope > 0 then begin // Solving  $A*q^3 + B*q^2 + D = 0$ 
```

```
    _A := local_slope; // must be positive
```

```
    _B := local_Pk - local_qk*local_slope; // Could be pos or neg
```

```
    _D := -_Pmech_sqrAfmax; // must be negative
```

```
    // Because  $A > 0$  and  $D < 0$ , then there is exactly 1 sign change
```

```
    // By Descartes Rule of Signs, this means there is ONE positive answer
```

```
    result := true;
```

```
    Local_q := SolveCubicAndReturnPositiveRealAnswer(_A, _B, 0, _D);
```

```
  end;
```

```
end;
```

```
Exit; // exit no matter what here. If a local_slope < 0, don't do anything
```

```
end;
```

```
i := i - 1;
```

```
end;
```

```
end;
```

Derivative Function

```
procedure LOCAL_HandleNonWakeup(theStateOffset : integer; theDeriv, themax, themin : double);
var local_s : double;
begin
    local_s := StateVector[theStateOffset];
    if ( local_s >  themax) then StateVector[theStateOffset] := themax;
    if ( local_s <  themin) then StateVector[theStateOffset] := themin;
    Derivative[theStateOffset] := theDeriv;
    if ( local_s >= themax) and ( Derivative[theStateOffset] > 0 ) then
        Derivative[theStateOffset] := 0;
    if ( local_s <= themin) and ( Derivative[theStateOffset] < 0 ) then
        Derivative[theStateOffset] := 0;
end;

var local_Pelec,
    local_Speed,
    local_PropIn,
    local_IntIn,
    local_SpeedError : double;
    local_Deriv : double;
Begin
    local_Pelec := PElec*GenObject.MVABase/fTrate;
    if finvTsp < 0 then local_Speed := GenObject.GetBusFreqPU // bus frequency
    else local_Speed := actualWPU; // rotor speed
    If IgnoreStateVector[1] Then StateVector[1] := local_Pelec;
    If IgnoreStateVector[2] Then StateVector[2] := local_Speed;
    If IgnoreStateVector[4] Then StateVector[4] := StateVector[1] - 1.0;
    Derivative[2] := abs(finvTsp)*(local_Speed - StateVector[2]);
    Derivative[4] := finvTd*(StateVector[2] - 1.0 - StateVector[4]);

    local_SpeedError := cache_Spref - StateVector[2] + GenObject.Paux/GetTRate; // AAA 03/13/20
    if fFD = 0 then begin
        Derivative[1] := 0;
        // proportional control DOES include gate droop when FD = 0
        local_PropIn := local_SpeedError - fRg*cache_LastTimeStepGateCommand;
        local_IntIn := local_PropIn;
    end
    else begin
        Derivative[1] := finvTpe*(local_Pelec - StateVector[1]);
        // proportional control DOES NOT include Pelec droop term when FD = 1
        local_PropIn := local_SpeedError;
        local_IntIn := local_SpeedError - fRe*StateVector[1];
    end;
    LOCAL_HandleNonWakeup(3, fKi*local_IntIn, fgmax - local_PropIn*fKp, fgmin - local_PropIn*fKp);
    cache_GateCommand := fKp*local_PropIn + StateVector[3]; // proportional + Integrator
    // Derivative block effect
    If not IgnoreStateVector[4] Then begin
        cache_GateCommand := cache_GateCommand -
            fKd*finvTd*(StateVector[2] - 1.0 - StateVector[4]);
    end;
    if cache_GateCommand > fgmax then cache_GateCommand := fgmax;
    if cache_GateCommand < fgmin then cache_GateCommand := fgmin;

    local_Deriv := fInvTg*( fKg *(cache_GateCommand - StateVector[6]) - StateVector[5] );
    LOCAL_HandleNonWakeup(5, local_Deriv, fvelm, -fvelm);
    LOCAL_HandleNonWakeup(6, StateVector[5], fgmax, fgmin);

    // gate buffer
    if (StateVector[6] < fbuv) AND (Derivative[6] < -fbuv) then Derivative[6] := -fbuv;
    // handle gate backlash
    cache_gv := StateVector[6];
    if fBacklash_blg <> nil then cache_gv := fBacklash_blg.output( cache_gv );

    cache_BC := fB_GV.XtoY(cache_GateCommand);
    if ( fInvTbd > 0 ) then begin
        local_Deriv := cache_BC - StateVector[7];
        if ( local_Deriv > fdbbd ) then local_Deriv := local_Deriv - fdbbd
        else if ( local_Deriv < -fdbbd ) then local_Deriv := local_Deriv + fdbbd;
        Derivative[7] := finvTbd * local_Deriv;
        cache_bh := StateVector[7];
    end
    else begin
        StateVector[7] := cache_bc;
        Derivative[7] := 0;
    end;
```

```

// note this won't always update the bh value!
// That's fine, this is hysteresis. It only moves once the deviation is large enough
if (cache_bc - fdbbd) > cache_bh then cache_bh := cache_bc - fdbbd;
if (cache_bc + fdbbd) < cache_bh then cache_bh := cache_bc + fdbbd;
end;

if finvTbs > 0 then begin
  local_Deriv := cache_bh - StateVector[78];
  if ( local_Deriv > fdbbs ) then local_Deriv := local_Deriv - fdbbs
  else if ( local_Deriv < -fdbbs ) then local_Deriv := local_Deriv + fdbbs;
  local_Deriv := finvTbs * local_Deriv;
  if local_Deriv > +fblv then local_Deriv := +fblv
  else if local_Deriv < -fblv then local_Deriv := -fblv;
  Derivative[8] := local_Deriv;
end
else begin
  StateVector[8] := cache_bh;
  Derivative[8] := 0;
end;

cache_bb := StateVector[8];
if fBacklash_blb <> nil then cache_bb := fBacklash_blb.output( cache_bb );
cache_ab := fbgvmin + (1 - fbgvmin)*cache_bb;
cache_af := cache_ab * cache_gv;

// when cache_af value input to water dynamics is an extremely small number,
// the dynamics can become extremely fast because we divide by cache_af.
// Model starts entering a region we have 0/0 as Q value also goes to zero.
// Model these dynamics as algebraic and instantly change the state value to handle this.
// Otherwise numerical problems related to 0/0 will result
if cache_af < 0.005 then StateVector[9] := sqrt(fHdam)*cache_af;
cache_Hscroll := sqr(StateVector[9]/cache_af);

// Don't allow the water flow to drop below 0.0001 per unit (more avoid 0/0 situations)
LOCAL_HandleNonWindup(9, finvTw*( fHdam - cache_hscroll ), 1E10, 0.0001);
end;

```

Parameters:

trate	Turbine power base, MW
fd	Flag for proportional elec power signal; 0 = Speed control mode - droop is based on gate servo stroke; 1 = Load control mode - droop is based on electric power
re	Permanent droop for electrical power feedback, pu
rg	Permanent droop for gate position feedback, pu
tpe	Electric power feedback transducer time const, sec
tsp	Shaft speed transducer time const, sec
kp	Governor proportional gain
ki	Governor integral gain
kd	Governor derivative gain
td	Derivative gain filter time constant, sec
velm	Maximum gate actuator velocity, pu/sec
gmax	Maximum gate actuator stroke, pu
gmin	Minimum gate actuator stroke, pu
buf	Upper limit of gate buffer region, pu
buv	Max gate closing rate in buffer region, pu
kg	Pilot servovalve gain
tg	Pilot servovalve time constant, sec
blg	Backlash in ring linkage
dbbd	Deliberate sliding deadband in digital blade cam output
tbd	Time constant of sliding deliberate dead band, sec
blb	Backlash in blade adjustment linkage
dbbs	Mechanical deadband in blade servovalve/motor
tbs	Blade servo time constant, sec
bgvmin	Flow area factor of blades when at minimum position
blv	Maximum stroking rate of blade servomotor
dturb	Turbine speed sensitivity constant
pgc	Electric motoring power with gates fully closed, pu
deff	Off blade angle power decrease factor
hdam	Operating head, pu
tw	water inertia time constant, sec
sprate	speed-load setpoint adjustment rate, pu/sec
gv0	Abscissa value 0 for wicket gate and blade curves
gv1	Abscissa value 1 for wicket gate and blade curves
gv2	Abscissa value 2 for wicket gate and blade curves
gv3	Abscissa value 3 for wicket gate and blade curves
gv4	Abscissa value 4 for wicket gate and blade curves
gv5	Abscissa value 5 for wicket gate and blade curves
gv6	Abscissa value 6 for wicket gate and blade curves

gv7	Abscissa value 7 for wicket gate and blade curves
gv8	Abscissa value 8 for wicket gate and blade curves
gv9	Abscissa value 9 for wicket gate and blade curves
pgv0	Value 0 for Curve of wicket gate flow area
pgv1	Value 1 for Curve of wicket gate flow area
pgv2	Value 2 for Curve of wicket gate flow area
pgv3	Value 3 for Curve of wicket gate flow area
pgv4	Value 4 for Curve of wicket gate flow area
pgv5	Value 5 for Curve of wicket gate flow area
pgv6	Value 6 for Curve of wicket gate flow area
pgv7	Value 7 for Curve of wicket gate flow area
pgv8	Value 8 for Curve of wicket gate flow area
pgv9	Value 9 for Curve of wicket gate flow area
bgv0	Value 0 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv1	Value 1 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv2	Value 2 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv3	Value 3 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv4	Value 4 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv5	Value 5 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv6	Value 6 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv7	Value 7 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv8	Value 8 for Curve of Kaplan turbine blade flow area as function of gate servo stroke
bgv9	Value 9 for Curve of Kaplan turbine blade flow area as function of gate servo stroke