Development plan for the linearization about a rotational speed set-point in OpenFAST

E. Branlard, J. Jonkman, B. Jonkman

August 3, 2020

Introduction

This document describes the implementation of an algorithm to perform linearization about a rotational set-point in *OpenFAST*. Currently, the user needs to manually adjust operating parameters until the desired rotational speed is reached, or until the transients are eliminated, before the linearization can be performed. This procedure is here automatized using a simple iterative algorithm. The method supports simulations with zero rotational speed, a fixed rotational speed (generator off), or simulations with a variable speed (generator on). In the later case, the algorithm presented adjusts one of the following operating parameter to reach the target rotational speed: the blade pitch angle, the generator torque or the neutral yaw position of the turbine. A proportional gain on the rotational speed error is used to adjust the parameters. To speed-up the convergence, the damping may be artificially increased in the iteration step. At the end of the iteration algorithm, the transients have dissipated, the rotor is at the target rotational speed and a steady or periodic operating point is reached. The linearization is then performed for one rotor revolution (if applicable) without the previously added damping. The algorithm is such that it only involves modifications of the *OpenFAST* glue-code and of `ServoDyn`.

1 OpenFAST implementation

1.1 Basic workflow and changes to the code

The basic workflow is listed below. More details follow in subsequent sections.

- New parameters are read from the input files (given in subsection 1.2)
- If the parameter `CalcSteady` is false, the *OpenFAST* linearization process will happen as before (at `LinTimes`)
- If `CalcSteady` is true, *OpenFAST* will iterate and update a controller parameter until the rotational speed matches the parameter `RotSpeed` given in *ElastoDyn*'s input file, and then perform the linearization at this periodic operating point. A different controller variable is adjusted in the iterative step based on the value of the parameter `TrimCase`: the neutral yaw `YawNeut`, the generator torque `GenTrq` or the blade pitch `BlPitch`. The procedure follows the steps below.
- After the initialization of *ElastoDyn*, additional initialization inputs are passed to *ServoDyn* for its initialization: the glue-code inputs `CalcSteady`, `TrimCase` and `TrimGain` and the

reference rotational speed `RotSpeedRef`. These initialization inputs will be used by *ServoDyn* to adjust one of the controller parameter based on the current rotational speed error. If the reference rotation speed is 0 or if the generator degree of freedom is off, no controller trimming is required.

- *ServoDyn* is initialized. The discrete-time state `CtrlOffset` is added to the module to keep track of the controller parameter offset.
- The glue-code starts a special time stepping loop where a convergence criteria is checked upon after each revolution, or time step if the rotational speed is zero. The time stepping loop has the following characteristics:
  - At each time step, if the controller trimming is active, *ServoDyn* updates its controller offset state (`CtrlOffset`) based on the proportional gain `TrimGain` and the error in rotational speed. This offset is added to the controller commands of *ServoDyn* and will hence have an influence on *ElastoDyn*.
  - At each time step, the glue-code adds additional damping, via external forces, to the structure of the modules *ElastoDyn* and *BeamDyn*.
  - At given azimuthal positions (defined by `NAzimStep`), the glue-code computes the relative difference between the output vector of the current revolution and the previous revolution. If the rotational speed is zero, the difference is computed between two successive time steps and the number of azimuthal step is effectively 1.
  - When this difference is below the tolerance `TrimTol` for all the reference azimuthal positions, the simulation has reached a periodic steady state, which also implies that the controller offset parameter has also converged and the rotational speed of the rotor matches the requested set-point. The time-stepping is stopped
- The linearization is performed for one rotor revolution (if applicable) at steps of `NAzimStep`. The operating point is at the requested rotational speed and it uses the controller offset obtained by the iteration procedure above.

The following sections describe the changes needed to the code:

- subsection 1.2: new input parameters to be added to the *OpenFAST* glue-code
- subsection 1.3: changes to *ElastoDyn* to return needed information
- subsection 1.4: iterative procedure of the glue-code to ensure a periodic steady state is reached
- subsection 1.5: changes to *ServoDyn* to compute the controller parameter offset
- subsection 1.6: glue-code procedure to increase the damping and accelerate the convergence

1.2 New glue code inputs

The following input are added to the main *OpenFAST* input file (`.fst` file):

- `CalcSteady - Calculate a steady-state periodic operating point before linearization (-) (switch)`
- `TrimCase - Controller parameter to be trimmed {1:yaw; 2:torque; 3:pitch} [used only when CalcSteady=True]`

- `TrimTol` - Tolerance for the rotational speed convergence [>eps] [used only when CalcSteady=True]
- `TrimGain` - Proportional gain for the rotational speed error (rad/(rad/s) or Nm/(rad/s)) [>0] [used only when CalcSteady=True]
- `Twr_Kdmp` - Damping factor for the tower (N/(m/s)) [>=0] [used only when CalcSteady=True]
- `Bld_Kdmp` - Damping factor for the blade (N/(m/s)) [>=0] [used only when CalcSteady=True]

Note: `NAzimStep` is equivalent to `NLinTimes`, which is already in the *OpenFAST* input file. Also note that in implementation, `TrimTol` must be larger than epsilon.

Linearization inputs are read in `Fast_Subs.f90`, routine `FAST_ReadPrimaryFile` about line 2273. They are returned in the structure named `p` or `p_FAST` of type `FAST_ParameterType`. The parameters above need to be added to the `FAST_Registry.txt` file as `FAST_ParameterType`.

1.3 New initialization outputs from *ElastoDyn*

New registry types The initialization outputs `RotSpeed` and `isFixed_GenDOF` are added:

```
InitOutputType  ReKi     RotSpeed       - - - "Initial or fixed rotor speed" rad/s
InitOutputType  Logical  isFixed_GenDOF - - - "Whether the generator DOF is fixed
   (true) or free (false)" -
```

These variables are sent to the initialization routine of the *ServoDyn* module.

Note: We can't use the *ElastoDyn* output type to initialize *ServoDyn* because the initialization routines do not necessarily calculate output variables (the values defining the output meshes are the only values required to be initialized in the output type). The only exception to this is when *BeamDyn* is used.

1.4 Main glue-code procedure

Main program

- If `CalcSteady` is false, set `Twr_Kdmp` and `Bld_Kdmp` to 0 and proceed as usual
- If `CalcSteady` is true, follow the iterative procedure below

Iterative procedure The subscript p is used to refer to the *previous* time step, the subscript c is used for the *current* time step, the subscript 0 is used to refer to the target azimuthal positions. The azimuthal angle ψ at a given time step are taken from the outputs of *ElastoDyn*:

$$\psi = \texttt{ED\%Output(1)\%LSSTipPxa} \tag{1}$$

The following storage variables are used by the iterative algorithm:

The following steps make up the iterative procedure:

- If the generator degree of freedom is off or if the rotational speed is zero, `TrimCase` is set to 0 so that the trimming is cancelled. This step is done prior to the initialization of *ServoDyn* presented in subsection 1.5.
- If the reference rotational speed is 0, ensure that `NAzimStep`=1. The number of rotations is then understood as the number of time steps.

Variable	Dimensions	Description
n_{rot}	1×1	Number of full rotor revolutions completed; if the reference rotational speed is zero, each time step is considered a full revolution.
j	1×1	Index into target azimuth positions, $j \in [1, \texttt{NAzimStep}]$
n_y	1×1	Total number of outputs included in the linearization analysis of all modules, excluding any `WriteOutput` and extended output values
$\boldsymbol{\psi}_0$	$1 \times \texttt{NAzimStep}$	Target azimuthal positions, $\boldsymbol{\psi}_0[j] \in [0, 2\pi)$
$\boldsymbol{y}_c$	$1 \times n_y$	Output vector (from all modules) at current time step (used for interpolation)
$\boldsymbol{y}_p$	$1 \times n_y$	Output vector (from all modules) at previous time step (used for interpolation)
$\boldsymbol{y}_0$	$1 \times n_y$	Output vector at a target azimuthal position, interpolated using $\boldsymbol{y}_c$ and $\boldsymbol{y}_p$
$\boldsymbol{Y}_0$	$n_y \times \texttt{NAzimStep}$	Output vector interpolated at each target azimuthal position $\psi_0[j]$ (stored from previous revolution)
$\boldsymbol{\epsilon}_y$	$1 \times \texttt{NAzimStep}$	Relative error in the output vector between two revolutions at the same target azimuthal position

- Initialize the number of full rotor revolutions and the index of target azimuthal positions:

$$n_{\text{rot}} = 0, \qquad j = 1 \tag{2}$$

- Perform time stepping until `TMax` (the time loop will be stopped before `TMax` if the convergence criteria is met). For each time step:
  - Call the time step integration routine `FAST_Solution_T`. This routine applies an increased damping (based on `Twr_Kdmp` and `Bld_Kdmp`, see subsection 1.6) If the rotational speed is non-zero and if the generator is on, *ServoDyn* applies an offset to the controller variables (based on ϵ_Ω, see subsection 1.5).
  - Store the current azimuthal angle and output vector: ψ_c and $\boldsymbol{y}_c$
  - If $t = 0$ (or first time step):
    * Set the initial azimuthal position as $\psi_{\text{init}} = \psi_c$ The azimuthal angle is stored as a number in the interval $[0; 2\pi)$ (2π excluded), i.e. $\psi_c = \text{mod}(\psi, 2\pi)$, implemented as ψ_c=`Zero2TwoPi(`ψ`)`.
    * Set the vector of target azimuthal positions $\boldsymbol{\psi}_0$ (also in $[0; 2\pi)$):

$$k = 1..\texttt{NAzimStep}, \quad \boldsymbol{\psi}_0[k] = \text{mod}(\psi_{\text{init}} + (k-1)\Delta\psi, 2\pi), \quad \Delta\psi \triangleq \frac{2\pi}{\texttt{NAzimStep}} \tag{3}$$

    * Set $\boldsymbol{y}_p = \boldsymbol{y}_c$ and $\psi_p = \psi_c$
  - If $t > 0$ and $(\psi_c - \psi_p) \geq \Delta\psi$, return an error: the rotor is spinning too fast; the time step or `NazimStep` are too large.
  - If $\psi_c \geq \psi_0[j]$ (take care with the 2π boundary)
    * Interpolate the output vector to the target azimuthal position $\psi_0[j]$ using the current

output values $\boldsymbol{y}_c$ and the previous ones $\boldsymbol{y}_p$:

$$\text{if } t = 0 \text{ or } \Omega_{\text{ref}} = 0,\ \boldsymbol{y}_0 = \boldsymbol{y}_c, \quad \text{else} \quad \boldsymbol{y}_0 = \boldsymbol{y}_p + (\boldsymbol{y}_c - \boldsymbol{y}_p)\frac{\psi_0[j] - \psi_p}{\psi_c - \psi_p} \tag{4}$$

Note: outputs that are 3D rotations should be transformed to logarithmic maps. Note: special care is needed if angles are close to 0 or 2π, in which case they should be taken between $-\pi$ and π. Note: The output interpolations will be performed by the auto-generated routines in the FAST Registry. I am implementing this to include using the extrap-interp order used in the FAST input file. In the case that the rotational speed is zero, the extrap-interp order for the CalcSteady algorithm is changed to 0 so it will use the current value. I also modified the FAST Registry and NWTC_Library to handle interpolation and extrapolation of angles.

  * If $n_{\text{rot}} > 0$, compute the mean squared relative error of the output vector at the azimuthal position $\psi_0[j]$ between the current revolution and the previous one:

$$\epsilon_y^2[j] = \frac{1}{n_y}\sum_i \left(\frac{y_0[i] - Y_0[i,j]}{y_{\text{ref}}[i]}\right)^2 \tag{5}$$

    The reference value y_{ref} is defined in Equation 8.
    Note: if the variable $y_0[i]$ is in radian or degree, the difference of the variable should be taken between $-\pi$ and π, implemented using `MPi2Pi`.

  * Store the interpolated value

$$\boldsymbol{Y}_0[:,j] = \boldsymbol{y}_0 \tag{6}$$

  * Increment j

- Set the current values as previous values for the next time step

$$\psi_p \leftarrow \psi_c, \qquad \boldsymbol{y}_p \leftarrow \boldsymbol{y}_c \tag{7}$$

- If $j >$ `NAzimStep`:
  * Increment n_{rot}
  * Check convergence over all azimuthal positions: $\epsilon_y^2[k] <$ `TrimTol` for all k
  * If converged, exit the time loop
  * Otherwise, compute a reference value for each of the index of the output vector, based on the maximum and minimum values taken over one rotor revolution.

$$y_{\text{ref}}[i] = |\max(Y_0[:,i]) - \min(Y_0[:,i])| \quad \text{if} \quad y_{\text{ref}}[i] > 10^{-6}, \quad \text{else} \quad y_{\text{ref}}[i] = 1 \tag{8}$$

$$\text{For angles,} \quad y_{\text{ref}}[i] = \min(\pi, y_{\text{ref}}[i]) \tag{9}$$

  * Set $j = 1$ and continue the time stepping.

- If the time loop run up to `TMax`, return an error, otherwise perform the linearization step below

Linearization

- The standard linearization procedure takes place

1.5 Changes in *ServoDyn*

In the updated implementation, *ServoDyn* has the possibility to modify some of its outputs based on offset that is updated internally as a discrete state.

New registry types The inputs `RotSpeedRef`, `TrimCase`, `TrimGain` are added:

```
InitInputType IntKi TrimCase   - - - "Controller parameter to be trimmed"  -
InitInputType ReKi  TrimGain   - - - "Proportional gain on rotational speed error"
    -
InitInputType ReKi  RotSpeedRef - - - "Reference rotational speed"  rad/s
```

These inputs should also be added as `ParameterType` in the registry file. The discrete state `CtrlOffset` (also noted x_{off}) is added:

```
DiscreteStateType ReKi  CtrlOffset  - - - "Controller offset parameter" -
```

Glue-code transfer before the init routine The controller trimming option of *ServoDyn* requires additional parameters from the glue-code and *ElastoDyn*. These parameters need to be transferred via the `SrvD_InitInputType` structure. Currently, these transfer occur in the routine `FAST_InitializeAll` of `FAST_Subs.f90`. The following transfer is added:

```
InitInData_SrvD%TrimCase    = p_FAST%TrimCase
InitInData_SrvD%TrimGain    = p_FAST%TrimGain
InitInData_SrvD%RotSpeedRef = InitOutData_ED%RotSpeed
```

If the generator degree of freedom is off or if the rotational speed is zero, `TrimCase` is set to 0 so that the trimming is canceled. Note that this is done in the glue code prior to calling *ServoDyn* (because *ServoDyn* does not know any DOF information from the structural code).

Initialization routine `Srvd_Init` The added variables from `InitInputType` are copied to the `ParameterType` variables. The state variable `CtrlOffset` is initialized to 0. If the parameter `TrimGain` is not strictly positive, an error is thrown.

Update state routine `Srvd_UpdateDiscState` A simple proportional gain on the rotational speed error is used to correct the control parameters. The error in rotor speed ϵ_Ω is the difference between the target speed and the current rotor speed:

$$\epsilon_\Omega = \Omega_c - \Omega_{\mathrm{ref}} \tag{10}$$

The current rotor speed is $\Omega_c =$ `u%RotSpeed` where `u` is defined at time `t` (i.e., not `t+1`). The offset on the controller variable is computed using the speed error and a proportional gain $k_p > 0$. The offset is computed as:

$$x_{\mathrm{off}} = x_{\mathrm{off}} + s\, k_p\, \epsilon_\Omega \tag{11}$$

The variable s above accounts for possible sign adjustments. The offset is such that it will converge to a constant value as ϵ_Ω tends to 0. When `TrimCase`=1, x_{off} is the yaw angle offset (in rad). When `TrimCase`=2, x_{off} is the generator torque offset (in Nm). When `TrimCase`=3, x_{off} is the pitch angle offset (in rad). For the pitch and generator torque, $s = 1$. Indeed, when the rotational speed is faster than Ω_{ref} ($\epsilon_\Omega > 0$), the pitch or generator torque needs to be increased to lower the rotational speed. The opposite holds when the rotor spins slower than Ω_{ref}. On the other hand, when $\epsilon_\Omega > 0$, the yaw angle needs to be increased if the yaw angle is positive, or decreased if this angle is negative, in order to decrease the rotational speed. For this case, $s = \text{sign}(\theta_{\text{yaw},0} + x_{\text{off}})$, where $\theta_{\text{yaw},0}$ is the neutral yaw angle defined by `p%YawNeut`. Another subtlety arises for the yaw case, since the main variable of *ServoDyn* is actually the yaw moment. Yet, it is more convenient to manipulate an offset on the yaw angle since the offset sign depends on the yaw angle. This issue will be addressed in the next paragraph. The update of the discrete state is implemented as follows:

```
if ((TrimCase==2).or.(TrimCase==3)) then
    xd%CtrlOffset += (u%RotSpeed - p%RotSpeedRef) * p%TrimGain
else if ((TrimCase==1) then
    xd%CtrlOffset += (u%RotSpeed - p%RotSpeedRef) * sign(p%TrimGain, p%YawNeut +
   xd%CtrlOffset)
else
    xd%CtrlOffset = 0
endif
```

Output routine `SrvD_CalcOutput` The output variables of *ServoDyn* are directly modified using the offset `CtrlOffset`. As mentioned in the previous paragraph, in the yaw case, the main variable output by *ServoDyn* is the yaw moment and not the yaw angle. The part of the yaw moment that depends on the yaw angle is computed as:

$$Q_{\text{yaw}} = -k_{\text{yaw}}(\theta_{\text{yaw,ED}} - \theta_{\text{yaw},0}) \tag{12}$$

Hence, if $\theta_{\text{yaw},0}$ is replaced by $\theta_{\text{yaw},0} + x_{\text{off}}$, it is seen that the yaw moment is given the offset $k_{\text{yaw}} x_{\text{off}}$. For the implementation, the control outputs are trimmed just after their computation within the `SrvD_CalcOutput` routine, that is after calling `Pitch_CalcOutput`, `Torque_CalcOutput` and `Yaw_CalcOutput`, as follows:

```
if (TrimCase==1) then
    y%YawMom     = y%YawMom      + xd%CtrlOffset * p%YawSpr
else if (TrimCase==2)
    y%GenTrq     = y%GenTrq      + xd%CtrlOffset
else if (TrimCase==3)
    y%BlPitchCom = y%BlPitchCom  + xd%CtrlOffset
else
    ! do nothing
endif
```

By doing the update in `Svrd_CalcOutput`, it is ensured that the operating point will be about the proper conditions. Indeed, in `Svrd_GetOP`, the operating point variables are set from the outputs directly:

```
y_op(Indx_Y_BlPitchCom) = y%BlPitchCom
y_op(Indx_Y_YawMom)     = y%YawMom
y_op(Indx_Y_GenTrq)     = y%GenTrq
```

It is important to note that the routines `CalculateStandardYaw` and `CalcuateTorque` returns values without offset. These routines are for instance called by `Yaw_UpdateStates` and `Torque_UpdateStates`.

Linearization routine `SrvD_JacobianPInput` This routine computes the partial derivatives of outputs with respect to inputs. The *ServoDyn* inputs in the linearization analysis include only `Yaw`, `YawRate`, and `HSS_Spd`. The modified output equations in `SrvD_CalcOutput` are not functions of any of these variables, thus the partial derivatives are unchanged.

1.6 Additional glue-code procedure to increase the damping

Artificial damping forces are added to the external forces applied on the structure. For now, the extra damping is only applied to *ElastoDyn* and *BeamDyn*. The extra damping force is set to be proportional to the velocity of each node of the structure. The proportionality constants `Twr_Kdmp` and `Bld_Kdmp` are used respectively for nodes on the tower or the blade. In general, the force $\boldsymbol{F}$ on a given node of velocity $\boldsymbol{v}$ is updated as follows:

$$\boldsymbol{F} \leftarrow \boldsymbol{F} - k_{\text{dmp}}\ \boldsymbol{v} \tag{13}$$

To avoid damping the rotation of the blade, the following is applied for nodes along the blade:

$$\boldsymbol{F} \leftarrow \boldsymbol{F} - k_{\text{dmp}}\ (\boldsymbol{v} - \boldsymbol{\Omega}_c \times \boldsymbol{r}) \tag{14}$$

where $\boldsymbol{r}$ is the instantaneous position vector of a blade node. The forces are usually found as inputs of a module while the kinematics are found in the outputs. The additional damping forces can be implemented in the procedure `ED_InputSolve` of the glue-code. For *ElastoDyn*, the update of the forces

```
u%TowerPtLoads%Force(1:3,J) -= Twr_Kdmp * y%TowerLn2Mesh%TranslationVel(1:3,J)
u%BladePtLoads(K)%Force(1:3,J) -= Bld_Kdmp * (
    y%BladeLn2Mesh(K)%TranslationVel(1:3,J) - Vrot)
```

where K is the blade number and J is the node index (along the tower or blade, looping til `NNodes`), u and y are the module input and outputs, and $\texttt{Vrot} = \boldsymbol{\Omega_c} \times \boldsymbol{r}$ is the velocity due to the rotation of the blade node, computed using Equation 15.

For *BeamDyn*, the additional damping is added as follows:

```
u(k)%DistrLoad%Force(1:3,J) -= Bld_Kdmp * (y(k)%BladeMotion%TranslationVel(1:3,J)
    - Vrot)
```

The variable $\texttt{Vrot} = \boldsymbol{\Omega_c} \times \boldsymbol{r}$ is computed for each blade node using

$$\begin{aligned}
\boldsymbol{\Omega}_c &= \Omega_c \cdot \texttt{y\%HubPtMotion\%Orientation(1,:,1)} \\
\boldsymbol{r} &= \boldsymbol{r}_{\text{node}} - \boldsymbol{r}_{\text{hub}} \\
\boldsymbol{r}_{\text{hub}} &= \texttt{y\%HubPtMotion\%Position(1:3,1)} + \texttt{y\%HubPtMotion\%TranslationDisp(1:3,1)} \\
\boldsymbol{r}_{\text{node}} &= \texttt{y\%BladeLn2Mesh(K)\%Position(1:3,J)} + \texttt{y\%BladeLn2Mesh(K)\%TranslationDisp(1:3,J)}
\end{aligned} \tag{15}$$

where $\boldsymbol{r}_{\text{node}}$ is defined for *ElastoDyn* above. It is defined in *BeamDyn* as:

$$\boldsymbol{r}_{\text{node}} = \texttt{y(k)\%BldMotion\%Position(1:3,J)} + \texttt{y(k)\%BldMotion(K)\%TranslationDisp(1:3,J)}$$

assuming that `y(k)%BldMotion` is a sibling mesh of `u(k)%DistrLoad`. In the case that these meshes are not siblings, `MeshMapData%y_BD_BldMotion_4Loads(k)` should be used in place of `y(k)%BldMotion`.