CFDLab

The University of British Columbia

TOSCA User Guide

Exec. Sebastiano Stipa
Date. December 19, 2024

Contents

1 Overview

TOSCA (Toolbox fOr Stratified Convective Atmospheres) is an incompressible finite-volume code, formulated in curvilinear coordinates. Turbulence formulation is done through the Large Eddy Simulation (LES) approach, but inviscid problems can also be solved. The solver uses a fractional-step method to couple momentum balance and mass conservation through a pressure correction equation. Several spatial and time discretization schemes are available (listed in Sec. 4). Depending on user needs, a potential temperature transport equation can also be solved. When this is the case, a buoyancy term is added in the momentum equation to provide temperature-velocity coupling through the Boussinesq approximation. Several source terms are available, among which a wind farm force, damping layers, driving pressure gradients and Coriolis force. The code is designed to simulate atmospheric boundary layers and wind turbines in complex terrains, so it features an advanced immersed boundary method (IBM) - both stationary and moving - which can be used to resolve terrain features, as well as stationary or moving bodies. Great effort has been put in developing a comprehensive acquisition system, which allows to gather flow and turbine statistics of different kinds. At a low level, TOSCA makes extensive use of the PETSc library, making it extremely parallel efficient (it has been tested up to 70k cores).

The TOSCA package, once fully compiled, is made of two executables, `tosca` and `windToPW`, which are used for computing the solution and result visualization respectively.

2 Installation

In order to be installed, TOSCA requires a working C/C++ compiler, PETSc (version 3.14.x, 3.15.x), Open MPI (version 4.0.x, 4.1.x), HDF5 (only for `windToPW`) and HYPRE (needed by PETSc in order to build some of the matrix solvers we use). TOSCA has been tested with the above combinations, it could work with older/newer versions but it has not been tested (especially older versions). We recommend the following versions of the above libraries:

- gcc: 9.2.0 (https://gcc.gnu.org/).
- PETSc: 3.15.5 (https://ftp.mcs.anl.gov/pub/petsc/).
- Open MPI : 4.1.2 (https://www.open-mpi.org/software/ompi/v4.1/).
- HYPRE : 2.20.0 (https://github.com/hypre-space/hypre/tree/hypre_petsc) (check version in /src/CMakeLists.txt).
- HDF5 : 1.12.1 (https://www.hdfgroup.org/downloads/hdf5/).

Prior to install TOSCA, we suggest to create a folder named `Software` inside `$HOME`, where the PETSc, HYPRE and TOSCA folders will be located. Before the installation procedure is described, two important notes are made. First, if you already have OpenFOAM installed, and your set-up is such that OpenFOAM environment variables are automatically loaded as you open the terminal, this will create problems when trying to compile TOSCA. We suggest creating an *alias* in your `.bashrc` file, which will be used to load OpenFOAM environment variables only when you wish to use the latter, and not automatically. This will

allow, for instance, to select your default Open MPI library - if any - , the one downloaded with PETSc or the one you compiled, instead of the Open MPI version selected by the OpenFOAM environment. Regarding the second note, some integration issues have been observed between PETSc and HYPRE. PETSc provides integration for HYPRE, but they are two distinct libraries. We found that not every HYPRE version integrates correctly with PETSc, so we strongly suggest to use the recommended HYPRE version. If the version correct version cannot be found online, e-mail sebstipa@mail.ubc.ca and ask for the HYPRE version which provides a stable PETSc integration. We can now move on to explain how to compile TOSCA. In order to do so, please follow these steps:

1 check your compiler version with `gcc -version`

2 download PETSc into `$HOME/Software`

3 download HYPRE into `$HOME/Software`

4 download OpenMPI. You can download the binaries, compile it from source or have PETSc download it and link it for you by adding `-download-mpicc` in the next step).

5 configure PETSc. In this step HYPRE will be automatically compiled. Open MPI will be compiled if `-download-mpicc` is added to the configuration step, otherwise Open MPI installation location should be passed using `-with-mpi-dir='your-path-to-mpicc'` as an addition to the following suggested configuration options:

```
./configure --with-fc=0 --download-f2cblaslapack --with-mpi-dir
    ='your--path--to--mpicc' --download-hypre='your--path--to--
    hypre' --with-64-bit-indices=1 --with-debugging=0
```

6 Make PETSc with `make all`.

7 Check PETSc installation with `make check`.

8 Save an environment variable that will tell TOSCA where PETSc is installed in your `.bashrc`:

```
echo "export PETSC_DIR=$HOME/your--path--to--petsc" >> $HOME/.
    bashrc
```

9 Save an environment variable that will tell TOSCA which PETSc architecture is required in your `.bashrc`. Note: this is the folder within `$PETSC_DIR` with a name beginning with `arch-`. In an optimized typical installation, it will be `arch-linux-c-opt`:

```
echo "export PETSC_ARCH=arch-linux-c-opt" >> $HOME/.bashrc
```

10 Add the PETSc shared libraries to your library path environment variable in your `.bashrc`:

```
echo "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PETSC_DIR/
    $PETSC_ARCH/lib" >> $HOME/.bashrc
```

11 Reload the environment variables with

```
source $HOME/.bashrc
```

12 Go inside `TOSCA/src` directory and compile the executables with `make tosca` and `make windToPW`.

13 Test the installation: copy the two executables (`tosca` and `windToPW`) inside the `TOSCA/tests/NeutralABLTest` and run `./tosca` for serial computations, `mpirun -np 'your-num-of-procs'./tosca` for parallel computations. Once the simulation finishes (or you can kill it after a few iterations), run `./windToPW` to create ParaView files.

3 Getting Started

TOSCA is designed to run on massive supercomputers with hundreds of cores, thanks to its excellent parallel strong scaling efficiency. Nevertheless, some small test cases, which should run on most PC architectures, are provided which allow users to familiarize with the code and its capabilities.
One should keep in mind that TOSCA uses generalized curvilinear coordinates, so that all kinds of structured meshes can be handled (cartesian and deformed). From a mathematical point of view, this allows to use a cartesian-like discretization approach in the curvilinear directions. Loosely speaking, such directions are defined depending on how mesh cells are indexed in the loops, i.e. through i,j,k indexing. It is clear from Fig. 1 that, unless the mesh is cartesian, it is impossible to refer to boundary patches in terms of their caresian coordinates. As a consequence, frequently in this document we will speak in terms of curvilinear directions (k,j,i, or ζ, η, ξ) rather than cartesian coordinates. When ABL capabilities are activated in TOSCA, a cartesian (possibly stretched) mesh must be used, where z is the vertical direction, y is the spanwise direction and x is the streamwise direction. When this is the case, the following convention is adopted for relating curvilinear to cartesian directions: k (or ζ) direction corresponds to x, j (or η) direction corresponds to z and i (or ξ) direction corresponds to y. This must be kept in mind when applying boundary conditions, as boundaries are referred to as *iLeft*, *iRight*, *jLeft*, *jRight*, *kLeft* and *kRight*, as shown in Fig. 1. TOSCA automatically handles this transformation if a .xyz file format is used, and only the k,j,i = x,z,y convention has to be kept in mind in order to know which boundary is the wall and so on. When the mesh is deformed, a unique relation between the two set of coordinates doesn't exist anymore, and curvilinear coordinates are assigned depending on how the mesh file has been created. In particular TOSCA always uses nested loops with k,j,i ordering to index mesh cells. As a consequence, to yield the situation displayed on the right of Fig. 1, the .grid file should store all points with j=1 first, then j=2 and so on. To further clarify this concept, consider the case where a cartesian mesh is provided trough the .grid format. To retain TOSCA's convention in this case, the file should be created indexing the points with a nested loop with x,z,y ordering. In this case, as the file is read from TOSCA, coordinates are stored in the k,j,i directions.
The above discussion is only needed by the user to know which boundary is which, consequently being able to apply boundary condition the way he intends to. Not respecting the convention doesn't have an impact on the code behavior if ABL is de-activated (`-abl` flag set to 0 or omitted in the `control.dat` file).

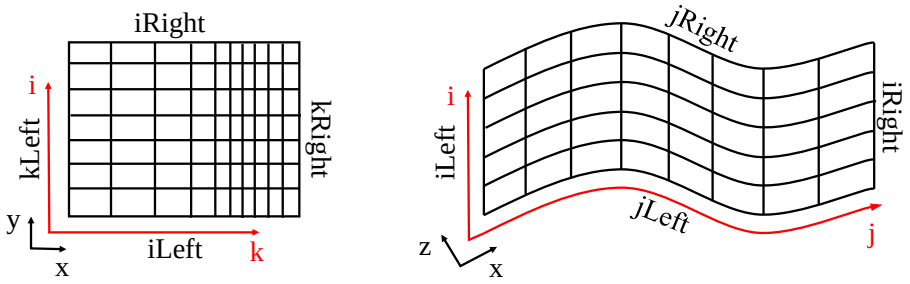

Figure 1. Left: TOSCA's convention $k, j, i = x, z, y$, needed when running ABL simulations. Right: generalized curvilinear structured mesh. Boundary conditions are applied on curvilinear boundaries, which are referred as *iLeft*, *iRight* etc.

3.1 The NREL 5MW Example Case

The *NREL5MWTest* simulates the flow around an NREL 5MW wind turbine. The `mesh.xyz` mesh file provides the mesh points along the x,y,z axes. The domain goes from -250 to 250 m in the x and y direction, while it goes from -90 to 210 m in the z direction. Due to the TOSCA convention, x will be the k direction, y will be the i direction and j will be the z direction. Such information is used to set boundary conditions inside the `boundary` folder. Boundary conditions are provided for velocity and sub-grid-scale (SGS) viscosity (temperature equation is not active in this example). Pressure boundary conditions in TOSCA are Neumann conditions at all boundaries, hence they don't need to be provided. If the case has periodicity in any curvilinear direction, pressure solution internally accounts for that. Looking at the `nut` file, we can see that the domain is periodic in the j direction (y), a zero gradient condition is applied at the inlet and outlet (k direction), while *nut* is set to zero at the top and bottom patches (j direction). The keyword `internalField` specifies the initial condition (see Sec. 4 for the available options), which is set to `spreadInflow` for this case, meaning that the 2D field at the `kLeft` boundary will be set throughout the domain. Since *nut* has a zero gradient boundary condition at the `kLeft` boundary, this field is not set in reality, so the initial field will be equal to zero for the SGS viscosity. Regarding the velocity boundary and initial conditions, listed in the `U` file, we can see that on the i-patches the `periodic` keyword is used. In fact, all fields should be set to `periodic` on periodic boundaries, as these imply a change in mesh connectivity. The top (*jRight*) boundary is set to `slip`, while for the ground (*iLeft*) a wall model is used (`type -3` corresponds to the Shumann wall model, see Sec. 4). At the outlet (*kRight*) the velocity is set to *zeroGradient*, while at the inlet a `fixedValue` boundary condition is used, with the velocity equal to 5 m/s along the x direction (vectors should be always provided with their cartesian components, curvilinear reasoning is only used for boundaries), depicted in violet in Fig. 2. Another

possible inlet boundary conditions for this case is

```
kLeft inletFunction
{
    type 1
    Uref (9.0 0.0 0.0)
    Href 90
    uPrimeRMS 0.0
}
```

which provides an exponential inflow with a 9 m/s wind at the reference height of 90 m (blue in Fig. 2).

In TOSCA, inlet functions are only provided for the *kLeft* patch, and wall models are only provided for the *jLeft* and *jRight* patches. As a consequence, if these capabilities are needed, the inlet should coincide with the *kLeft* patch, while the ground should be located on the *jLeft* or *jRight* patch (or both). Conversely, since *fixedValue* is available for all patches and `-abl` is not activated for this example case, any patch could be selected as inlet. We also suggest to try `noSlip` boundary condition at the wall (which is also available for all patches) in order to see the difference w.r.t. the wall model.

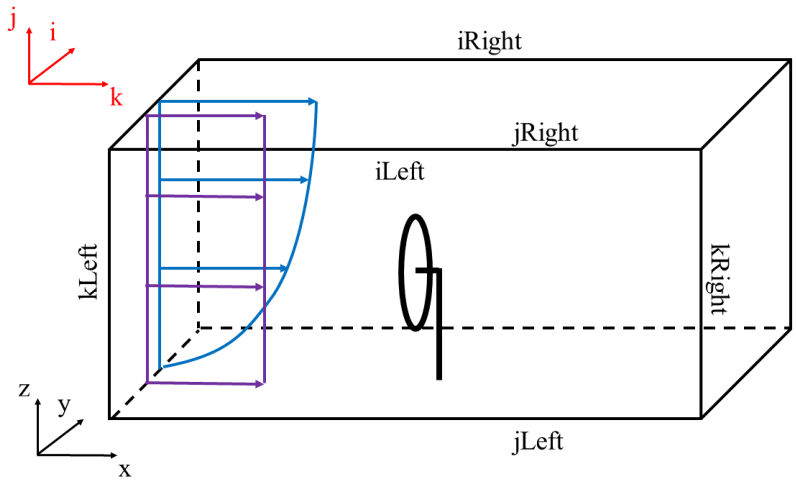

Figure 2. Sketch of the NREL 5MW turbine test case.

Finally, initial velocity is also set to `spreadInflow`, so that the 2D field at the inlet ghost nodes is spread throughout the domain.

Inside the `turbines` folder, turbine and wind farm definition is provided. The top level file is `windFarmProperties`, where output settings and the wind farm array are defined. In this case, a single wind turbine is provided and its base is located at x = 0, y = 0, z = -90 m (the base is located at the wall). `A1` indicates the ID if the wind turbine, and it is just a name. In its definition, the `turbineType` and `turbineModel` have to be specified. The first is the name of a file which contains all necessary turbine information, while the second is the type of model. Depending on the model, more or less information is required in the `turbineType` file. In this example, the turbine is modeled using the actuator line model (`ALM`). The latter requires the same level of information as the actuator disk model (`ADM`),

while the uniform actuator disk (UADM) and the actuator farm models (AFM) do not require e.g. the airfoils and bladeData sub-dictionaries contained in the turbineType file. In this example, all turbine controllers (pitch, yaw and angular speed) are active. An interesting variation of this example is to simulate how the turbine responds to an initial misalignment with the flow direction, obtained by setting a non-zero y-component different to the inlet flow velocity, or to the initial rotor direction (rotorDir entry in the turbineType file).

In the control.dat file, the main simulation parameters are defined. In this example case, the simulation will start from 0, until it reaches a time of 500 s. Time step is dynamically adjusted based on the CFL value of 0.8, but it will also be adjusted to exactly hit multiples of acquisition periods required for example by the mechanical energy budgets (equal to 2 s and set inside sampling/keBudgets by the keyword avgPeriod) or 3D field averages (equal to 3 s and set inside the control.dat file by the keyword -avgPeriod, which refers to the averages activated by -averaging set to 1).

After copying tosca executable inside the case directory, the case can be run in serial by typing

```
./tosca
```

or in parallel with e.g. 4 processors by typing

```
mpirun -np 4 ./tosca
```

As the case starts, a postProcessing directory, which stores all acquisition outputs, and a fields directory, which stores the fields and turbines checkpoints files, are created. Checkpoints are created every 10 seconds, as defined by the keyword -timeInterval in the control.dat file. Such value can be changed during the simulation, and an input update will be triggered. If the users desires to write data every N iterations instead of seconds, the keyword -intervalType should be changed to timeStep. If the user wants to delete all previous checkpoint files after writing each new checkpoint, the keyword -purgeWrite should be set to 1. If the user desires to write the checkpoint and terminate the simulation at the next iteration, the keyword -intervalType should be set to writeNow.

After the simulation finished, ensure that the keyword -postProcessFields is set to 1 in the control.dat file, then launch the application windToPW by typing

```
./windToPW
```

after it has been copied in the case directory. This will create an XMF folder which contains 3D and 2D (if -sections are activated in the control.dat file) fields to be visualized in Paraview. After opening Paraview, navigate inside the XMF folder and open the files with the .xmf extension in order to visualize the results. If the case is too big to be contained in memory (this holds for bigger cases), the application windToPW can be launched in parallel, but it doesn't support 3D field post processing. As a consequence, the -postProcessFields keyword should be set to 0 in the control.dat file, and only 2D sections will be processed in parallel.

4 Input Files

TOSCA uses ASCII input files, organized in *files*, *dictionaries* and *subdictionaries*. The latter two levels of description are always embodied using '{}' parenthesis. The code provides some level of input checking, meaning that non-recognized inputs are followed by an error message listing available possibilities. TOSCA has a standardized case structure, of which two examples are given in Fig. 3.

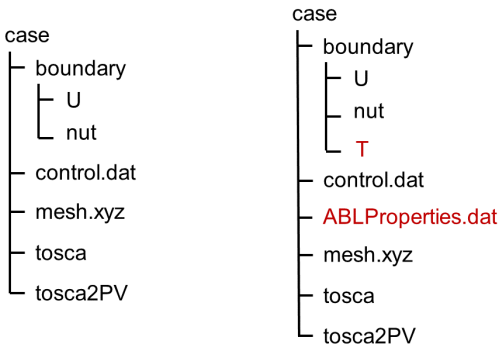

Figure 3. Examples of TOSCA's case structures.

The left example corresponds to the simplest possible structure, while temperature solution and atmospheric boundary layer capabilities are activated on the right. The `control.dat` file is TOSCA's top-level dictionary, and contains information about the type of simulation, time stepping, numerical schemes, I/O, and more. Optional flags are activated here, and they usually require additional input files, such as `ABLProperties.dat` and `T` in the example shown in Fig. 3. These additional files are listed in Sec. 4, when their corresponding activation flag is described. Inside the `boundary` folder, initial and boundary conditions are set in files having the same name as the field they describe. The mesh can be provided in different formats, in Fig. 3 a cartesian *xyz* format is used, which only provides 1D discretization along the three axes. The files `tosca` and `tosca2PV` are the two executables, which must be copied inside the case directory if their path is not added to the environment variable `$PATH`.

4.1 Spatial Mesh

The code can read two types of mesh formats, namely *.xyz* and *.grid*. The former is used for Cartesian (possibly stretched) meshes, and it very convenient as it only provides the discretization along the three axes. An example of this format is given by

```
Nx Ny Nz
x(k=1,j=1,i=1) y(k=1,j=1,i=1) z(k=1,j=1,i=1)
x(k=2,j=1,i=1) y(k=2,j=1,i=1) z(k=2,j=1,i=1)
:
x(k=Nk,j=1,i=1) y(k=Nk,j=1,i=1) z(k=Nk,j=1,i=1)                    :
x(k=1,j=1,i=1) y(k=1,j=1,i=1) z(k=1,j=1,i=1)
x(k=1,j=1,i=2) y(k=1,j=1,i=2) z(k=1,j=1,i=2)
:
x(k=1,j=1,i=Ni) y(k=1,j=1,i=Ni) z(k=1,j=1,i=Ni)
```

```
x(k=1,j=1,i=1) y(k=1,j=1,i=1) z(k=1,j=1,i=1)
x(k=1,j=2,i=1) y(k=1,j=2,i=1) z(k=1,j=2,i=1)
:
x(k=1,j=Nj,i=1) y(k=1,j=Nj,i=1) z(k=1,j=Nj,i=1)
```

where the $k, j, i = x, z, y$ convention to relate the curvilinear and Cartesian frames of reference has been used. Regarding the *.grid* format, coordinates are provided as

```
Ni Nj Nk
x(k=1,j=1,i=1).....x(k=1,j=1,i=Ni)
:                        :
x(k=1,j=Nj,i=1)....x(k=1,j=Nj,i=Ni)
x(k=2,j=1,i=1)....x(k=2,j=1,i=Ni)
:                        :
x(k=Nk,j=Nj,i=1)...x(k=Nk,j=Nj,i=Ni)
y(k=1,j=1,i=1).....y(k=1,j=1,i=Ni)
:                        :
y(k=1,j=Nj,i=1)....y(k=1,j=Nj,i=Ni)
y(k=2,j=1,i=1)....y(k=2,j=1,i=Ni)
:                        :
y(k=Nk,j=Nj,i=1)...y(k=Nk,j=Nj,i=Ni)
z(k=1,j=1,i=1).....z(k=1,j=1,i=Ni)
:                        :
z(k=1,j=Nj,i=1)....z(k=1,j=Nj,i=Ni)
z(k=2,j=1,i=1)....z(k=2,j=1,i=Ni)
:                        :
z(k=Nk,j=Nj,i=1)...z(k=Nk,j=Nj,i=Ni)
```

In both the *.xyz* and *.grid* formats, if any periodicity is present, its type must be set at the beginning of the file using

```
-iPeriodicType 1 or 2
-jPeriodicType 1 or 2
-kPeriodicType 1 or 2
```

Note that, conversely to the *.xyz* format, the *.grid* mesh format can be very heavy, as all coordinates for each mesh point are specified. This is used when the mesh is deformed and a unique relation between the two set of coordinates does not exist anymore. In this case, curvilinear coordinates are assigned depending on how the *.grid* mesh file has been created. In particular TOSCA always uses nested loops with k, j, i ordering to index mesh cells. As a consequence, to yield the situation displayed on the right of Figure 1, the *.grid* file should store all points with j=1 first, then j=2 and so on. To further clarify this aspect it is worth considering the case where a Cartesian mesh is provided trough the *.grid* format. To retain TOSCA's convention in this case, the file should be created indexing the points with a nested loop with x, z, y ordering. In this case, as the file is read from TOSCA, coordinates are stored in the k, j, i directions. The $k, j, i = x, z, y$ convention has to be kept in mind when defining boundary conditions at every patch. Not respecting the convention does not have an impact on the code behavior if ABL capabilities are de-activated by setting the `-abl` flag to 0 (or omitting it) in the `control.dat` file (see Section 4.2)

4.2 Case Structure

TOSCA uses ASCII input files, organized in *files*, *dictionaries* and *subdictionaries*. The latter two levels of description are always embodied using '{}' parentheses. The code provides some level of input checking, meaning that non-recognized inputs are followed by

an error message listing available possibilities. TOSCA has a standardized case structure, of which two examples are given in Figure 4. The left example corresponds to the simplest

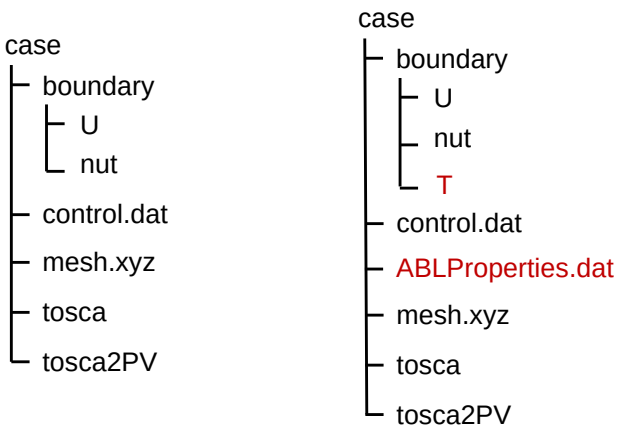

Figure 4. Examples of TOSCA's case structure. The situation to the right corresponds to the case where `-abl` and `-potentialT` flags are activated in the *control.dat* file.

possible structure while, on the right, temperature solution and atmospheric boundary layer capabilities are activated. The `control.dat` file is TOSCA's top-level dictionary, and contains information about the type of simulation, time stepping, numerical schemes, I/O, and more. Optional flags are activated here, which trigger the read of additional input files, such as `ABLProperties.dat` and `T` in the example shown in Figure 4. These additional files are listed in Section 4.3, when their corresponding activation flag is described. Inside the `boundary` folder, initial and boundary conditions are set within files that have the same name as the described field. In the example of Figure 4, the mesh is given in a *.xyz* format. The files `tosca` and `tosca2PV` are the two executables, which should be copied inside the case directory if their path is not added to the environment variables.

4.3 Input Data Description

Input parameters contained in `control.dat` are subdivided into 8 groups, depending on their area of interest. These are *Time Controls*, *I/O Controls*, *Solution Flags*, *Solution Controls*, *Constants*, *Mesh*, *Acquisition* and *Post Processing*. The latter group is only needed for the `tosca2PV` executable.

keyword	description
Time Controls	
`-startFrom`	It can be set to `startTime` (requires `-startTime` entry), or `latestTime`. In the last case, the latest time available in the `fields` directory is used as initial time (the code exits if the folder is not present, or it doesn't contain any time folders)
`-startTime`	Specifies the initial time of the simulation if the `startFrom` keyword is set to `startTime`. Disregarded otherwise.
`-endTime`	Defines the time at which the simulation ends.
`-timeStep`	Initial time step size in seconds.
`-adjustTimeStep`	If set to 0, the time step will remain fixed and equal to the specified `timeStep` size. If set to 1 (requires `-cfl`), the time step will be adjusted based on the CFL value and I/O settings. This means that time step size will be varied based on both the solution and in order to land on those time values designated for data writing.
`-cfl`	Specifies the CFL value to be maintained. Disregarded if `adjustTimeStep` is set to 0.
`-timePrecision`	Specifies the number of digits after the comma that are used to write files and expected to read them. Note that, if the time folder storing the initial condition has a different number of digits, the simulation will throw an error. To solve this one can act both on the `timePrecision`, or rename the folder using a `timePrecision` number of digits.
I/O Controls	
`-intervalType`	It can be set to `adjustableTime`, `timeStep` or `writeNow`. In the last case, a checkpoint followed by termination of the simulation will be triggered.
`-timeInterval`	Specifies how often a checkpoint file is written. If the `-timeInterval` is set to `adjustableTime`, the time interval between two checkpoints is expressed in seconds. If `-timeInterval` is set to `timeStep`, the time interval refers to the number of iterations.
`-purgeWrite`	If set to 1 eliminates all previous checkpoint files every time that a checkpoint is written (in order to save disk space).
Solution Flags	
`-les`	Specifies if LES model is active (1: standard Smagorinsky, 3: dynamic Smagorinsky with cubic averaging, 4: dynamic Smagorinsky with lagrangian averaging), or not (set to 0).
`-potentialT`	Specifies if potential temperature transport equation is solved (set to 1) or not (set to 0).
`-abl`	Specifies if an ABL simulation is run. Requires additional file `ABLProperties.dat`

keyword	description
-windplant	Specifies if wind turbines are present in the simulation (set to 1) or not (set to 0). Requires turbine models definitions in turbines directory.
-ibm	Specifies if immersed bodies are present in the simulation (set to 1) or not (set to 0). Requires additional input in IBM directory.
-zDampingLayer	Specifies if vertical Rayleigh damping layer is present in the simulation. Requires additional input in ABLProperties.dat file if activated.
-xDampingLayer	Specifies if horizontal fringe region is present in the simulation. Requires additional input in ABLProperties.dat file if activated. Concurrent precursor can be enabled with this flag.
-kLeftRayleigh	Specifies if horizontal Rayleigh damping at kLeft boundary is present in the simulation. Requires additional input in ABLProperties.dat file if activated.
-kRightRayleigh	Specifies if horizontal Rayleigh damping at kRight boundary is present in the simulation. Requires additional input in ABLProperties.dat file if activated.
-canopy	Specifies if wind farm canopy model is present in the simulation. Requires additional input in ABLProperties.dat file if activated.
-overset	Specifies if overset mesh is present in the simulation. Requires additional input in Overset directory if activated.
-inviscid	If set to 1 allows to disable viscous terms. Default value is 0.
Solution Controls	
-meshFileType	Defines the format of the mesh input file. It can be set to cartesian or curvilinear.
-dUdtScheme	Time discretization scheme, it can be set to forwardEuler (explicit first order, usually unstable), rungeKutta4 (explicit fourth-order Runge-Kutta) or backwardEuler, which corresponds to the second-order implicit Crank-Nicholson scheme (explicit selection of the Crank Nicholson scheme will be made available). For long simulations the backwardEuler scheme is preferred, as it can run with CFL greater than 1 and it is unconditionally stable. For simulations affected by constraints other than the CFL (e.g. blade rotation in actuator line model), rungeKutta4 is a good alternative.
-divScheme	Determines which divergence scheme is used for the discretization of the advection fluxes. It can be set to central (second-order symmetric scheme, dispersive), quickDiv (third-order upwind-biased quadratic scheme, diffusive), weno3 (fourth-order weighted essentially non-oscillatory scheme, diffusive), centralUpwind (vanLeer blending of central and quadratic scheme, to balance diffusion and dispersion), centralUpwindW (weighted version, for graded/non-uniform meshes).
-relTolU	Requires -dUdtScheme set to backwardEuler, discarded otherwise. Allows to set the relative exit tolerance for the Newton method used to solve implicit discretized momentum equation, default value 1e-30.

keyword	description
-absTolU	Requires -dUdtScheme set to backwardEuler, discarded otherwise. Allows to set the absolute exit tolerance for the Newton method used to solve implicit discretized momentum equation, default value 1e-5.
-poissonSolver	Allows to specify the library used to solve the pressure equation, it can be set to HYPRE or PETSc. HYPRE is suggested, as it has proved to work better than PETSc.
-hypreSolverType	Allows to choose the solution method for the linear system if -poissonSolver is set to HYPRE, discarded otherwise. Set to 1 to use the Generalized Minimum Residual (GMRES), set to 2 to use the preconditioned Conjugate-Gradient (PCG) method. Default value is 1.
-poissonTol	Allows to set the exit tolerance for the pressure solver. Default value is 1e-8.
-poissonIt	Set the maximum number of iterations for the pressure solver. Default value is 8.
-amgCoarsenType	Since TOSCA uses the Algebraic Multi-Grid (AMG) preconditioner when the -poissonSolver is set to HYPRE, this entry allows to set the coarsening method. Available entries are 0 (CLJP), 6 (Falgout), 8 (PMIS), 10 (HMIS). Default value is 10.
-amgThresh	Allows to set the AMG threshold. Default value is 0.5. For distorted meshes a value of 0.6 is suggested.
-amgAgg	Allows to set the level of aggressive coarsening. Default value is 0 (not used).
-pTildeBuoyancy	If set to 1, buoyancy force is recast into a buoyancy gradient and pressure is defined accordingly. Default value is 0 (not used).
-dTdtScheme	Can be set to backwardEuler (implicit first-order) or rungeKutta4 (explicit fourth-order). For ABL simulations backwardEuler is suggested.
-relTolT	Requires -dUdtScheme set to backwardEuler. Allows to set the relative exit tolerance for the Newton method used to solve implicit discretized temperature equation, default value 1e-30.
-absTolT	Requires -dUdtScheme set to backwardEuler. Allows to set the absolute exit tolerance for the Newton method used to solve implicit discretized temperature equation, default value 1e-5.
-max_cs	Maximum value for the LES model C_s coefficient, default value is set to 0.5.
Solution Constants	
-nu	Sets the molecular (kinematic) viscosity of the working fluid.
-rho	Sets the density of the working fluid (used e.g. to compute forces).
-Pr	Requires -potentialT set to 1. Sets the Prandtl number of the working fluid.
-tRef	It is a required parameter when -potentialT is active and -abl is not. Sets the reference potential temperature of the flow.
Acquisition Controls	
-probes	Activates probes acquisition. Requires additional input files inside sampling/probes directory.

keyword	description
-sections	Activates acquisition of sections to be visualized in ParaView. Requires additional input files in sampling/surfaces directory.
-averageABL	Activates planar averages at every cell-level in the z-direction. Requires -abl to be active.
-averageABLPeriod	Output period of the ABL planar averages. It is a required parameter, even if -averageABL is set to 0, for concurrent-precursor simulations.
-averageABLStartTime	Time at which ABL planar averages are started. It is a required parameter, even if -averageABL is set to 0, for concurrent-precursor simulations.
-average3LM	Activates vertical averages within layer at user-defined points. Requires additional inputs in sampling directory.
-perturbABL	Activates acquisition of perturbation fields at the same location as sections to be visualized in ParaView. Requires additional inputs in sampling directory.
-averaging	It can be activated by setting to 1, 2 or 3 to get a higher amount of three-dimensional averaged fields.
-avgPeriod	Average period of three-dimensional averages. Fields are written at checkpoint times in the correspondent time folder.
-avgStartTime	Start time of three-dimensional averages.
-phaseAveraging	These averages are a duplicate of the averages, but are useful if one wants to perform both unconditioned-averages and phase-averages, e.g. at multiples of some characteristic time, in the same simulation.
-phaseAvgPeriod	Average period of three-dimensional phase averages. Fields are written at checkpoint times in the correspondent time folder.
-phaseAvgStartTime	Start time of three-dimensional phase averages.
-keBudgets	Set to 1 to activate mechanical energy budgets. Requires additional inputs in sampling directory.
-writePressureForce	Writes pressure force on the IBM surface.
-computeQ	Writes 3D field of Q-criterion at checkpoint times.
-computeL2	Writes 3D field of Lambda2-criterion at checkpoint times.
-computeFarmForce	Writes 3D field of wind farm body force at checkpoint times.
-computeSources	Writes 3D field of each source term present in the momentum equation at checkpoint times.
-computeBuoyancy	Writes 3D field of buoyancy term in the momentum equation at checkpoint times.
Post Processing Controls	
-postProcessFields	Activate to post process 3D fields. It should be deactivated (set to 0) for too big cases to be fit in the memory of a single node.
-writeRaster	Activate to write raster file from jSections.
-sections	Activate to post process binary sections and write XMF and HDF5 files to be visualized in Paraview.
-postProcessPrecursor	Activate to post process also fields from the concurrent precursor simulation.

Table 1. Available inputs in TOSCA's control.dat file.

Post processing controls are only read by `tosca2PV`, the TOSCA post processor. The latter converts binary fields generated from `tosca` into HDF5 files. An additional file is written in xml format, which keeps track and collects the names of all HDF5 files. This can be opened in Paraview using the XMFReader. Currently, three-dimensional field conversion from binary to XMF is only available in serial. Hence, those cases with a size such that they cannot be contained in memory cannot be visualized unless TOSCA is coupled with Paraview Catalyst. Conversely, parallel post processing is enabled for two-dimensional sections of the flow.
Regarding the available boundary conditions in TOSCA, these are listed in Table 2 and must be specified in their respective field files (e.g. `U`, `nut` and `T`) inside the `boundary` directory. Only for the *kLeft* patch, TOSCA features special boundary conditions referred to as *inletFunctions*, such as inflow data mapping from a database for successor simulations, or velocity and temperature profiles that are specific for ABL flows. These special functions are reported in Table 3. The pressure field does not require the specification of boundary conditions. In fact, except when two opposite patches are `periodic`, a `zeroGradient` boundary condition is always applied on pressure by TOSCA and this is internally handled by the code.

syntax	description
`fixedValue` *val*	Available for `U`, `nut` and `T`. For scalars *val* is a scalar value, for vectors is provided as (val_x, val_y, val_z).
`fixedGradient` *val*	Available for `T`, *val* is a scalar value normal to the patch.
`zeroGradient`	Available for `U`, `nut` and `T`.
`slip`	Available for `U`.
`noSlip`	Available for `U`.
`periodic`	Available for `U`, `nut` and `T`. Requires keywords `iPeriodicType`, `jPeriodicType`, `kPeriodicType` in the mesh file depending on the patch to which it is applied.

`velocityWallFunction`
`{`
` type` *type*
` kRough` *val*
` gammaM` *val*
` kappa` *val*
` thetaRef` *val*
` uStarEval` *type*
`}`
	Available only for `U`. Applies wall models to velocity and temperature (if active); `type` is the type of wall function (only -3 available), `kRough` is equivalent roughness z_0, `gammaM` is a model coefficient, `kappa` is the von Karman constant, `thetaRef` is the reference potential temperature and `uStarEval` can be set to `averaged` or `localized` for horizontally homogeneous and non-homogeneous flows, respectively.

`inletFunction`
`{`
` type` *type*
` parameters...`
`}`
	Available for `U`, `nut` and `T` only at the *kLeft* patch; `parameter...` indicates additional parameters for the specific inlet function `type`. See Table 3 for all available possibilities.

Table 2. Available boundary conditions in TOSCA.

parameters	description
`type 1` - power law profile	
`Uref` (val_x, val_y, val_z) `Href` *val*	Power law velocity profile $\mathbf{U} = \mathbf{U}_{\text{ref}}(z/H_{\text{ref}})^{\alpha}$, with $\alpha = 0.107027$, `uPrimeRMS` adds random fluctuations; *kLeft* and *kRight* as bottom and top boundaries.

parameters	description
`uPrimeRMS` *val*	
`type 2` - logarithmic profile	
`directionU` (val_x,val_y,val_z) `hInversion` *val* `frictionU` *val* `kRough` *val*	Logarithmic velocity profile $\mathbf{U} = u* /0.4\ln(z/z_0)\mathbf{e}_U$; $\mathbf{e}_U$ given by `directionU`, $u*$ by `frictionU`, z_0 by `kRough`; $\mathbf{U}$ is constant above `hInversion`.
`type 3` - unsteady mapped inflow	
`n1Inflow` *val* `n2Inflow` *val* `n1Periods` *val* `n2Periods` *val* `n1Merge` *val*	Mapping of inflow data from `inflowDatabase/U`, `inflowDatabase/nut`, `inflowDatabase/T` to *kLeft* patch. Data has `n1Inflow`, `n2Inflow` points in j,i directions and is periodized with `n1Periods` and `n2Periods`. Data at 10 top $j-$cells is averaged if `n1Merge` is set to 1.
`type 4` - unsteady interpolated inflow	
`n1Inflow` *val* `n2Inflow` *val* `n1Periods` *val* `n2Periods` *val* `n1Merge` *val* `n2Shift` *val* `shiftSpeed` *val* `sourceType` *type* `cellWidth1` *val* `cellWidth2` *val*	Mapping of inflow data from `inflowDatabase/U,nut,T`. Data has `n1Inflow`, `n2Inflow` points in j,i directions and is periodized with `n1Periods` and `n2Periods`. Data at 10 top $j-$cells is averaged if `n1Merge` is set to 1 and $i-$shifted with speed `shiftSpeed` if `n2Shift` is set to 1. Inflow grid may be different from patch grid. For uniform inflow grid `sourceType` is set to `uniform` and j,i spacings `cellWidth1` and `cellWidth2` should be provided. For stretched inflow grid `sourceType` is set to `grading` and inflow mesh should be provided in `inflowDatabase/inflowMesh.xyz`.
`type 5` - Nieuwstadt (1983) model	
`directionU` (val_x,val_y,val_z) `hInversion` *val* `hReference` *val* `frictionU` *val* `kRough` *val* `latitude` *val*	Applies the Nieuwstadt (1983) model with veer to *kLeft* patch. Flow is directed along `directionU` at `hReference`, uniform above `hInversion`. Latitude (`latitude`), $u*$ (`frictionU`) and z_0 (`kRough`) are model parameters.
`type 6` - sinusoidally varying $i-$th component	
`Uref` (val_x,val_y,val_z) `amplitude` *val* `periods` *val*	Uniform inflow `Uref` where magnitude varies sinusoidally along i with amplitude `amplitude` and `periods` $i-$periods.

Table 3. Inlet functions available in TOSCA for the *kLeft* boundary patch.

The boundary conditions listed in Table 2 can be specified in `U`, `nut` and `T` files as follows

```
internalField 'entry'
kLeft  'entry'
kRight 'entry'
jLeft  'entry'
jRight 'entry'
iLeft  'entry'
iRight 'entry'
```

where `'entry'` indicates one of the boundary condition syntax defined in Table 2. The `internalField` keyword determines how the initial condition is applied to the simulation. The available possibilities are listed and explained in Table 4.

syntax	description

`uniform`
`{`
`value` *val*
`perturbations` *val*
`}`
	Available for `U`, `nu`, `T`. For vectors *val* is replaced with (val_x,val_y,val_z); `perturbations`, if set to 1, applies sinusoidal perturbations to trigger turbulence, only required for `U`.

`readField`	Available for `U`, `nu`, `T`. Reads field from `fields` directory.
`ABLFlow`	Sets initial log profile for `U`, while `T` is set according to the Rampanelli and Zardi (2003) model. Requires `-abl` set to 1 in *control.dat*, and `ABLProperties.dat` file.
`spreadInflow`	copies the flow from the *kLeft* ghost cells at every $k-$plane. It is useful to ensure perfect consistency between the inlet boundary condition and the internal field when using `inletFunction`.

`linear`
`{`
`tRef` *val*
`tLapse` *val*
`}`
	Ony available for `T`, sets an initial temperature characterized by a linear lapse rate `tLapse` along j and ground temperature `tRef`.

Table 4. Available initial conditions in TOSCA.

In the remainder of this section, input relative to the `-abl` and `-windplant` flags reported in Table 1 are described. For brevity, settings relative to the acquisition system (i.e. probes, slices, energy budgets and turbulence statistics), `-overset` and `-ibm` flags are not included in the present manuscript. The interested reader can consult the available example cases at the TOSCA's GitHub repository.
By activating ABL capabilities through the `-abl` flag, the additional file `ABLPr- operties.dat` is read by TOSCA. This can be used to activate different aspects of TOSCA, such as Rayleigh damping regions, fringe regions, the Coriolis force, velocity and temperature controllers, advection damping and canopy force. Table 5 lists and explains all keywords listed in the `ABLProperties.dat` file.

syntax	description
`hRough` *val*	Equivalent roughness height z_0.
`uRef` *val*	Reference velocity at height `hRef`.
`hRef` *val*	Reference height.
`hInv` *val*	Capping inversion height.
`dInv` *val*	Capping inversion width.
`gInv` *val*	Potential temperature jump across capping inversion.
`tRef` *val*	Reference potential temperature.
`gTop` *val*	Lapse rate above capping inversion.
`gABL` *val*	Lapse rate below capping inversion.
`vkConst` *val*	Von Karman constant.
`smearT` *val*	[**?**] smearing parameter.
`coriolisActive` *val*	Coriolis force activation flag.
`fCoriolis` *val*	Coriolis parameter. Should be computed as $7.272205217\sin(\phi)\cdot 10^{-5}$, where ϕ is latitude.
`controllerActive` *val*	Activates velocity controller and reads `controllerProperties`.
`controllerActiveT` *val*	Activates temperature controller and reads `controllerProperties`.

syntax	description
`controllerActivePrecursorT` *val*	Activates temperature controller in concurrent-precursor and reads `controllerProperties`.
`perturbations` *val*	Adds sinusoidal perturbations to trigger turbulence if set to 1 when initial condition is `ABLFlow`.

`controllerProperties`
`{`
` relaxPI` *val*
` controllerMaxHeight` *val*
` controllerType` *type*
` alphaPI` *val*
` timeWindowPI` *val*
` geostrophicDamping` *val*
` geoDampingAlpha` *val*
` geoDampingStartTime` *val*
` geoDampingTimeWindow` *val*
` hGeo` *val*
` alphaGeo` *val*
` uGeoMag` *val*
` controllerAvgStartTime` *val*
`}`
	Contains inputs for velocity and temperature controllers. Required when `controllerActive`, `controllerActiveT` or `controllerActivePrecursorT` are set to 1; `controllerType` can be `pressure`, `geostrophic`, or `average`. Type `average`, used for wind farm simulations, averages sources history contained in `inflowDatabase/momtumSource` for $t >$ `controllerAvgStartTime`. Type `pressure` uses geostrophic damping when `geostrophicDamping` is 1. Type `geostrophic` requires level `hGeo` to sample velocity, initial geostrophic angle `alphaGeo` and desired G `uGeoMag`.

`yDampingProperties`
`{`
` yDampingStart` *val*
` yDampingEnd` *val*
` yDampingDelta` *val*
` yDampingAlpha` *val*
`}`
	Place-holder for the lateral fringe region, currently not implemented.

`xDampingProperties`
`{`
` xDampingStart` *val*
` xDampingEnd` *val*
` xDampingDelta` *val*
` xDampingAlpha` *val*
` xDampingAlphaControlType` *type*
` xDampingLineSamplingYmin` *val*
` xDampingLineSamplingYmax` *val*
` xDampingTimeWindow` *val*
` uBarSelectionType` *val*
` parameters...`
`}`
	Defines fringe region parameters, activated with `-xDampingLayer` 1 in *control.dat*; `xDampingAl- phaControlType` can be `alphaFixed` (constant damping coeff. `xDampingAlpha`) or `optimized` (requires `xDampingLineSamplingYmin`, `xDampingLineSamplingYmax` and `xDampingTimeWindow`). The way TOSCA computes the source term in the fringe region is selected with `uBarSelectionType` and can be set to 1,2,3, 4. Type 3 corresponds to the concurrent-precursor method. Each type with relative parameters is described in Table 6.

`zDampingProperties`
`{`
` zDampingStart` *val*
` zDampingEnd` *val*
` zDampingAlpha` *val*
` zDampingAlsoXY` *val*
` zDampingXYType` *type*
`}`
	Defines the input parameters for the Rayleigh damping layer and requires `-zDampingLayer` 1 in `control.dat`. If `zDampingAlsoXY` is set to 1 also horizontal velocity components are damped, `zDampingXYType` can be set to 1 (desired velocity is averaged at the inlet cells) or 2 (desired velocity is averaged from concurrent-precursor, requires `-xDampingLayer` 1 and `uBarSelectionType` 3).

syntax	description

canopyProperties
{
xStartCanopy *val*
xEndCanopy *val*
yStartCanopy *val*
yEndCanopy *val*
zStartCanopy *val*
zEndCanopy *val*
cftCanopy *val*
diskDirCanopy
(val_x,val_y,val_z)
}
	Defines input parameters for the canopy model.
	Requires -canopy 1 in *control.dat*.

advectionDampingProperties
{
advDampingStart *val*
advDampingEnd *val*
advDampingDeltaStart *val*
advDampingDeltaEnd *val*
}
	Parameters used to set the advection damping method of Lanzilao and Meyers (2022a).
	Requires -advectionDamping 1 in *control.dat*.

kLeftDampingProperties
{
kLeftPatchDist *val*
kLeftDampingAlpha *val*
kLeftDampingUBar
(val_x,val_y,val_z)
kLeftFilterHeight *val*
kLeftFilterWidth *val*
}
	Defines Rayleigh damping layer at the *kLeft* patch.
	Requires -kLeftRayleigh 1 in *control.dat*.
	Damping transitions from zero to max across a layer of width kLeftFilterWidth centered at kLeftFilterHeight, and is applied between the *kLeft* patch and a plane at a distance kLeftPatchDist from the *kLeft* patch to obtain the desired velocity kLeftDampingUBar.

kRighDampingProperties
{
kRightPatchDist *val*
kRightDampingAlpha *val*
kRightDampingUBar
(val_x,val_y,val_z)
kRightFilterHeight *val*
kRightFilterWidth *val*
}
	Defines Rayleigh damping layer at the *kRight* patch.
	Requires -kRightRayleigh 1 in *control.dat*.
	Damping transitions from zero to max across a layer of width kRightFilterWidth centered at kRightFilterHeight, and is applied between the *kRight* patch and a plane at a distance kRightPatchDist from the *kRight* patch to obtain a desired velocity kRightDampingUBar.

Table 5. Available entries in TOSCA's ABLProperties.dat file.

syntax	description
type 0 - logarithmic profile	
directionU (val_x,val_y,val_z) hInversion *val* frictionU *val* kRough *val*	It is a Rayleigh damping layer that can be used with periodic boundary conditions. Applies inletFunction type 2 inside the fringe. Refer to Table 3 for the corresponding keywords.
type 1 - unsteady mapped streamwise constant	
n1Inflow *val* n2Inflow *val* n1Periods *val* n2Periods *val*	Applies inletFunction type 3, with n1Merge always active, at every x location inside the fringe. Refer to Table 3 for the corresponding keywords.

syntax	description
type 2 - unsteady interpolated streamwise constant	
`n1Inflow` *val* `n2Inflow` *val* `n1Periods` *val* `n2Periods` *val* `sourceType` *type* `cellWidth1` *val* `cellWidth2` *val*	Applies `inletFunction` type 4, with `n1Merge` always active, at every x location inside the fringe. Data is interpolated to the fringe ij grid. Refer to Table 3 for the corresponding keywords.
type 3 - concurrent-precursor method	
Creates a second instance of TOSCA that runs a precursor simulation within the fringe region. Data are exchanged without processor communication as domain decomposition partitions the concurrent precursor identically to the successor. Requires `-precursorSpinUp` 0,1,2 in `control.dat`; 0 reads from checkpoint and uses streamwise periodic boundaries, 1 initializes the precursor flow using `spreadInflow` and applies `inletFunction` type 4, 2 reads from checkpoint and applies `inletFunction` type 4. Type 1 should be always used first, then 0 or 2 can be selected.	
type 4 - Nieuwstadt (1983) model	
`directionU` (val_x,val_y,val_z) `hInversion` *val* `frictionU` *val* `kRough` *val*	Uses `inletFunction` type 5 throughout the fringe region. Refer to Table 3 for the corresponding keywords.

Table 6. Available `uBarSelectionType` for the fringe region method implemented in TOSCA.

As can be noticed from Table 6, activating the fringe region through `-xDamp- ingLayer` in the *control.dat* file, and setting `uBarSelectionType` to 3 in *ABLProperties.dat* is sufficient to enable the concurrent-precursor method within TOSCA. In fact, the problem definition for the concurrent-precursor domain is completely handled internally by the TOSCA code. Moreover, a specific domain decomposition is applied for any specific problem that allows to avoid any communication between different processors during the computation when evaluating the momentum source term in the successor domain.
Regarding the introduction of wind turbines within the simulation domain, this is activated by setting the `-windplant` to 1 in the *control.dat* file. If this is the case, TOSCA expects an additional directory `turbines` within the main case directory, where the wind farm layout, turbine information and control data is stored. The wind farm layout and the desired model used for each wind turbine (different models can be used for different turbines within the same simulation) are specified in the `windFarmProperties` file, which entries are summarized in Table 7.

syntax	description
`windFarmName` *type*	Name of the wind farm.
`arraySpecification` *type*	Currently only type `onebyone` is available.
`debug` *val*	Prints turbine information, involves more parallel communication.

`writeSettings`
`{`
`timeStart` *val*
`intervalType` *type*
`timeInterval` *val*
`}`
	Settings for wind turbine data acquisition. This is started after t = `timeStart`, data are written every `timeInterval` seconds or iterations, for `intervalType` set to `adjustableTime` or `timeStep`, respectively; `adjustableTime` requires `-adjustTimeStep` 1 in *control.dat*.

syntax	description

`turbineArray`
`{`
` turbine 1 ...`
` turbine 2 ...`
` ...`
` turbine N ...`
`}`
	Wind farm specification. Entries required for each turbine, i.e. `turbine 1` to `turbine N`, are specified in Table 8. N is the number of wind turbines in the farm.

Table 7. Available entries in TOSCA's `windFarmProperties` file.

syntax	description

`turbineID`
`(`
` turbineType` *type*
` turbineModel` *type*
` baseLocation`
(val_x, val_y, val_z)
` windFarmController` *val*
`)`
	Entries for each turbine in Table 7; `turbineType` file stores the wind turbine definition (one file can define more turbines); `turbineModel` can be `ALM`, `ADM`, `uniformADM` or `AFM`; `windFarmController` requires look-up tables for pitch or C_T inside `turbines/control` directory used for wind farm control applications.

Table 8. Turbine specification in TOSCA's `windFarmProperties` file.

As can be seen from Table 7, there are no limits on the number of wind turbines which can be defined in TOSCA. Specific parameters for each wind turbine are contained in the `turbineType` file (see Table 8). Turbines can be of the same type or consisting of many different types within a wind farm. In the former, only one definition file is required within the `turbines` directory, whereas for the latter a number of files equal to the number of wind turbine types within the farm is required. The name of the turbine definition file is decided by the user and its entries of the turbine definition file are listed in Table 9. If the wind turbine is equipped with generator torque, pitch and yaw controllers, additional information is required in the control definition which should be contained inside the `turbines/control` directory, together with the wind farm controller information if this is activated from Table 8.

syntax	description
`rTip` *val*	Rotor radius.
`rHub` *val*	Hub radius.
`hTower` *val*	Tower height.
`overHang` *val*	Nacelle overhang.
`precone` *val*	Blade precone.
`towerDir` (val_x, val_y, val_z)	Tower direction from base to top.
`rotorDir` (val_x, val_y, val_z)	Rotor direction facing wind.
`upTilt` *val*	Rotor up-tilt.
`includeTower` *val*	Activates tower model.
`includeNacelle` *val*	Activates nacelle model.
`nBlades` *val*	Number of blades.
`rotationDir` *val*	Rotation direction, either `cw` or `ccw`.
`nRadPts` *val*	Number of radial points. Ignored for AFM.
`nAziPts` *val*	Number of azimuthal points. Ignored for AFM, ALM and AALM.

syntax	description
`epsilon` *val*	Projection width for ALM, ADM and UADM models.
`epsilon_x` *val*	Projection width in x for AFM, chord multiplier for AALM x projection.
`epsilon_y` *val*	Projection width in y for AFM, thickness multiplier for AALM y projection.
`epsilon_z` *val*	Projection width in z for AFM, radial element multiplier for AALM z projection.
`initialOmega` *val*	Initial rotor rotation speed.
`projection` *type*	Required for ALM, `isotropic` coincides with ALM, `anisotropic` selects AALM.
`sampleType` *type*	Required for ALM, UADM and AFM. Can be `rotorDisk`, `momentumTheory` and `integral` for AFM; `rotorUpstream`, `givenVelocity` and `rotorDisk` for UADM; `rotorDisk` and `integral` for ALM.
`Ct` *val*	Turbine thrust coefficient. It coincides with C_T or C'_T based on `sampleType`; `rotorDisk` and `integral` should use C'_T; `momentumTheory` and `givenVelocity` should use C_T.
`Uref` *val*	Reference velocity for the wind turbine, used to compute thrust for the UADM when `sampleType` is `givenVelocity`. Used for turbine acquisition otherwise.

`towerData`
`{`
` Cd` *val*
` rBase` *val*
` rTop` *val*
` nLinPts` *val*
` epsilon` *val*
`}`
	Required if `includeTower` is set to 1. Defines tower properties used to compute the tower drag.

`nacelleData`
`{`
` Cd` *val*
` epsilon` *val*
`}`
	Required if `includeNacelle` is set to 1. Defines nacelle properties used to compute the nacelle drag.

`bladeData`
`{`
` (r_1 c_1 t_1 afID_1)`
` (... ... ... ...)`
` (r_N c_N t_N afID_N)`
`}`
	Look-up table of blade information. Each line corresponds to the radial location `r_i` and specifies chord `c_i`, twist `t_i`, and airfoil ID `afID_i` with reference to the `airfoils` list. AALM also requires blade thickness between `t_i` and `afID_i`.

`airfoils`
`{`
` airfoil_1`
` ...`
` airfoil_N`
`}`
	List of airfoils used along the blade. The row in the list corresponds to `afID_i` in the `bladeData` list. Each airfoil name requires a file in `turbines/airfoils` containing tables of α, C_l, C_d for the specific airfoil.

Table 9. Available entries in TOSCA's wind turbine definition file.

Notably, these are two different controllers, i.e. specific to the wind turbine and to the wind farm, respectively. Information relative to the wind turbine controller are not reported in

this manuscript, but can be consulted from TOSCA's example cases at the TOSCA's GitHub repository.