TAU-Python Interface Guide for Release 2010.1.0

March 18, 2010

Contents

NOTE: This is the 1st draft of this Guide and only the most commonly used TAU-Python functions and classes have been included.

1 TAU-Python Basic Functions

This section of the guide contains descriptions of the functions of the TAU-Code which may be called from a user-created Python script. As far as possible, the data that is required to be in memory for the function-call to work will be listed. Please note that this guide is a work in progress, as well as being TAU-Release dependent.
The function descriptions are based on the assumption that the user script has been launched from the command line using:

```
tau.py <user-script.py> <tau-para-file> <log-file>
```

1.1 General functions

These functions are in most if not all cases an interface to the TAU library functions. Many of these functions control the storage of data that can be exchanged between the modules of the TAU-Code.

Function	`tau_close_surface_buffer()`
Description	Closes the surface stream buffer, which contains surface solution data from the Solver.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	By closing the buffer, modules which use surface solution data (like Deformation) will be forced to read the data from file.

Function	`tau_free_dualgrid()`
Description	Free the dualgrid memory and set the pointer to NULL.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	This function should always be used at the end of a loop where the Preprocessor is in the loop, like with a Deformation or Adaptation loop.

Function	`tau_mpi_rank()`
Description	Gives the MPI rank (domain) number of the process.
Parameters	`none`
Returns	`int rank`: MPI rank number
Notes	-

Function	`tau_mpi_nranks()`
Description	Gives the number of MPI processes on which TAU is running.
Parameters	`none`
Returns	`int nranks`: Total number of MPI processes on success
Notes	-

Function	`tau_msg(msg)`
Description	Writes a string to the TAU log-file. Please note that the output from the Python print-command does not appear in the log-file.
Parameters	`string msg`: A string containing message to be written to the log-file.
Returns	`int status`: 0 on success
Notes	-

1.2 Preprocessor-specific functions

The following functions are specific to the Preprocessor module of TAU. They are listed in the order in which they appear in the `tau_python.i` interface file.

Function	`tau_prep_print_options(para_path)`
Description	Print information on implemented boundary treatments, grid metrics and agglomeration to the log-file.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	-

Function	`tau_prep_init_params(para_path)`
Description	Initialize all preprocessor-specific parameters.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	Can be called multiple times if parameters are updated.

Function	`tau_prep_init()`
Description	Initialize the dualgrid structures and store grid parameters.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Requires that the parameters have been initialized.

Function	`tau_prep_init_my_primgrid_memory()`
Description	Changes default parameters such that the Preprocessor can free the primary grid, if directed to do so.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_prep_free()`
Description	Frees Preprocessor boundary parameters and comments. Does not free the primary grid.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_prep_compute()`
Description	Creates the dualgrid for the Solver. Tests the primary grid before creating the dualgrid, and makes a final check on the dualgrid.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Requires that the Preprocessor is fully initialized.

Function	`tau_prep_read_primgrid(para_path, field_io)`
Description	Reads in the primary grid. Will automatically check if grid needs to be partitioned.
Parameters	`string para_path`: path and name of TAU parameter file `int field_io`: flag to tell partitioner whether or not to write partitioned field data to disk
Returns	`int status`: 0 on success
Notes	If a field-output (restart) file is available it will also be partitioned and re-written to the disk. To suppress the re-writing of the field data to disk, set the field-io flag to zero.

Function	`tau_prep_write()`
Description	Writes all dualgrids to disk.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Requires a dualgrid in memory.

Function	`tau_prep_scatter(para_path)`
Description	Scatter files (grid and restart, if available) if running in parallel.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	This function is usually not called manually, since the Preprocessor will automatically scatter the files.

Function	`tau_prep_reset_wdist()`
Description	Recompute wall-distances, laminar regions and roughness (if transition or roughness settings have been changed).
Parameters	`none`
Returns	`int status`: 0 if nothing done, 1 if wall-distance re-computed
Notes	-

Function	`tau_prep_init_transition_params(trans_filename)`
Description	Initializes (re-initializes) all transition maps from data in the parameter file.
Parameters	`string trans_filename`: name of file containing transition parameters
Returns	`int status`: 0 if nothing done, 1 if transition parameters initialized
Notes	-

1.3 Solver-specific functions

The following functions are specific to the Solver module of TAU. They are listed in the order in which they appear in the `tau_python.i` interface file.

Function	`tau_solver_alloc_prims(ntime_level)`
Description	Allocate memory for the primitive variables.
Parameters	`int ntime_level`: number of time levels to allocate (default = 1)
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if memory is freed in between. Requires a dualgrid pointer.

Function	`tau_solver_init_params(para_path)`
Description	Initialize all solver-specific parameters, which include basic, grid-metric, unsteady and boundary parameters.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if parameters are updated. A dualgrid pointer (or file) must be available.

Function	`tau_solver_init_settings()`
Description	Initialize the solver based on the settings given by the parameters that have been read (from file or buffer).
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if parameters are updated. Requires a dualgrid pointer and that all parameters have been initialized.

Function	`tau_solver_initialize_prims()`
Description	Initialize the primitive variables (from scratch, file or buffer).
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Called during initialization. Requires that memory has been allocated for the primitive variables. Can be called multiple times.

Function	`tau_solver_prepare_prims()`
Description	Initialize solver data like bdry-conditions-to-prims, initial-cons-values, and extra init values. If required, also initialize Harmonics and APSIM data.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Called during initialization. Requires that memory has been allocated for the primitive variables. Can be called multiple times.

Function	`tau_solver_init_chimera(para_path)`
Description	Check if chimera boundaries exist, if they do then initialize the chimera data structures.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if parameters are updated. Requires a dualgrid pointer.

Function	`tau_solver_init_linvars(stream_linear)`
Description	-
Parameters	`string stream_linear`: name of a TAU stream
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_start(para_path)`
Description	Can be used to initialize the Solver from scratch or restart after the Solver has been stopped.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	Requires a dualgrid pointer and that Solver memory is free (not allocated).

Function	`tau_solver_stop(field_var)`
Description	Can be used to stop the Solver and free all Solver memory. Will buffer the required restart variables and any extra field variables, if specified.
Parameters	`string field_vars`: a string of extra field variables to be stored in buffer
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_buffer_field_stop(buffer_all, field_var)`
Description	Store Solver field variables in a stream buffer, which can be exchanged with other modules.
Parameters	`int buffer_all`: True - buffer all variables, False - only variables needed for restart `string field_vars`: a string of extra field variables to be stored in buffer
Returns	`int status`: 0 on success
Notes	The existing Solver memory will be freed. The stream name is hardcoded as 'field_output_stream'.

Function	`tau_solver_get_filename()`
Description	The name of the current output filename, including all iteration counters and time stamps, will be returned.
Parameters	`none`
Returns	`string pval_filename`: the name of the current pval file
Notes	-

Function	`tau_solver_free()`
Description	Frees all memory that is private to the Solver. Data like the primary grid, dualgrid or buffered solutions are not freed, since they can be used by other modules.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_get_dualgrid(para_path)`
Description	Check if dualgrid is available in memory, if not then read it in from file.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times, and should always be called when a new dualgrid has been created. Requires either a dualgrid pointer or file.

Function	`tau_solver_inner_loop(inner_step_start, inner_step_stop, min_residual)`
Description	For steady simulations a single iteration of this loop is a full multigrid cycle of the simulation. For unsteady simulations a single iteration of this loop is a full inner iteration (a pseudo time step). The arguments of the function can be set up such that control is returned to Python after a single iteration, or after the full number of iterations have been done.
Parameters	`int inner_step_start`: iteration at which the loop starts (default = 0) `int inner_step_stop`: iteration at which the loop stops (default = n_inner_steps) `double min_residual`: residual at which convergence is assumed (default = min_residual)
Returns	`int converged`: 0 if not converged
Notes	The solver should be fully initialized before this function is called.

Function	`tau_solver_inner_loop_iteration_number()`
Description	For unsteady simulations the function returns the number of inner iteration time steps given by the user. For steady simulations the number of maximum iterations is returned.
Parameters	`none`
Returns	`int n_inner_steps`: number of iterations for the solver inner loop
Notes	In a Python control script the return value can be used in two ways: How often the inner loop is called How many iterations the inner loop completes before returning to the script

Function	`tau_solver_min_residual()`
Description	The residual value at which the solver inner loop is assumed to be converged (assuming Cauchy convergence control is not active)
Parameters	`none`
Returns	`double min_residual`: minimum residual for convergence criteria `(default = 1.0e-16)`
Notes	-

Function	`tau_solver_outer_loop_init()`
Description	Prints the value of the initial residual (from a restart file) to the log-file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_outer_loop_iteration_number()`
Description	For unsteady simulations the function returns the number of physical time steps given by the user. For steady simulations the default value of 1 is returned.
Parameters	`none`
Returns	`int n_time_steps`: number of iterations for the solver outer loop (default = 1)
Notes	-

Function	`tau_solver_unsteady_advance_time()`
Description	For unsteady simulations, increments the physical time. In the case of global time step scheme it updates the eddy viscosity and gradients of the primitive variables.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called. Can be called multiple times. Each call increments the physical time value by the given time step increment.

Function	`tau_solver_unsteady_advance_motion()`
Description	For unsteady simulations, launches the internal calculation of the motion matrices (if external motion is not active). Launches chimera hole-cutting and search, and extrapolation of solution to new time level.
Parameters	`none`
Returns	-
Notes	The solver should be fully initialized before this function is called. Effects of multiple calls within a physical time step are currently unknown.

Function	`tau_solver_unsteady_get_physical_time()`
Description	The function returns the actual physical time at which the Solver is at that moment.
Parameters	`none`
Returns	`double physical_time`: the actual physical time of an unsteady simulation
Notes	This information can be used by external motion modules to determine the motion for this physical time level.

Function	`tau_solver_unsteady_monitoring()`
Description	Prints monitoring line for the end of a physical time step to the log-file, and increases the physical time step iteration count
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called.

Function	`tau_solver_unsteady_update()`
Description	Launches the Solver internal checks of solution output criteria. If the criteria fit, a field-, surface- or cutplane-output is done, and the parameter file is updated.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called.

Function	`tau_solver_unsteady_store_harmonics()`
Description	-
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_full_multigrid()`
Description	Does a full multigrid start for the solver (simulation starts on the coarse dualgrids).
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called. Can be called multiple times (side effects, if any, are currently unknown).

Function	`tau_solver_received_quit_signal()`
Description	Used to check if TAU has received a signal to quit.
Parameters	`none`
Returns	`int status`: 0 if no signal received
Notes	-

Function	`tau_solver_output_only()`
Description	Check if user only wants to have TAU write cut-plane or surface solution file(s).
Parameters	`none`
Returns	`int status`: 0 if false (default)
Notes	-

Function	`tau_solver_output_field()`
Description	Writes a field solution file and updates the parameter file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called.

Function	`tau_solver_output_surface()`
Description	Writes a surface solution file and updates the parameter file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called.

Function	`tau_solver_buffer_surface()`
Description	Buffers all surface solution variables in a stream buffer.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The stream name is hardcoded as 'surface_output_stream'.

Function	`tau_solver_output_cutplane()`
Description	Writes a cut-plane solution file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called.

Function	`tau_solver_output_monitor()`
Description	Finalizes the output of the monitor history file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	By calling this function, the '.tmp' post-fix will be removed from the filename.

Function	`tau_solver_inner_iteration_field_output()`
Description	-
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_inner_iteration_surface_output()`
Description	-
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_inner_restart()`
Description	-
Parameters	`none`
Returns	`int status`: 0 - False, 1 - True
Notes	-

Function	`tau_solver_inner_offset()`
Description	-
Parameters	`none`
Returns	`int offset`: -
Notes	-

Function	`tau_solver_print_init(para_path)`
Description	Initialize the output level to be used for log-file output.
Parameters	`string para_path`: path and name of TAU parameter file
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if parameters are updated.

Function	`tau_solver_print_signals()`
Description	Print information regarding signal handling to the Solver part of the TAU logfile.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if parameters are updated.

Function	`tau_solver_print_options()`
Description	Print information on most of the implemented options available for the chosen Solver version.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	Called during initialization. Can be called multiple times if parameters are updated.

Function	`tau_solver_print_integrals()`
Description	Prints out a table of aerodynamic forces and moments for each boundary part to the log-file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	The solver should be fully initialized before this function is called. The function can be called at any time during a simulation (inner or outer loop).

Function	`tau_solver_print_mainlooptime()`
Description	Prints the time spent in the main Solver loop to the log-file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_print_time_exchange_points()`
Description	Prints the time spent in exchanging point-data to the log-file.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_print_elapsed_time()`
Description	Prints elapsed physical time steps since the function was last called.
Parameters	`none`
Returns	-
Notes	-

Function	`tau_solver_print_unconverged_loops()`
Description	For unsteady dual-time simulations, prints out the number of unconverged outer loops.
Parameters	`none`
Returns	`int status`: 0 on success
Notes	-

Function	`tau_solver_get_integrals(part_name)`
Description	Returns a Python-list like string [Fx, Fy, Fz, Mx, My, Mz] of the forces and moments for the given part-name.
Parameters	`string part_name`: name of boundary mapping part
Returns	`string integral_list`: a Python list containing integral forces and moments
Notes	The moments are calculated around the origin as specified in the parameter file, or in the case of an unsteady simulation with motion, around the origin of the main motion node.

Function	`tau_solver_update_external_farfield_conditions()`
Description	If updates to the farfield conditions (velocity, density, pressure) have been sent to the motion module of TAU, this function can be used to update the farfield state of the Solver.
Parameters	`none`
Returns	-
Notes	The effects of using this function have not been fully validated / verified.

Function	`tau_solver_update_external_motion_farfield()`
Description	If external motion parameters have been sent to the TAU motion module, the changes in the direction of the farfield velocity vector are updated by calling this function.
Parameters	`none`
Returns	-
Notes	-

Function	`tau_solver_update_external_motion_whirlflux()`
Description	If external motion parameters have been sent to the TAU motion module, the extra source terms for the flux calculations are updated by calling this function.
Parameters	`none`
Returns	-
Notes	-

Function	`tau_solver_chimera_search()`
Description	Allows the chimera search function to be explicitly started from Python.
Parameters	`none`
Returns	-
Notes	A dualgrid pointer is required.

2 TAU-Python Basic Classes

This section of the guide contains descriptions of the TAU-Python classes which may be imported and used in a user-created Python script. As far as possible, the options for the public methods of the classes will be described. Please note that this guide is a work in progress, as well as being TAU-Release dependent.
The class descriptions are based on the assumption that the user script has been launched from the command line using:

```
tau.py <user-script.py> <tau-para-file> <log-file>
```

2.1 Class PySolv.py

This class encapsulates the standard TAU Solver functions in methods which clearly mark the separate steps the Solver handles, like initialization, iteration loops and output.
Before the class can be used it has to be imported:

```
import PySolv
```

and an instance of the class has to be created:

```
Solver = PySolv.Solver(para_path, cluster=None)
```

Here the arguments serve the following function:

- `para_path`: path and name of TAU parameter file
- `cluster=`: an instance of the PyCopyCluster class (default=None)

If the PyCopyCluster class is not being used, the instantiation can be done using:

```
Solver = PySolv.Solver(para_path)
```

Once instantiated, the functions of the class can be called in the following manner:

```
Solver.function_name(arguments)
```

The main functions of the PySolv class are the following:

Function	`init(cmd, verbose, n_time_steps, n_inner_cycles, n_inner_steps, online_monitoring)`
Description	Initializes the Solver based on the given arguments and the TAU parameter file.
Parameters	`cmd`: tells init what to do. Can be one of 'all', 'para', or 'para_update' (default='all') `verbose`: switch to turn writing of available parameters to log-file on or off (default=-1) `n_time_steps`: number of iterations for the solver outer loop (default=-1) `n_inner_cycles`: number of times the inner loop is called (default=-1) `n_inner_steps`: number of inner iterations before control returns to the Python script (default=-1) `online_monitoring`: a dictionary with the name of the program to use for online monitoring (default=empty)
Returns	-
Notes	The initialization is very much dependent on what the cmd= value is, since a lot of the logic checks what is already initialized and what still needs to be initialized: cmd='all' initializes everything that hasn't been initialized before, cmd='para' initializes only the parameters, cmd='para_update' re-initializes the parameter settings. If the default values for the time-steps / inner-cycles / inner-steps parameters are not overwritten by the user, they will be overwritten based on what is in the TAU parameter file.

Function	`inner_loop()`
Description	The function performs n_inner_cycles of n_inner_steps for the steady state convergence in the inner loop.
Parameters	`none`
Returns	`int converged`: 0 if not converged
Notes	-

Function	`outer_loop()`
Description	The function performs the physical time-stepping outer loop.
Parameters	`none`
Returns	-
Notes	This function will call the inner_loop() function in a loop over n_time_steps. Time and motion advancement are also taken care of in this function.

Function	`output()`
Description	Take care of output of field, surface, cut-plane and monitor files.
Parameters	`none`
Returns	-
Notes	-

Function	`stop(buffer_all, field_var)`
Description	Stops the Solver and frees the Solver memory, but buffers the field data in stream memory.
Parameters	`buffer_all`: boolean False / True (default=False) `field_var`: a string specifying extra field variables to store (default=None)
Returns	-
Notes	When buffer_all=True, all field data will be buffered, when False only data required for restart will be buffered. Extra field variables can be flagged for output using the field_var argument (currently only used/tested for frequency-domain solver).

Function	`finalize()`
Description	Free all private Solver memory.
Parameters	`none`
Returns	-
Notes	Data like the primary grid, dualgrid or buffered solutions are not freed.

Function	`run(verbose)`
Description	Runs the complete Solver without any type of interactivity through Python.
Parameters	`verbose`: switch to turn writing of available parameters to log-file on or off (default=1)
Returns	-
Notes	Using this function is almost the same as launching the Solver in stand-alone mode.

2.2 Class PyPrep.py

This class encapsulates the standard TAU Preprocessor functions in methods which clearly mark the separate steps the Preprocessor handles, like initialization, computation and output. Before the class can be used it has to be imported:

```
import PyPrep
```

and an instance of the class has to be created:

```
Prep = PyPrep.Preprocessing(para_path)
```

Here the argument serves the following function:

- `para_path`: path and name of TAU parameter file

Once instantiated, the functions of the class can be called in the following manner:

```
Prep.function_name(arguments)
```

The main functions of the PyPrep class are the following:

Function	`init(cmd, verbose, write_dualgrid, free_primgrid, field_io)`
Description	Initializes the Preprocessor based on the given arguments and the TAU parameter file.
Parameters	`cmd`: tells init what to do. Can be one of 'all', 'para', or 'para_update' (default='all') `verbose`: switch to turn writing of available parameters to log-file on or off (default=-1) `write_dualgrid`: switch to turn writing of dualgrid to file on or off (default=-1) `free_primgrid`: switch to free primary grid memory or not (default=-1) `field_io`: switch to turn writing of partitioned restart files on or off (default=-1)
Returns	-
Notes	The initialization is very much dependent on what the cmd= value is, since a lot of the logic checks what is already initialized and what still needs to be initialized: cmd='all' initializes everything that hasn't been initialized before, cmd='para' initializes only the parameters, cmd='para_update' re-initializes the parameter settings. If the default values for the write_dualgrid / free_primgrid / field_io parameters are not overwritten by the user, the default behavior is to write the dualgrid to file, free the primary grid memory, and not write re-partitioned restart files to disk.

Function	`compute()`
Description	The function launches the computation of a dualgrid.
Parameters	`none`
Returns	-
Notes	-

Function	`output()`
Description	The function writes a dualgrid to file.
Parameters	`none`
Returns	-
Notes	If write_dualgrid=0 the function-call will simply return without writing the dualgrid to file.

Function	`finalize()`
Description	Free all private Preprocessor memory.
Parameters	`none`
Returns	-
Notes	-

Function	`run(verbose, write_dualgrid, free_primgrid)`
Description	Runs the complete Preprocessor without any type of interactivity through Python.
Parameters	`verbose`: switch to turn writing of available parameters to log-file on or off (default=-1) `write_dualgrid`: switch to turn writing of dualgrid to file on or off (default=-1) `free_primgrid`: switch to free primary grid memory or not (default=-1)
Returns	-
Notes	Using this function is almost the same as launching the Preprocessor in stand-alone mode.

2.3 Class PyPara.py

This class encapsulates and extends the parameter handling for all modules of the TAU-Code. The class can be used to query and change parameters.
Before the class can be used it has to be imported:

```
import PyPara
```

and an instance of the class has to be created:

```
Para = PyPara.Parafile(para_path)
```

Here the argument serves the following function:

- `para_path`: path and name of TAU parameter file

Once instantiated, the functions of the class can be called in the following manner:

`Para.function_name(arguments)`

The main functions of the PyPara class are the following:

Function	`print_para()`
Description	Print the initial parameter value (without updates).
Parameters	`none`
Returns	`-`
Notes	-

Function	`get_para_value(key, format)`
Description	The function returns the value(s) of the given TAU general parameter. Block parameters are not returned by this function.
Parameters	`key`: the TAU parameter keyword `format`: return the value(s) of the parameter in the given format (default='string')
Returns	`the value(s) of the parameter in the specified format.`
Notes	The format of the return value(s) can be as a string of all entries after the ':' separator (format='string'), or the first entry of the values, which is useful when getting filenames without any trailing comments (format='single_key'), or as a Python list containing the separated entries (format='string_list').

Function	`get_para_filename(key)`
Description	The function simply returns the parameter filename for the given keyword.
Parameters	`key`: the TAU parameter keyword for the filename in question
Returns	`filename`: the filename of the parameter file specified by the given keyword
Notes	This function is simply a wrapper for the call get_para_value(key, format='single_key').

Function	`get_block_para_value(key, block_number, block_key)`
Description	The function simply returns the value of a specific block parameter, for a given block.
Parameters	`key`: the TAU parameter keyword `block_number`: number of the block in which the parameter is `block_key`: the keyword used to denote the end of the block (default='block end')
Returns	`The value of the given parameter, as a string.`
Notes	-

Function	`update(para_dict, block_key, block_id, key, key_value, sub_file)`
Description	This function can be used to update both general and block parameters in the TAU parameter file and buffer.
Parameters	`para_dict`: a Python dictionary of TAU parameter keywords and values `block_key`: keyword used to denote the end of a block (default='none') `block_id`: identification number of block to be updated (default=-1) `key`: a TAU block-parameter keyword (default='none') `key_value`: value of a block parameter (default='none') `sub_file`: name of sub-parameter file, like bmap or motion file (default=None)
Returns	-
Notes	The default setup of this function is to update general parameters in the main TAU parameter file / buffer, based on the keyword-value pairs found in the 'para_dict' argument. The parameter buffer and file are always updated when this function is used, but the changes only take effect in the TAU modules if the update takes place before the modules are initialized, or by using the 'para_update' command when re-initializing the TAU module (e.g. Solver). If no TAU module has been run prior to calling this function, no buffer exists to store the updated parameters, but they will be updated in the parameter file regardless.

3 TAU-Python Use Cases

This section of the guide contains some of the more common use-cases for which TAU-Python offers a definitive advantage over TAU stand-alone.
The descriptions of the use-cases are based on the assumption that the simulation has been launched using submit_taujob.py and a simspec file for the given case:

```
submit_taujob.py <simspec-file>
```

3.1 Use Case: Simple Polar

For this case we want to get a flow solution for 3 specific angles of attack. To minimize file-IO, no dualgrid will be written to disk, and no restart-files will be read in from disk. The script, as presented here, is broken into sections with detailed descriptions of what is taking place; a similar script, script_polar.py, is a part of the TAU Release.

First we import all of the TAU-Python specific modules which we intend to use. We need PyPara to update the parameters for the given polar points. PySolv and PyPrep for the solver and preprocessor, obviously, and PyCopyCluster for file handling, such that porting our simulation from PC to a cluster will be a snap.

```
# import TAU-Python modules
import PyPara
import PySolv
import PyPrep
import PyCopyCluster
```

We want to initialize the PyCopyCluster class first, since a lot of the file-handling that is taken care of automatically is controlled by this class, like partitioning the primary grid and setting up the recommended directory structure.

```
# create an instance of the PyCopyCluster class and initialize it
CopyCluster = PyCopyCluster.CopyCluster(para_path)
CopyCluster.init()
```

Now we create instances of the remaining classes. The Solver class has the CopyCluster instance as a parameter, since we want the output from the Solver to be automatically moved to the pre-defined directories.

```
# create instances of the remaining classes
Para   = PyPara.Parafile(para_path)
Prep   = PyPrep.Preprocessing(para_path)
Solver = PySolv.Solver(para_path, cluster=CopyCluster)
```

To set up our polar simulation we define here the parameters we want to update, in this case the angle-of-attack and the prefix of the solution-file.

```
# define parameters that will be updated during the simulation
# we want to change angle-of-attack and use the angle as part of
# the output filename
alpha_list = [2.0, 4.0, 8.0]
outputprefix = "solution"
```

Since we will be updating the same parameters multiple times from within a loop, it makes sense to create a function which handles the required updates and which we can simply call from within the loop. The function returns Python dictionaries with the parameter updates.

```
# create a help-function where parameter dictionaries are created
def get_para_bmap(alpha, outputprefix)
  """
  return dictionaries containing keywords and values for
  para_list and bmap_list which have to be updated
  """

  suffix = "Alpha=" + str(alpha)

  # dictionary for parameter update
  para_list = {}
  para_list['Output files prefix'] = outputprefix + "_" + suffix

  # dictionary for bmap-parameter update
  bmap_list = {}
  bmap_list['Angle alpha (degree)'] = alpha

  return para_list, bmap_list
```

Now that we have everything we need for the simulation, we can start the actual TAU computation. The first step is to compute the dualgrid, and this we can do outside the polar-loop, since the grid does not change during the course of the simulation. We also specify that we do not want the dualgrids written to file, since we will take advantage of the in-memory data-transfer available with TAU-Python.

```
# create dualgrid, do not write it to disk
Prep.run(write_dualgrid=0)
```

We initialize the Solver once outside the polar loop; with this initialization, memory is allocated and parameter settings are parsed from the parameter file.

```
# initialize the solver
Solver.init()
```

Now we start the actual polar loop. The TAU parameters are updated, and the Solver is re-initialized to account for those changes.

```
# loop over all angles of attack, updating parameters as needed
for angle in alpha_list:

  # call helper-function to get parameter changes
  para, bmap = get_para_bmap(angle, outputprefix)

  # update general parameters, then bmap block-parameters
  Para.update(para)
  Para.update(bmap, block_key='block end', key='Type', key_value='farfield')

  # re-init solver to activate changed parameter input, and run simulation
  Solver.init(cmd='para_update', verbose=0)
  Solver.outer_loop()
  Solver.output()

# free solver memory / data
Solver.finalize()

# exit python
tau("exit")
```