A3PI User Guide

David Bizzozero

May 31, 2021
v. 0.5

Contents

1 Introduction

The A3PI workflow management tool is built in Python and is designed to streamline the integration of ACE3P, specifically the Omega3P sub-module of ACE3P, with Impact-T for coupled cavity design with beam dynamics studies. The workflow is set up with a single configuration file which contains all necessary step instructions, paths, and input parameters for the constituent codes.

1.1 Workflow Input Overview

The A3PI Workflow is designed with the Python ConfigParser class and its extended interpolation option. The ConfigParser class lets Python manage changes to a given workflow input file before sending it to workers (e.g. CPU nodes) for evaluation. The extended interpolation option allows the ConfigParser to copy values which are changed (e.g. physical length of a component) to all appropriate places (e.g. cubit geometry, Acdtool postprocess routine, Impact-T input file, etc.).

The workflow configuration file is generally long, but it combines all necessary inputs for each code in a single place. In this section, we will overview an example workflow input file and explain each keyword section. Each keyword section must be surrounded by square brackets `[]`. Within a given section, keyword entry pairs must be separated by an equals `=` sign. If more than one value is to be assigned, then the arguments for the keywords should be separated by a comma `,` (spaces work for some sections too). Comments may be made using the number symbol `#`.

Folder structure with A3PI is handled as follows: the workflow configuration file is located in the *base* folder. When this file is parsed by A3PI, it will accordingly generate workflow *worker* folders in which a copy of the workflow configuration file is placed, with corresponding adjustments to desired variables. To clarify, the workflow configuration file is *unique* for each use of A3PI, but when evaluating multiple workflows (e.g. for optimization), each workflow *worker* folder will have its own copy of the *worker* configuration file but with differences in the variable section `[VARS]` and a few other settings.

1.1.1 Common Workflow Sections

The `[VARS]` section is used to store all variables which are expected to change for multiple workflow runs. For the example below, the variable `dz` is set initially to `0.0`. However, during optimization, this value can be changed and the numerical value of `dz` will be automatically

inserted elsewhere in the workflow accordingly. **Note: variable names in this example are not built-in and can be added, changed, or removed as desired; the only requirement is that the variable names are consistent throughout the file.**

```
[VARS]
#Define the various paramters that are updated each workflow

dz = 0.0
sig_xy = 3.0e-4
sig_z = 6.0e-5
sol_str = 0.150
gun_scale = 2.00e6
gun_freq_shift = 1.98e6
gun_freq = 2.00e8
gun_phase = 161.0
gun_fid = 300
b1_phase = 301.0
b2_phase = 301.0
b3_phase = 301.0
b4_phase = 301.0
```

Next, the `[PATHS]` section defines all the code paths needed for the workflow to run. Each code can have a different path but should all be accessible from the workflow input file's directory (absolute path names are preferred). The `workflow_path` is special in that it defines the folder to create for a given workflow.

The idea is that A3PI is run from a base directory with this workflow configuration file. Next, this configuration file is interpolated with the appropriate numeric replacements, such as `dz` in the previous example section, and a new folder named `workflow_folder` is created. The interpolated workflow input file (which is now static) is placed into the new folder for evaluation by a worker (e.g. CPU node). When using LibEnsemble to run multiple concurrent workflow evaluations, each folder will have an automatically appended number unique for each worker (e.g. `workflow_folder_0`, `workflow_folder_1`, etc.).

```
[PATHS]
#Define paths to where the executables are found

cubit_path = mypath/cubit
acdtool_path = mypath/acdtool
omega3p_path = mypath/omega3p
impactt_path = mypath/ImpactTexe_knl_gnu
workflow_path = workflow_folder
```

The next section includes special platform-specific run options such as how many cores to use for Acdtool postprocess, Impact-T, and various input data files. The `run_cmd` value is the string used to call the code, and if omitted, it the workflow will try to call the code via its path and not using a job manager (e.g. SLURM on NERSC systems). The `static_files`

list includes all necessary data files needed to run the workflow. For example, this list can include the base geometry files for Cubit, static field maps for Impact-T, or any other files that need to be copied into the `workflow_folder`. **These static files must be in the base directory with the workflow configuration file.**

```
[RUN_PARAMETERS]
#Define run command, HPC options, and list of static files needed for workflow

A3PI_mode = single
acdtool_cores = 32
impactt_cores = 64
static_files = rfdata1, rfdata2, cubit_geometry.jou, cubit_acis_file.sat
```

1.1.2 Cubit Workflow Section

To run Cubit, a geometry file must be provided with the `file` keyword. Since a workflow may need to compute meshes for multiple accelerator components, different Cubit geometries can be defined by separate sections `[CUBIT_ELEMENT_1]`, `[CUBIT_ELEMENT_2]`, and so forth. This numbering ID scheme will also be used in the tasks section at the end of the workflow configuration file.

The key and value pairs will be used to to overwrite the corresponding variables at the beginning of the journal `.jou` file. For example, this workflow would replace the numeric value of `dz` given in the `[VARS]` section when interpolating for use in a workflow run. Then when Cubit is called, the journal file is edited with the corresponding value of `dz` and the mesh is generated. **Careful consideration must be taken to ensure that the Cubit journal file can accept arbitrary values (within a specified range) for the key parameters.** Using incorrect syntax or missing a value of a key parameter may create an erroneous journal file which can cause Cubit to fail or generate a bad mesh.

The output filename can be arbitrary as the workflow will automatically know to use it with Acdtool meshconvert by its Cubit element ID number (in this example case `1`).

```
[CUBIT_ELEMENT_1]
#Define a Cubit element for meshing and use by Omega3P
#See Cubit user guide for more information

#file = journal file with geometry
file = moveCathode.jou

#Define the parameters in the journal file to update
keys = dz

#Define values to corresponding parameters in the journal file to update
values = ${VARS:dz}

#Define output meshfile in .gen format
output_file = SRF-GUN-cathR10mm.gen
```

1.1.3 Omega3P Workflow Section

The Omega3P input parameters require an input file with the `.in` suffix. Normally, a user would provide this file; however, in A3PI, this input file is generated automatically with the workflow. The entries in the Omega3P file are constructed from parameters with prefixes such as `ModelInfo`, `FiniteElement`, `EigenSolver`. The Omega3P input uses a similar keyword value system to the workflow configuration file, but it can allow for nested entries. Since the ConfigParser utility for Python does not allow for nested sections with the square brackets `[]`, a workaround was built to parse the boundary condition information, using spaces and commas.

The input filename should be different for each mode desired. The cores required should be consistent with the resources of the computing workers (e.g. a single KNL node on Cori has 68 physical cores). **Careful consideration of mesh sizes should be made when using large amounts of memory resources as to avoid out-of-memory errors**. The mode file ID `mode_fid` should match the same ID as used in Impact-T's naming convention; defining it as an integer from 1–999 in the `[VARS]` section will usually avoid conflicts. Lastly, the `ModelInfo_File` can be either a fixed mesh in `.ncdf` format or the name of a Cubit section. In the case of listing a Cubit section, the workflow will use the `output_file` given from that particular section.

```
[OMEGA3P_MODE_1]
#Define an Omega3P simulation for a specific mode
#See Omega3P user guide for more information

#file = input filename (file will be generated automatically)
file = o3p_1.in

#cores = number of cores needed for simulation
cores = 320

#Define mode file ID to be consistant with ImpactT
mode_fid = ${VARS:gun_fid}

#Define mesh filename from either a static .ncdf file or cubit_element_*
ModelInfo_File = cubit_element_1
ModelInfo_Tolerant = 1

#Set boundary conditions for surfaces (sep. bc types with a comma)
ModelInfo_BoundaryCondition = magnetic 1 2 5, electric 6

#Set finite element order and curved surfaces option
FiniteElement_Order = 2
FiniteElement_CurvedSurfaces = on

#Define eigensolver data and tolerances
EigenSolver_NumEigenvalues = 1
```

```
EigenSolver_FrequencyShift = ${VARS:gun_freq_shift}
```

1.1.4 Acdtool RF Postprocess Section

After Omega3P has been run, a new folder in the workflow folder, named by default `omega3p_results`, will be created. This results folder contains all the fields and mode data for the input geometry and parameters from the `OMEGA3P_MODE` section. Because Omega3P is designed to overwrite this output folder after each calculation, a user must run the Acdtool postprocess routine after each Omega3P step. The keyword entries in this section with the `RFField` and `OpenPMD_IMPACT` prefixes are used to generate the input file for Acdtool.

In summary, the Acdtool use in this step is to extract the field data in a small Cartesian grid near where a particle beam is expected to pass (typically centered about z-axis). The output from the Acdtool postprocess step in A3PI will be a set of openPMD `.hdf5` files containing field data for the desired region. These field data files will then either interpolated from, or directly imported into Impact-T. A more detailed overview is provided in the input files section.

As a final note, the Acdtool postprocess workflow step will use the existing `omega3p_results` folder for field data and cannot choose an older Omega3P run if the results have been overwritten. It is therefore advised to run this Acdtool postprocess step after each Omega3P step in the workflow.

```
[RFPOST_MODE_1]
#Define parameters for acdtool postprocess routine
#See acdtool user guide for more information

#file = input filename (file will be generated automatically)
file = rfpost_1.in

#Set RFField parameters
RFField_ResultDir = omega3p_results
RFField_FreqScanID = 0
RFField_ModelID = 0
RFField_xsymmetry = magnetic
RFField_ysymmetry = magnetic
RFField_gradient = -1
RFField_cavityBeta = 1.0
RFField_reversePowerFlow = 0
RFField_x0 = 0.0
RFField_y0 = 0.0
RFField_gz1 = -0.375
RFField_gz2 = 0.668
RFField_npoint = 1000
RFField_fmnx = 20
RFField_fmny = 20
RFField_fmnz = 100
```

```
#Set OpenPMD_IMPACT parameters
OpenPMD_IMPACT_ionoff = 1
OpenPMD_IMPACT_nx = 101
OpenPMD_IMPACT_ny = 101
OpenPMD_IMPACT_nz = 301
OpenPMD_IMPACT_x1 = -0.015
OpenPMD_IMPACT_x2 = 0.015
OpenPMD_IMPACT_y1 = -0.015
OpenPMD_IMPACT_y2 = 0.015
OpenPMD_IMPACT_z1 = 0.000
OpenPMD_IMPACT_z2 = 0.300
```

1.1.5 Interpolate Field Workflow Section

This specialized section is needed when a collection of previous field data from various Omega3P runs is to be used for quick interpolation. For example, if it is expected that the fields vary slowly with the cavity shape parameter `dz`, then one can run the (Cubit)→(Acdtool meshconvert)→(Omega3P)→(Acdtool postprocess) chain of steps for a few values of `dz`. This then can greatly speed up the workflow computation in subsequent runs by approximating the output of the chain via coordinate interpolation of the fields.

In this example with file ID 300, the field data is stored in 6 folders named `field_data_0`, `field_data_2`, ..., `field_data_10`. These folders are located in the base directory with the workflow configuration file and contain data files: `data300.h5`, `data301.h5`, `data302.h5`, `data303.h5` which would correspond to the 3D field data outputs from Acdtool postprocess `E_Real.h5`, `E_Imag.h5`, `B_Real.h5`, `B_Imag.h5` renamed respectively for the appropriate value of `dz`. Alternatively, if 1D on-axis field data pre-formatted for Impact-T is already available, the corresponding `rfdata300` file for each value of `dz` can be placed in each folder instead.

As of the current version of A3PI, only a single parameter may be interpolated (in the example below `dz`), but the interpolation spacing (precomputed values for `dz`) need not be uniform, but any requested value must lie between the maximum and minimum values (in this case example, `dz` must be between `0` and `10`). Thus, the interpolation step can emulate the effect of running the (Cubit)→(Acdtool meshconvert)→(Omega3P)→(Acdtool postprocess) chain for any value of `dz` with less effort. Furthermore, the actual field data for each sampled `dz` need not use the same exact Cartesian grid, but for best results the grids should not vary much in `dz`.

```
[INTERPOLATE_FIELD]
#Define field to be interpolated from existing field data and fids

#Folder names to find file ID
folder_name = 'field_data_?'
folder_vals = 0, 2, 4, 6, 8, 10
interp_var = ${VARS:dz}
```

1.1.6 Impact-T Workflow Section

The input parameters for Impact-T are separated into two sections: a beam parameters section, and a lattice section. In the beam parameters section `[IMPACTT_PARAMETERS]`, the 9 entries correspond to the first 9 rows in the Impact-T input file. The A3PI workflow will parse these entries and generate an automatic Impact-T input file to be run. The benefit of this automated file generation is to allow for the use of the variables stored in the `[VARS]` section to coordinate an entire workflow when changing various parameters.

Familiarity with Impact-T is recommended when using `[VARS]` variables to be interpolated with the ConfigParser. **Another restriction is that the size of the processor array `procs` (in this example 8×8) must be compatible with the `impactt_cores` value defined in the `[RUN_PARAMETERS]` section (in this example 64).**

```
[IMPACTT_PARAMETERS]
#Define the ImpactT input parameters (line numbers 1-9 here)
#See ImpactT user guide for more information

#processors = col row
procs = 8 8

#steps = dt Nstep Nbunch
steps = 4.0e-12 2000000 1

#parts = PSdim Npart integF errF diagF imchgF imgCutOff
parts = 6 5000 1 0 2 0 0.02

#mesh = Nx Ny Nz bcF Rx Ry Lz
mesh = 32 32 32 1 0.15 0.15 1.0e5

#dist = distType restartF substepF Nemission Temission
dist = 112 0 0 1300 6.5e-11

#*dist = sig* sigp* mu*p* *scale p*scale xmu* xmu*
xdist = ${VARS:sig_xy} 0.001 0.0 1.0 1.0 0.0 0.0
ydist = ${VARS:sig_xy} 0.001 0.0 1.0 1.0 0.0 0.0
zdist = ${VARS:sig_z} 0.0014 0.0 1.186189e-6 1.0 0.0 0.002

#beam = I/A Ek/eV Mc2/eV Q/e freq/Hz phs/rad
beam = 0.26 1.0 0.511005e6 -1.0 1.3e9 0.0
```

In the Impact-T lattice section `[IMPACTT_LATTICE]`, each line will correspond to a lattice line in the Impact-T input file. The keys with prefix `elem_` are not name-specific in the automated Impact-T input file generator, but each element must listed in the workflow configuration file in order. The `elem_` lines are helpful for the user when setting up the lattice, but a given element can be commented out with the `#` symbol without requiring a renumbering of the other elements. Lattice elements with a lot of text input, such as `elem_01` in the example, can be either a single long line, or separated into multiple indented lines.

```
[IMPACTT_LATTICE]
#Define the ImpactT lattice (line numbers 10+ here)
#See ImpactT user guide for more information

#Start 3D SC
elem_00 = 0.0 1 20 -5 -2.0 -2.0 -2.0

#SRF gun element
elem_01 = 0.3 105 20 111 0.0 ${VARS:gun_scale} ${VARS:gun_freq}
          ${VARS:gun_phase} ${VARS:gun_fid} 0.1 0.0 0.0 0.0 0.0

#Focusing solenoids
elem_02 = 0.48 105 20 105 0.250e0 0.0 0.0 0.0 5 0.1 0.0 0.0 0.0 0.0 0.0
          ${VARS:sol_str}

#Boosting cavities
elem_03 = 1.346 10 20 105 2.6702 3.00000e+07 1.3e9 ${VARS:b1_phase}
          400 0.1 0.0 0.0 0.0 0.0 0.0 0.0
elem_04 = 1.346 10 20 105 4.0538 3.00000e+07 1.3e9 ${VARS:b2_phase}
          400 0.1 0.0 0.0 0.0 0.0 0.0 0.0
elem_05 = 1.346 10 20 105 5.4374 3.00000e+07 1.3e9 ${VARS:b3_phase}
          400 0.1 0.0 0.0 0.0 0.0 0.0 0.0
elem_06 = 1.346 10 20 105 6.8210 3.00000e+07 1.3e9 ${VARS:b4_phase}
          400 0.1 0.0 0.0 0.0 0.0 0.0 0.0
```

***** Note on Impact-Z implementation *****

Impact-Z shares a similar file input file structure as Impact-T; however, due to the various features of A3PI, a full configuration parsing of an Impact-Z parameter and lattice section has not yet been implemented.

As a workaround, the `[RUN_PARAMETERS]` section can include an external input file with the keyword `impactz_input_file`, analogous to the procedure in defining a input file for Impact-T. This will then instruct A3PI to use the selected file when running Impact-Z (e.g. for example after an Impact-T run). However, since this file is externally supplied and not parsed, no keyword interpolation is possible for Impact-Z (e.g. for use in the context of optimization).

As in the case of Impact-T, be sure to **include any external field files needed for Impact-Z in the static files list** in the `[RUN_PARAMETERS]` section.

1.1.7 LibEnsemble Workflow Section

This section defines all the necessary parameters and settings for multi-objective optimization using libEnsemble with DEAP. A typical usage would be to optimize some parameters (e.g. `dz`, `sig_xy`, etc.) in the `[VARS]` section with respect to multiple objectives such as `xy_rms_emittance` and `z_rms_size` while remaining within some constraints on other parameters such as `beam_energy`. To accomplish this, we can set up the `[LIBENSEMBLE]`

section as follows.

We start by defining the `num_workers` value which should be at most 1 less than the number of nodes provided. This is because libEnsemble reserves one node for the managing thread. We can also set the number of `nodes_per_worker` here, but the default is 1. The current version of A3PI does not support dynamic node balancing yet which would allow workers to share nodes as needed if certain steps in a workflow require more resources than other steps.

For the genetic algorithm, the population size and maximum number of generations is specified. During the evolutionary process, the NSGA-II algorithm used by A3PI enforces a constant population to ensure nodes are used more efficiently.

Next, the input variables are listed as strings and must match the names of variables listed in the `[VARS]` section. This is to ensure the variables are properly linked and updated when the workflow is run. Each variable needs its own lower and upper bound in the parameter range; this is provided as a list which follows the order the variables were listed in the vars entry within the `[LIBENSEMBLE]` section. **Note: the number of input variables listed in the `vars` line must match the number of bounds provided.** In this example, the number of input variables is 9.

The objectives, are listed next in a similar form but should also include signed weights for the relative behavior desired for each parameter. In the example, both objectives are to be minimized so the weights are set to -1. **Note: the number of output quantities listed in the `objectives` line must match the number of `weights` provided.** In this example, the number of objectives is 2. The complete list of available objectives are given in the table below. Note, for the transverse xy options, denoted with a *, use the $||\cdot||_2$ norm of the respective x and y options to represent the average radial component.

Objective and Constraint List		
Name	—	**Description**
`beam_energy`	—	Beam energy [MeV] (column 4 in fort.18)
`x_rms_size`	—	Transverse x-size [m] (column 4 in fort.24)
`x_rms_emittance`	—	RMS x-emittance [m] (column 8 in fort.24)
`y_rms_size`	—	Transverse y-size [m] (column 4 in fort.25)
`y_rms_emittance`	—	RMS y-emittance [m] (column 8 in fort.25)
`xy_rms_size`	—	Transverse xy-size* [m]
`xy_rms_emittance`	—	RMS xy-emittance* [m]
`z_rms_size`	—	Longitudinal z-size [m] (column 3 in fort.26)
`z_rms_emittance`	—	RMS z-emittance [m] (column 7 in fort.26)

Optionally, constraints can be provided and A3PI uses a simple linear penalty factor for objective functions which a constraint parameter does not satisfy the bounds provided. In this example, if the `beam_energy`, which is determined automatically by reading a corresponding Impact-T output file, does not lie in the prescribed range, a penalty factor is applied. This penalty factor F is defined for constraints C_i with corresponding bounds $[C_{\min}, C_{\min}]$ and constraint penalty scaling terms P_i as:

$$F = 1 + \sum_i [\max\{C_{\min} - C_i, 0\} + \max\{C_i - C_{\max}, 0\}] \cdot P_i$$

Thus, for the example, if the `beam_energy` value is 66 MeV (default units from Impact-T), then the penalty factor $F = 1$ (no penalty). However, if the beam energy drops to 45 MeV, then the penalty factor $F = 6$. This is then applied to the objective function evaluations y (e.g. $y_1 =$ `xy_rms_emittance` and $y_2 =$ `z_rms_size`) by the formula:

$$y_j^{\text{penalized}} = y_j^{\text{original}} \cdot F^{-\text{sign}(y_j)w_j}$$

For minimization objectives, the weight factor w_j is negative and thus can be penalized by increasing y_j (or decreasing y_j if it is negative). Likewise, for maximization objectives, with $w_j > 0$, y_j can be penalized by decreasing it (or increasing y_j if it is negative). **Note: the number of constraint penalty quantities listed in the `const_penalty` line must match the number of `constraints` provided.** In this example, the number of objectives is 2.

Next, for the NSGA-II algorithm in DEAP, we provide the mutation crossover probability `cxpb`, mutation parameter `eta`, and independent mutation paramter `indpb`. These parameters are covered more in detail on the DEAP documentation.

Lastly, we can optionally define termination criteria for the optimization. Currently, only `sim_max` is supported by A3PI, but more options can be added in a future version.

```
[LIBENSEMBLE]
#Define the parameters needed for libEnsemble multi-worker evaluation (with MPI)
#This section also defines parameters for DEAP multi-objective optimization

#If workers set to auto, number of workers is MPI.COMM_WORLD size -1
num_workers = auto
nodes_per_worker = 1

#Define population size, max number of generations, input dimensions, objectives
pop_size = 128
num_gen = 50

#Define the inputs and their respective bounds (uniformly random initially)
#Variable names must be string names of variables defined in the [VARS] section
vars = dz, sig_xy, sig_z, sol_str, gun_phase,
       b1_phase, b2_phase, b3_phase, b4_phase
lower_bounds = 0.0, 4.0e-4, 2.5e-5, 0.140, 150.0,
               280.0, 280.0, 280.0, 280.0
upper_bounds = 10.0, 8.0e-4, 4.0e-5, 0.170, 170.0,
               310.0, 310.0, 310.0, 310.0

#Define outputs to optimize and their weights (-1 to minimize, +1 to maximize)
objectives = xy_rms_emittance, z_rms_size
weights = -1, -1

#Define constraints as penalty, if a constraint parameter is outside bounds, the
#objective functions are multiplied by a factor of dist(const,bound)*penalty
constraints = beam_energy
```

```
const_lower_bounds = 50
const_upper_bounds = 100
const_penalty = 1.0

#Define crossover probability, mutation parameter, and independent mutation prob.
cxpb = 0.8
eta = 20.0
indpb = 0.09

#Define optional exit criteria [default is sim_max = pop_size*(num_gen+1)]
sim_max = 6528
```

1.1.8 Tasks Workflow Section

This section defines the list of tasks to perform for a single workflow calculation, excluding libEnsemble management and optimization with the DEAP library. The allowable tasks are currently `cubit`, `meshconvert`, `omega3p`, `rfpost`, `interpolate_1d`, `interpolate_3d`, `impactt`, and `impactz`. The task numbers are not used but are useful for a user to organize a workflow; the A3PI workflow uses only the **order** they are listed in the `[WORKFLOW]` section. Thus, to skip a given step, a user can comment out that particular step simply with the `#` symbol.

The input arguments following the task name depend on the task. For `cubit`, `meshconvert`, `omega3p`, and `rfpost` tasks, the workflow will use the following numeric argument to evaluate the corresponding numbered section `[CUBIT_ELEMENT_]`, `[OMEGA3P_MODE_]`, `[RFPOST_MODE_]` respectively. In the case of `meshconvert`, Acdtool will use the mesh name used in corresponding `[CUBIT_ELEMENT_]` section.

For the interpolation tasks `interpolate_1d` and `interpolate_3d`, the numeric argument corresponds to the file ID used in the associated field data folders. For `interpolate_1d` with file ID 300, a pre-formatted Impact-T on-axis 1D field file named `rfdata300` must exist in each of the field data folders. For `interpolate_3d` with file ID 300, the openPMD field data files `data300.h5`, `data301.h5`, `data302.h5`, `data303.h5`, which are outputs from the `rfpost` step, must exist in each of the field data folders. The interpolation routines are intended to approximate the fields from a given `cubit`, `meshconvert`, `omega3p`, and `rfpost` chain of tasks; interpolation routines should not be used with the full task chain.

The numeric argument for the `impactt` routine corresponds to a desired precomputing task done before running Impact-T. The argument `1` runs a phase scanning routine to determine optimal phases for the lattice; in this mode, the existing phases in the automated Impact-T input file are treated as the *design phase*. Also, the phase scanning routine will assume all field data has been normalized to a peak value of 1.0. The argument `2` runs the same phase scanning routine without the normalization assumption, this is useful when adjusting the RF power for lattice elements in optimization. Argument options `1a` and `2a` will perform the same phase scanning routine as in the arguments `1` and `2` respectively, but will not run Impact-T following the phase scan (intended for debugging). Any other numeric argument corresponds to running Impact-T as-is with the automatically generated input file. Impact-T will be run after the optional preprocessing.

Lastly, for `impactz`, which calls Impact-Z with the input file defined in the `RUN_PARAMETERS` section, the argument `1` will read an existing `fort.50` file located in the workflow folder and create the file `particle.in`; a copy of `fort.50` with the particle number appended to the head of the file. This is intended for use cases where an injector simulation with Impact-T can be used to create the initial particle distribution for Impact-Z. Any other argument for `impactz` will simply skip the creation of the `particle.in` file and run Impact-Z as-is.

Note: the phase scanning routine is not compatible with libEnsemble or DEAP in the current A3PI version. Thus for multi-objective optimization, users cannot use `impactt` with the arguments `1`, `1a`, `2`, or `2a`. This is because the phase scanning routine uses the multiprocessing Python package to optimize phases efficiently, and depending on the computing environment, there may be compatibility issues with running nested MPI processes. Lastly, multiple field interpolation steps are not supported in this version of A3PI.

In the example below, a task chain of 5 tasks: `cubit`, `meshconvert`, `omega3p`, `rfpost`, and `impactt` are defined as the workflow. The `interpolate_1d` task has been commented out but the task numbers do not matter; the tasks are executed in the top-down sequential order they are listed in this section.

```
[WORKFLOW]
#Define sequence of simulations for this workflow

task_00 = cubit 1
task_01 = meshconvert 1
task_02 = omega3p 1
task_03 = rfpost 1
#task_04 = interpolate_1d ${VARS:gun_fid}
task_05 = impactt 3
```

1.2 Executables and Python Modules

The A3PI code requires executables for Cubit, Acdtool (management code for ACE3P), Omega3P (code in the ACE3P suite), and Impact-T compiled for the given platform. These codes need not be in the workflow directory but should have their absolute paths listed in the workflow configuration file. A desired A3PI workflow can be called as a python script locally, or inside a batch file in HPC settings.

Additionally, for multi-objective optimization, libEnsemble and DEAP must be installed in the user's python path. The A3PI code uses libEnsemble to manage parallel workers on HPC systems and are executed for varying values of the workflow's variables defined in the workflow configuration file. Multi-objective optimization is then performed with the genetic algorithms in the DEAP library using the parameters given in the workflow configuration file. All multi-objective workflow evaluations will be run in a HPC job, set up with a batch file. Checkpointing and restarting existing optimizations, should a run fail, has not yet been implemented into A3PI, but is available in libEnsemble within a given optimization.

For the specific setup on Cori@NERSC, libEnsemble and DEAP can be installed on with the commands: `pip install libensemble --user` and `pip install deap --user`.

For additional information or other installation options, see the respective user guides for libEnsemble and DEAP.

1.3 Running A3PI

To run an A3PI workflow, a configuration file must be setup as defined in the previous topics. The `[PATHS]`, `[RUN_PARAMETERS]`, and `[WORKFLOW]` sections must be provided but all other sections are optional depending on the usage of A3PI.

2 Input Files

As defined in the previous section, the workflow configuration file is the master A3PI input file which controls all parameters pertaining to Cubit, Acdtool, Omega3P, Impact-T, libEnsemble, and DEAP, as well as more settings. This file is *unique* for each use of A3PI and a copy will be placed in each workflow worker instance with the appropriate updated variables in the `[VARS]` section. Thus a user need only supply this workflow configuration file to use A3PI. However, for more involved tasks, several more files must be provided.

2.1 Workflow Configuration File

This text file must be used for any A3PI workflow runs and is formatted to be parsed by the ConfigParser python module with the extended interpolation option. The sections must be labeled with square brackets `[]` and keyword value pairs separated by equals signs `=`. Sections that are not used in a given workflow can be omitted; for example, if only Impact-T is to be run, then the Cubit, Omega3P, etc. sections can be ignored.

2.2 Optional Data Files

2.2.1 Cubit Journal Files

If Cubit is to be used in a given A3PI workflow, a corresponding journal file in `.jou` format must be provided. Additionally, if the journal file uses ACIS macros, then the corresponding `.sat` file must be provided as well. The variable names in Cubit journal must match those used in the keys of the `[CUBIT_ELEMENT_]` section of the workflow configuration file. This is so that A3PI can automatically adjust a journal file with a desired parameter update when evaluating the workflow. More details can be found in the Cubit user manual.

2.2.2 Omega3P Mesh Files

The mesh files used by Omega3P must be in the `.ncdf` format, not the `.geo` file directly generated from Cubit. The Acdtool code has a "meshconvert" command to convert a mesh file from `.geo` to `.ncdf` format and must be included in the `[WORKFLOW]` tasks before running an Omega3P task. More details can be found in the Omega3P user manual. The Omega3P configuration file is generated automatically using the settings in the A3PI workflow configuration file.

2.2.3 Omega3P Field Files

The output eigenmode field data from Omega3P will be located in a folder named by default `omega3p_results`, within a given workflow worker folder. This folder contains volumetric field data stored in an unstructured tetrahedral finite-element mesh. The Acdtool code has a "postprocess rf" command which is used to evaluate and interpolate this field data onto a regular Cartesian grid to be used by Impact-T or for other purposes.

The output from Acdtool will be 4 field files in `.hdf5` format using the OpenPMD standard: `E_Real.h5`, `E_Imag.h5`, `B_Rreal.h5`, and `B_Imag.h5`. These field files will need to be renamed accordingly for use in Impact-T with a file ID and the next consecutive 3 integers. For example, if the desired file ID for Impact-T is 300, then the 4 files must be renamed `data300.h5`, `data301.h5`, `data302.h5`, and `data303.h5` respectively. Because possible conflicts by using this file ID naming convention, file IDs used in Impact-T must be separated by at least 4.

2.2.4 Impact Data Files

The input file used for Impact-T can be generated automatically using the settings in the A3PI workflow configuration file. The field data for Impact-T, typically called by lattice elements with codes 105 or 111, is either a text file such as `rfdata300` in the case of on-axis 1D field data, or a set of `.h5` files as described in the previous section for 3D field data. In the case of 3D field data, Impact-T must use a defined `.T7` file as well but this is generated automatically by A3PI. Also user-generated input particle distribution can be provided. See the Impact-T manual for more information on the necessary field file structures.

In the case of Impact-Z, an A3PI parser is not yet implemented and thus a separate user-generated input file must be provided and defined in the `[RUN_PARAMETERS]` section with the key `impactz_input_file`. Similarly, Impact-T can use an external user-generated input file by defining it in the `[RUN_PARAMETERS]` section with the key `impactt_input_file` if desired.

3 Output Files

3.1 Single-Run Workflow Output

If the A3PI workflow is to be run once, then the output will be files left in the workflow worker folder. For example, if the `[WORKFLOW]` task list includes `cubit`, `meshconvert`, `omega3p`, `rfpost`, and `impactt`, then the workflow worker folder will contain the outputs from each of those steps. The workflow worker folder will not be deleted or overwritten unless A3PI is called again with a workflow configuration file using the same workflow folder name.

3.2 Multi-Run Workflow Output using LibEnsemble/DEAP

To use A3PI for multi-objective optimization, desired objectives and parameters must be set in the workflow configuration file. The output will be a history array object which can be read by libEnsemble. This history array object contains the history of all workflow worker

fitness evaluations used by the DEAP library. See the libEnsemble user guide for more information about the history array object and other features.

4 Post-processing and Visualization

4.1 Matlab Scripts

A set of simple Matlab scripts are included to aid visualizing meshes and Omega3P fields in `.ncdf` format and output files from Impact-T. Additionally, libEnsemble history array objects can be converted to text files using the `A3PI_ParseHistory.py` script for use in the Matlab scripts.

4.1.1 A3PI_PlotField.m

This Matlab script will use a mesh file in `.ncdf` format (output from Acdtool meshconvert) and an Omega3P mode file, also in `.ncdf` format, which is contained in the `omega3p_results` folder and has the `.mod` suffix.

Next, since Omega3P can use symmetry planes to reduce the number of FEM elements when solving for eigenmodes, the mesh and mode files provided can be half or quarter of the desired geometry and extended via reflections across the symmetry planes. Therefore, the `type` variable must be manually set to: `'full'` (no symmetry planes), `'halfx'` (y-symmetric), `'halfy'` (x-symmetric), or `'quarter'` (both x and y-symmetric).

Furthermore, in the field solver for these symmetric planes, a boundary condition is used for the reflections. These boundary conditions can be either `'electric'` or `'magnetic'` and will reflect the modal field accordingly across the symmetry planes.

Lastly, a plotting color function `'fcolor'` can be provided to color the vertices of the mesh provided according to the modal fields. This function takes in the two arguments `E` and `B` which are $(3 \times N)$ arrays containing the vertex field data for N vertices, with the rows corresponding to the x, y, or z component respectively, and outputs a scalar value for plot coloring. For example, to plot the magnitude of the $\mathbf{E}$ field, the line: `fcolor = @(E,B) (sum(E.^2,1)).^0.5;` will generate the appropriate coloring.

The plotting routine is contained in the last section of the script and can be user-modified as desired. Since the field plotting uses the `trisurf` built-in Matlab command, any properties to this type of plot can be adjusted as needed. More information is accessible by typing: `doc trisurf` in the command prompt.

4.1.2 A3PI_PlotMesh.m

This Matlab script is similar to A3PI_PlotField.m in that it uses the same approach to load the `.ncdf` file for the mesh but will not color the vertices. Also, with the symmetry options as described in the A3PI_PlotField.m script, the `type` variable must be manually set to: `'full'` (no symmetry planes), `'halfx'` (y-symmetric), `'halfy'` (x-symmetric), or `'quarter'` (both x and y-symmetric).

The plotting routine is contained in the last section of the script and can be user-modified as desired. Since the mesh plotting uses the `trimesh` built-in Matlab command, any proper-

ties to this type of plot can be adjusted as needed. More information is accessible by typing: `doc trimesh` in the command prompt.

4.1.3 A3PI_PlotParticles.m

This Matlab script will read-in either a single or multiple particle data files in Impact format. The `mode` setting can be one of three options: `'single'`, `'interactive'`, or `'animation'`. For all modes, the plotting style `ptype` for the particles can be set to `'circles'` which uses the Matlab command `scatter3`, or 3D `'bubbles'` which uses the command `bubbleplot3`. In the latter case, `bubbleplot3` is not a built-in Matlab method and instead is found on the Matlab file exchange[3]. Due to computational cost of rendering 3D spheres, it is recommended to use the plotting method `circles` when using the script in interactive mode.

When using the script in `'single'` mode, the user must specify the Impact particle *file* name (typically `fort.50`) with the variable `ifile`. Also, the user can specify a maximum number of particles for rendering; useful for plotting if the number of particles is very large.

When using the script in either `'interactive'` or `'animation'` mode, the user must specify an Impact particle *folder* name with the variable `ifolder`. This folder should contain particle Impact particle files with the naming structure `'fort.xxxx'` where the number xxxx corresponds to a single file output from an Impact simulation.

Note: currently, to generate a time-series sequence of particle files, one must manually edit the Impact-T source code to output a subset of particles to a file within the time-stepping loop. A future version of Impact may include beam element codes to output this set of files automatically with the Impact input file.

Alternatively, a user can use this Matlab script with any time-sliced particle data by producing a set of text files with the following format: each particle is defined by a row with 6 values: x, p_x, y, p_y, z, and p_z. Each time-slice will require its own file with increasing index and the naming convention `'fort.xxxx'` as defined before. See the Impact-T documentation for more information on this formatting structure.

Next, other plotting options for the script can be found in the user settings section of A3PI_PlotParticles.m. Of particular note, the variables `xplot` and `yplot` are used to select the quantities to plot along the x and y-axes respectively. For example, `xplot = 'x'` and `yplot = 'px'` will plot the particles' x-position vs its x-momentum. The third axis is always the particles' z-position (longitudinal coordinate). Lastly, a video animation output can be created by using the `v_flag` and `v_name` variables with the `'animation'` mode.

4.1.4 A3PI_PlotHistory.m

This Matlab script will read-in libEnsemble history data parsed from A3PI_ParseHistory.py and plot multi-objective optimization data. The `mode` setting can be either `'interactive'` or `'animation'`. Two input text files `data_inputs` and `data_outputs` must be provided and are generated from A3PI_ParseHistory.py. The `data_inputs` file contains rows corresponding to input space parameters while the `data_outputs` file contains the output parameters per individual and generation.

For example, consider a multi-objective optimization problem over an input space with 7 parameters and 3 output objectives. For a population of 100 individuals over 20 generations,

the `data_inputs` file will contain 2100 rows and 7 columns with each column corresponding to a particular input parameter and each row corresponding to an individual {1,2,3,...,100} and generation {0,1,2,...,20}. Similarly, the `data_outputs` fill will contain 3 columns corresponding to the 3 output objectives and rows for each individual and generation pair. The rows 1-100 represent generation 0, the rows 101-200 represent generation 1, and so forth.

The A3PI_PlotHistory.m script will plot individuals per generation with two output objectives. The `plot_ind` variable is an integer array of length two containing the objective numbers to plot (i.e. the column numbers in `data_outputs`). For example, for an optimization run with 3 objectives, to plot the first and third objectives, use `plot_ind = [1,3];`.

Optionally, plot bounds for each axis, `pb1` and `pb2`, can be assigned to restrict the plotting domain of the window if desired. Also, the cell arrays `xlist` and `ylist`, can be defined to name each of the input parameters and the output objectives respectively; this is used exclusively in `'interactive'` mode for the custom data cursor. Lastly, as in the case of A3PI_PlotParticles.m, a video animation output can be created by using the `v_flag` and `v_name` variables with the `'animation'` mode.

5 Known Issues and Limitations

5.1 HPC Environment Considerations

While libEnsemble was developed for use in several HPC systems, A3PI was developed solely for Cori@NERSC and thus using A3PI in a HPC environment other than Cori@NERSC may require additional debugging. Additionally, A3PI should run in single-run mode on any system with the necessary executables (i.e. Python, Cubit, Omega3P, Impact, etc.); in this mode, external tasks are called with the subprocess Python module instead of the libEnsemble manager.

5.2 LibEnsemble Worker Considerations

As of build 0.7.1 of libEnsemble, a limit of 64 workers can be assigned to concurrently evaluate A3PI workflows. Next, the number of nodes required for A3PI is given by the (number of workers)$\times$(number of nodes per worker)+1; the extra node is used by libEnsemble and its persistent generator to select a new generation from an existing population. The number of nodes per worker cannot change during an A3PI optimization, which is a limitation when toggling the use of Omega3P since it generally uses multiple nodes. Because of this issue, multi-node workers may not be optimally utilized when not running Omega3P.

When using the HPC system Cori (NERSC), there is also the limitation that each node can only run a single workflow if MPI is used (generally with Acdtool, Omega3P, and Impact-T). This limitation is set by the NERSC SLURM resource manager to only allow a single MPI process per node, even if the MPI process does not use all available cores. If a node is hanging, libEnsemble will attempt to retry the workflow task, but can fail at times. Furthermore, due to this issue, the phase scanning routine for Impact-T cannot be used in conjunction with libEnsemble; however this limitation can be bypassed by assigning the phases as input parameters in the optimization.

5.3 DEAP Multi-Objective Optimization Considerations

When using the DEAP multi-objective optimization, managed by libEnsemble, various settings must be defined in the `[LIBENSEMBLE]` parameters section of the workflow configuration file. In particular, the maximum number of generations and population size should be fixed to ensure a full optimization can complete during the allocated time with HPC resources. This fixed-population restriction is also required if using the A3PI_ParseHistory.py script to export the libEnsemble history array to text files.

6 List of A3PI Files

This section lists the files contained in A3PI and provides a short summary of each. In the input/output file subsection, the "(auto.)" tag denotes that this file may be automatically generated or manually provided depending on the workflow. Generally, the files for each code that are provided manually must be listed in the `[RUN_PARAMETERS]` section of the workflow input file with the `static_files` key. For example, all data files used in Impact simulations must be listed except for data files created from Acdtool.

A3PI File List		
Name	—	**Description**
`A3PI_Acdtool.py`	—	Runs Acdtool meshconvert or postprocess rf
`A3PI_AcdtoolTemplate.py`	—	Creates Acdtool postprocess rf input file
`A3PI_Cubit.py`	—	Runs Cubit with desired parameters
`A3PI_CubitTemplate.py`	—	Updates desired variables in Cubit journal file
`A3PI_Impact.py`	—	Runs Impact-T/Z with desired opts. (phase scan)
`A3PI_ImpactTemplate.py`	—	Creates Impact-T input file from configuration file
`A3PI_Interpolate.py`	—	Contains 1D and 3D field interpolation routines
`A3PI_LibEnsemble.py`	—	Contains libEnsemble setup routines
`A3PI_Omega3P.py`	—	Runs Omega3P with desired parameters
`A3PI_Omega3PTemplate.py`	—	Creates Omega3P input file
`A3PI_WorkflowTemplate.py`	—	Parses A3PI config. file and manages workflow

Input/Output Files		
Name	—	**Description**
`Cubit journal (.jou)`	—	Used to create a mesh from parametric geometry
`Cubit ACIS file (.sat)`	—	Used for Cubit ACIS macros
`Omega3P mesh file (.ncdf)`	—	(auto.) used to run Omega3P
`Omega3P input file (.in)`	—	(auto.) used to run Omega3P
`Acdtool rfpost file (.in)`	—	(auto.) used to run Acdtool postprocess rf
`Impact input file (.in)`	—	(auto.) used to run Impact
`Impact rfdata files`	—	(auto.) used for Impact 1D field lattice elems.
`Impact data files (.h5)`	—	(auto.) used for Impact 3D field lattice elems.
`Impact grid files (.T7)`	—	(auto.) used for Impact 3D field lattice elems.
`LibEnsemble hist. (.npy)`	—	(auto.) history array output from libEnsemble

Input/Output Files Continued		
Name	—	**Description**
`LibEnsemble log file`	—	(auto.) log text file of libEnsemble progress
`LibEnsemble stats`	—	(auto.) stats file of worker evaluations

Post-processing and Plotting Scripts		
Name	—	**Description**
`A3PI_ParseHistory.py`	—	Python script to write history array to text files
`A3PI_PlotHistory.m`	—	Matlab script to plot history data from text files
`A3PI_PlotMesh.m`	—	Matlab script to plot mesh from .ncdf files
`A3PI_PlotField.m`	—	Matlab script to plot 3D fields from .h5 files
`A3PI_PlotParticles.m`	—	Matlab script to plot particle dists. from Impact-T

User-generated File Examples		
Name	—	**Description**
`A3PI_config_optimize`	—	A3PI config file for libEnsemble and field interpolation
`A3PI_config_single`	—	A3PI config file for A3PI workflow (without libEnsemble)
`test_optimize.batch`	—	Batch script to run A3PI with libEnsemble and field interp.
`test_single.batch`	—	Batch script to run A3PI workflow once (w/o libEnsemble)

External Codes and Packages		
Name	—	**Description**
`Acdtool`	—	Processing tool from ACE3P suite
`Bubbleplot3`	—	Matlab 3D sphere scatterplot code (v 1.1.0.0)
`Cubit`	—	Mesh generating software (v 15.2)
`DEAP`	—	Differential evolution python library (v 1.3.1)
`ImpactT`	—	Beam dynamics code (v 2.0)
`LibEnsemble`	—	Workflow management python library (v 0.7.1)
`Omega3P`	—	Eigenmode solver code from ACE3P suite

7 Troubleshooting and Common Errors

The A3PI workflow manager does not have a robust set of error messages yet. This section is intended to help debug common issues which may arise when using A3PI. A general approach for most errors is to carefully check the A3PI workflow configuration file for typos or other mismatched settings if copy-pasting between different simulations. Due to A3PI's combined input file design, most issues can be resolved this way.

- **A3PI crashes on initializing:**
  This can occur if there are missing keyword-value pairs in required sections or missing files (e.g. listed in `static_files`). Every A3PI run must have a correctly formatted

configuration file with `[PATHS]`, `[RUN_PARAMETERS]`, and `[WORKFLOW]` sections. Please refer to the example configuration files provided.

- **A3PI crashes on initializing optimization mode:**
  This can be caused by not having libEnsemble or DEAP installed correctly or using an inconsistent number of nodes with the worker number provided. If a list of worker input files and folders are created in the base directory, then A3PI is initializing correctly but failing during a task chain evaluation step.

- **A3PI crashes when running Cubit:**
  This can be caused by missing a required journal file (`.jou`) or supplemental ACIS file `.sat`; check that these files are included in the `static_files` list in the `[RUN_PARAMETERS]` section and are referenced in the appropriate `[CUBIT_ELEMENT]` sections. Also, Cubit may fail if an input parameter is improperly referenced when using A3PI for multi-objective optimization; consider testing the output geometries manually for extreme values of each geometric parameter.

- **A3PI crashes when running Acdtool meshconvert:**
  This may occur if the geometry mesh generated from Cubit is badly formed or missing; consider examining the `.geo` file generated from Cubit.

- **A3PI crashes when running Omega3P:**
  This can be caused by a number of issues including: insufficient (or hanging) nodes allocated to workers, insufficient cores allocated for Omega3P, a missing or bad `.ncdf` mesh file, or incorrect settings listed in the `OMEGA3P` sections. Also, if MPI errors are generated immediately, check that the programming environment is set to GNU instead of Intel by using `module swap PrgEnv-intel PrgEnv-gnu` in the batch file preamble. Occasionally, Omega3P may fail randomly and restarting the task with fewer cores may fix the issue.

- **A3PI crashes when running Acdtool postprocess rf:**
  This may occur if Omega3P failed and Acdtool attempts to postprocess the field data with a missing mode file. Also, check that the number of cores for Acdtool is consistent with the resources allocated; typically a KNL node should use at most 64-68 cores. Furthermore, check the parameters in the `[RFPOST_MODE]` sections for other issues.

- **A3PI crashes when using field interpolation:**
  This can occur if the data files are not properly set up in appropriate folders with the corresponding folder name structure. Also, check that the interpolant evaluation is inside the convex hull of all pre-computed values. For example, if data files are provided for a range of a parameter from 0 to 10, ensure that the function is called only for values within that domain; in particular the bounds for that parameter in the DEAP optimizer should be set to that range.

- **A3PI crashes when using Impact-T:**
  This can occur if the MPI programming environment is not compatible with the compiled version of Impact-T; it is recommended to use the GNU compiler for all A3PI

external codes. Next, check that the Impact-T input file is generated correctly and all necessary input data files are listed in the `[RUN_PARAMETERS]` section under the `static_files` key.

- **A3PI crashes when using Impact-T during phase scanning:**
  This can occur if libEnsemble and DEAP are used with the phase scanning routine, which uses the multiprocessing Python package. The phase scanning routine can only be used in single-run evaluations of A3PI workflows. To skip using the phase scanning routine, do not use the arguments: `1`, `1a`, `2`, or `2a` when assigning Impact-T in the workflow task chain. Also, there is a rare case when Cori@NERSC may yield MPI errors when using the phase scanning routine in a batch script; a current workaround is to use Cori nodes in interactive mode with `salloc` instead of `sbatch`.

- **A3PI crashes when using Impact-Z:**
  The Impact-Z implementation relies on an external input file to define the beam parameters and lattice structure; ensure that an appropriate input file is defined in the `[RUN_PARAMETERS]` section under the `impactz_input_file` key. If Impact-Z is assigned in the task chain with the argument `1`, a particle data file will be created from a prior Impact-T run, from the `fort.50` file. For this case, check that this `fort.50` file exists and is in the appropriate folder.

References

[1] Stephen Hudson et al, *libEnsemble User Manual*, Argonne National Laboratory, Rev 0.7.1, 2020.

[2] Félix-Antoine Fortin et al, *DEAP: Evolutionary Algorithms Made Easy*, J Mach Learn Res, vol 13, pp 2171–2175, July 2012.

[3] Peter (PB) Bodin, BUBBLEPLOT3, MATLAB Central File Exchange, source available at: https://www.mathworks.com/matlabcentral/fileexchange/8231-bubbleplot3 retrieved May 30, 2021.