abqcy

Release 0.1.0.post6+g794c988

WANG Hailin

Jun 17, 2023

User Manual

Write Abaqus Subroutines in Cython.

- GitHub repository: https://github.com/haiiliin/abqcy
- PyPI: https://pypi.org/project/abqcy
- Documentation: https://abqcy.readthedocs.io
- Read the Docs: https://readthedocs.org/projects/abqcy
- Bug report: https://github.com/haiiliin/abqcy/issues

1 Table of Contents

1.1 Getting Started

`abqcy` allows you to write your Abaqus subroutines in Cython. It provides a command line tool to compile your Cython code into an object file (`.obj`) that can be used by Abaqus.

Installation

You can install abqcy with pip:

```
pip install abqcy
```

or install it from source:

```
pip install git+https://github.com/haiiliin/abqcy
```

Environment Setup

abqcy requires a working Abaqus installation with user subroutines enabled. Make sure the abaqus command is available in the command line, otherwise you need to create a new system environment variable ABAQUS_BAT_PATH and set it to the path of the abaqus.bat file.

abqcy uses Cython to compile your Cython code into a C source file (.c). In order to compile the C source file into an object file (.obj) that can be used by Abaqus, the abaqus make command is used (it uses the MSVC cl compiler). Since the compiled .c file requires the Python headers and libraries, abqcy will try to find them automatically and update the INCLUDE and LIB environment variables. If it fails to find them, you need to update the INCLUDE and LIB environment variables manually.

Usage

Compile the Subroutine

You can now write your Abaqus subroutine in Cython, simple scripts can be found in *Examples*.

Note: In order to not mess up with the Cython declarations, you can add a companion .pxd file with the same name as your Cython .py or .pyx file, and put the Cython declarations in it. If you are not comfortable with keeping two files, you can just use the .pyx file with the Cython declarations.

See *Examples* for detailed examples.

After you have written your subroutine, you can compile it with the abqcy command:

```
abqcy compile <path-to-your-subroutine>
```

This will compile your subroutine into a C source file (.c) and a C header file (.h), and then they will be compiled into an object file (.obj) that can be used by Abaqus. These files are in the same directory as your subroutine.

Now you can use the subroutine in Abaqus, like:

```
abaqus job=Job-1 input=<model.inp> user=<subroutine>
```

Run an Abaqus Job, Post-process and Visualize the Results in a Single Command

You can use the `abqcy run` command to run an Abaqus job with your subroutine, post-process the results and visualize them in a single command:

```
abqcy run <script-or-inp> --user=<subroutine> --job=<job-name> --output=<output-dir> --
↪post=<post-process-script> --visualization=<visualization-script>
```

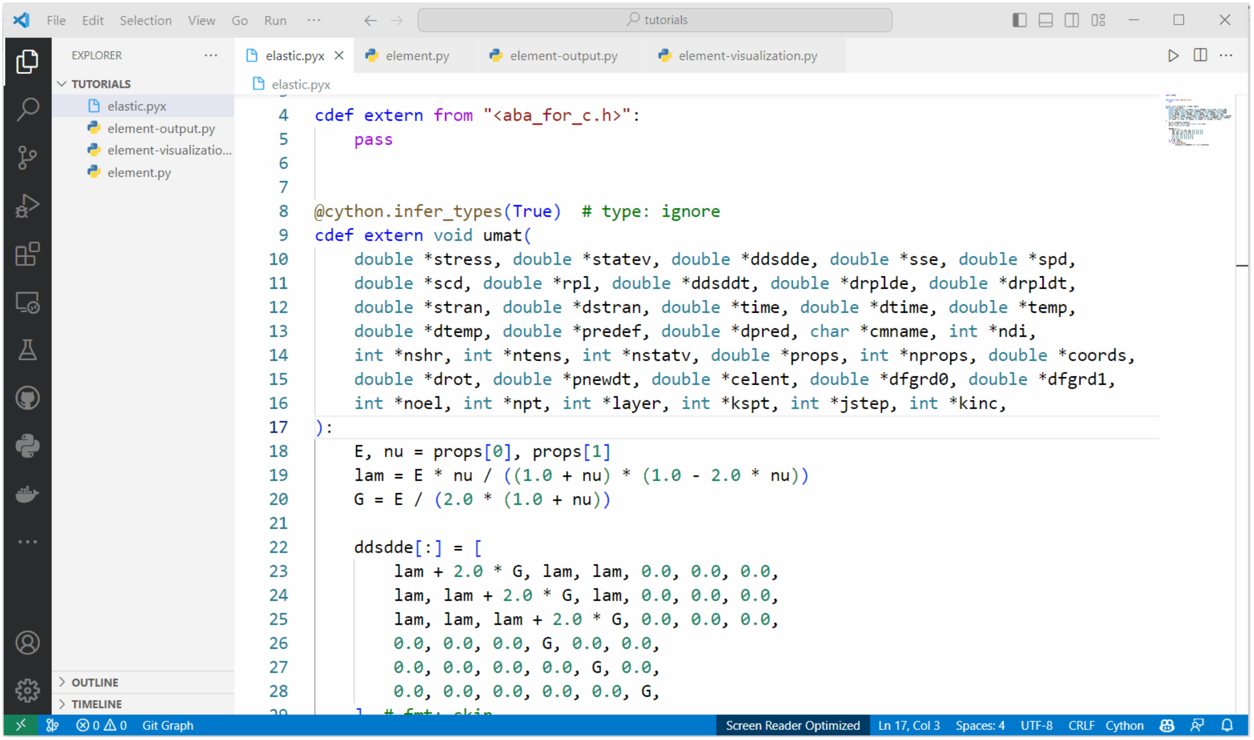
where:

- `script-or-inp`: a Python script (`.py`) file using the `abaqus cae` command to create the input file (`.inp`) or an input file (`.inp`) to run.
- `subroutine`: a Cython/Python file (`py` or `pyx`) or any other file that can be used by Abaqus as a user subroutine (`.f`, `.for`, `.c`, `.cc`, `.cpp`, `.cxx`). When using a Cython/Python file, the `abqcy compile` command will be used to compile it into an object file (`.obj`) before running the job.
- `job-name`: the name of the job to run. Defaults to the name of the input file.
- `output-dir`: the directory to store all the output files including models, subroutines, scripts, results, etc. Defaults to the current working directory.
- `post-process-script`: a Python script (`.py`) file to post-process the results using the `abaqus cae` command.
- `visualization-script`: a Python script (`.py`) file to visualize the results executed by the current Python interpreter.
- Additional flags are passed to the `abaqus` command.

See *abqcy.cli.AbqcyCLI.run()* method for more details.

1.2 Tutorial

This page is a tutorial for the `abqcy` project. It will guide you through the whole workflow of using `abqcy` to write an Abaqus user subroutine, create the model, extract the outout data, and visualize the results in a single command.



1. Write the user subroutine

The Abaqus user subroutine can be written in a Cython file. The subroutine is very similar to a Abaqus C/C++ subroutine, except that it is written in Cython syntax. Check *Examples* for some simple examples.

2. Create the model as an Abaqus input file

You can use the Python script to create an Abaqus model, check abqpy tutorials for a simple example. Noted that in the Python script, you should save the model into an Abaqus input file (`.inp`), so that the `abqcy` can read the model from the input file. For example:

```python
# Job
job = mdb.Job(name="element", model="Model-1")
job.writeInput()
```

If you are not familiar with Abaqus Python scripting, you can also use the Abaqus/CAE GUI to create an input file directly.

3. Write Python script to extract the output data from the Abaqus output database

You can use the Python script to extract the output data from the Abaqus output database (`.odb`). You can also find a simple example in the abqpy tutorials. Typically, this Python script will extract the output data from the Abaqus output database and save it into a data file.

4. Visualize the results

Data extracted from the Abaqus output database can be visualized using another Python script. For example, you can use the matplotlib library to plot the data.

5. Run the `abqcy` command

After all the above steps are completed, you can run the `abqcy run` to finish the whole workflow:

```
abqcy run --model=<script-or-inp> --user=<subroutine> --post=<script> --visualiation=<script>
```

In chronological order, the `abqcy run` command will:

- Generate an Abaqus input file from the Python script, if the `--model` option is a Python script.
- Compile the user subroutine to an object file (`.obj`), if the `--user` option is a Cython file.
- Run the Abaqus analysis with the `abaqus input=<inp> user=<obj>` command.
- Run the post-processing Python script with the `abaqus cae noGUI=<script>` command.
- Run the visualization Python script with the `python <script>` command, using the current Python interpreter (the Python interpreter where `abqcy` is installed) to visualize the results.

Example

The following is an example of scripts required by the `abqcy run` command:

```
import cython


cdef extern from "<aba_for_c.h>":
    pass


@cython.infer_types(True)  # type: ignore
cdef extern void umat(
    double *stress, double *statev, double *ddsdde, double *sse, double *spd,
    double *scd, double *rpl, double *ddsddt, double *drplde, double *drpldt,
    double *stran, double *dstran, double *time, double *dtime, double *temp,
    double *dtemp, double *predef, double *dpred, char *cmname, int *ndi,
    int *nshr, int *ntens, int *nstatv, double *props, int *nprops, double *coords,
    double *drot, double *pnewdt, double *celent, double *dfgrd0, double *dfgrd1,
    int *noel, int *npt, int *layer, int *kspt, int *jstep, int *kinc,
):
    E, nu = props[0], props[1]
    lam = E * nu / ((1.0 + nu) * (1.0 - 2.0 * nu))
    G = E / (2.0 * (1.0 + nu))

    ddsdde[:] = [
        lam + 2.0 * G, lam, lam, 0.0, 0.0, 0.0,
        lam, lam + 2.0 * G, lam, 0.0, 0.0, 0.0,
        lam, lam, lam + 2.0 * G, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, G, 0.0, 0.0,
```

(continues on next page)

(continued from previous page)

```
        0.0, 0.0, 0.0, 0.0, G, 0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, G,
    ]  # fmt: skip
    for i in range(6):
        for j in range(6):
            stress[i] += ddsdde[6 * i + j] * dstran[j]
```

```
from abaqus import *
from abaqusConstants import *
from caeModules import *
from driverUtils import *

executeOnCaeStartup()

# Model
model = mdb.models["Model-1"]

# Part
sketch = model.ConstrainedSketch(name="sketch", sheetSize=1.0)
sketch.rectangle((0, 0), (1, 1))
part = model.Part(name="part", dimensionality=THREE_D, type=DEFORMABLE_BODY)
part.BaseSolidExtrude(sketch=sketch, depth=1)

# Create sets
part.Set(name="set-all", cells=part.cells.findAt(coordinates=((0.5, 0.5, 0.5),)))
part.Set(name="set-bottom", faces=part.faces.findAt(coordinates=((0.5, 0.5, 0.0),)))
part.Set(name="set-top", faces=part.faces.findAt(coordinates=((0.5, 0.5, 1.0),)))
part.Surface(name="surface-top", side1Faces=part.faces.findAt(coordinates=((0.5, 0.5, 1.0),)))

# Assembly
model.rootAssembly.Instance(name="instance", part=part, dependent=ON)

# Material
material = model.Material(name="material")
material.UserMaterial(mechanicalConstants=(2.1e11, 0.3))
material.Depvar(n=2)

# Section
model.HomogeneousSolidSection(name="section", material="material", thickness=None)
part.SectionAssignment(region=part.sets["set-all"], sectionName="section")

# Step
step = model.StaticStep(
    name="Step-1",
    previous="Initial",
    description="",
    timePeriod=1.0,
    timeIncrementationMethod=AUTOMATIC,
    maxNumInc=100,
    initialInc=0.01,
    minInc=0.001,
```

(continues on next page)

(continued from previous page)

```python
    maxInc=0.1,
)

# Output request
field = model.FieldOutputRequest("F-Output-1", createStepName="Step-1", variables=("S", "E", "U"))

# Boundary condition
bottom_instance = model.rootAssembly.instances["instance"].sets["set-bottom"]
bc = model.DisplacementBC(
    name="BC-1", createStepName="Initial", region=bottom_instance, u1=SET, u2=SET, u3=SET, ur1=SET, ur2=SET, ur3=SET
)

# Load
top_instance = model.rootAssembly.instances["instance"].surfaces["surface-top"]
pressure = model.Pressure("pressure", createStepName="Step-1", region=top_instance, magnitude=1e9)

# Mesh
elem1 = mesh.ElemType(elemCode=C3D8I, elemLibrary=STANDARD, secondOrderAccuracy=OFF)
part.setElementType(regions=(part.cells,), elemTypes=(elem1,))
part.seedPart(size=0.1)
part.generateMesh()

# Job
job = mdb.Job(name="element", model="Model-1")
job.writeInput()
```

```python
import numpy as np
import visualization  # noqa
from abaqus import *
from abaqusConstants import *
from driverUtils import *

executeOnCaeStartup()

# Open output database
odb = session.openOdb("element.odb")

# Show the output database in viewport
session.viewports["Viewport: 1"].setValues(displayedObject=odb)

# Extract output data
dataList = session.xyDataListFromField(
    odb=odb, outputPosition=NODAL, variable=(("U", NODAL, ((COMPONENT, "U3"),)),), nodeSets=("INSTANCE.SET-TOP",)
)

data = np.array(dataList[0])
np.savetxt("U3.csv", data, header="time,U3", delimiter=",", comments="")
```

```python
import matplotlib.pyplot as plt
import pandas as pd

U3 = pd.read_csv("U3.csv")

fig, ax = plt.subplots()
ax.plot(U3["time"], U3["U3"])
ax.set_xlabel("Time (s)")
ax.set_ylabel("Displacement (m)")
ax.grid()
fig.savefig("U3.png", bbox_inches="tight", pad_inches=0.1)
```

```
abqcy run --model=element.py --user=elastic.pyx --post=element-output.py --
↪visualization=element-visualization.py
```

Note: You can check all the files in the docs/tutorials folder of the abqcy repository.

1.3 Examples

Below are some examples of how to use the library. To compile the examples into an object file (.obj) that can be used by Abaqus, you can run the following command:

```
abqcy compile <path-to-your-subroutine>
```

Note: It shoule be noted that temporary variables do not required to be typed in Cython except for integers. In the following examples, the cython.infer_types directive is used to infer types of untyped variables in function bodies including integers. This directive does a work similar to the auto keyword in C++ for the readers who are familiar with this language feature. It can be of great help to cut down on the need to type everything, but it also can lead to surprises.

See Determining where to add types for more information.

Example: Elastic umat subroutine

This example shows how to write an Abaqus elastic umat subroutine in Cython.

```python
import cython


cdef extern from "<aba_for_c.h>":
    pass


@cython.infer_types(True)  # type: ignore
cdef extern void umat(
    double *stress, double *statev, double *ddsdde, double *sse, double *spd,
    double *scd, double *rpl, double *ddsddt, double *drplde, double *drpldt,
    double *stran, double *dstran, double *time, double *dtime, double *temp,
```

(continues on next page)

(continued from previous page)

```
    double *dtemp, double *predef, double *dpred, char *cmname, int *ndi,
    int *nshr, int *ntens, int *nstatv, double *props, int *nprops, double *coords,
    double *drot, double *pnewdt, double *celent, double *dfgrd0, double *dfgrd1,
    int *noel, int *npt, int *layer, int *kspt, int *jstep, int *kinc,
):
    E, nu = props[0], props[1]
    lam = E * nu / ((1.0 + nu) * (1.0 - 2.0 * nu))
    G = E / (2.0 * (1.0 + nu))

    ddsdde[:] = [
        lam + 2.0 * G, lam, lam, 0.0, 0.0, 0.0,
        lam, lam + 2.0 * G, lam, 0.0, 0.0, 0.0,
        lam, lam, lam + 2.0 * G, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, G, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, G, 0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, G,
    ]  # fmt: skip
    for i in range(6):
        for j in range(6):
            stress[i] += ddsdde[6 * i + j] * dstran[j]
```

```
import cython


@cython.infer_types(True)  # type: ignore
def umat(
    stress, statev, ddsdde, sse, spd, scd, rpl, ddsddt, drplde, drpldt, stran, dstran,
    time, dtime, temp, dtemp, predef, dpred, cmname, ndi, nshr, ntens, nstatv, props,
    nprops, coords, drot, pnewdt, celent, dfgrd0, dfgrd1, noel, npt, layer, kspt,
    jstep, kinc,
):  # fmt: skip
    E, nu = props[0], props[1]
    lam = E * nu / ((1.0 + nu) * (1.0 - 2.0 * nu))
    G = E / (2.0 * (1.0 + nu))

    ddsdde[:] = [
        lam + 2.0 * G, lam, lam, 0.0, 0.0, 0.0,
        lam, lam + 2.0 * G, lam, 0.0, 0.0, 0.0,
        lam, lam, lam + 2.0 * G, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, G, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, G, 0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, G,
    ]  # fmt: skip
    for i in range(6):
        for j in range(6):
            stress[i] += ddsdde[6 * i + j] * dstran[j]
```

Note: You will need to add the Cython header file (`.pxd`) along with the Python file (`.py`) in order to use the Cython declarations.

```
cdef extern from "<aba_for_c.h>":
    pass


cdef extern void umat(
    double *stress, double *statev, double *ddsdde, double *sse, double *spd,
    double *scd, double *rpl, double *ddsddt, double *drplde, double *drpldt,
    double *stran, double *dstran, double *time, double *dtime, double *temp,
    double *dtemp, double *predef, double *dpred, char *cmname, int *ndi,
    int *nshr, int *ntens, int *nstatv, double *props, int *nprops, double *coords,
    double *drot, double *pnewdt, double *celent, double *dfgrd0, double *dfgrd1,
    int *noel, int *npt, int *layer, int *kspt, int *jstep, int *kinc,
)
```

Note: This file is required to use the Cython declarations in the Python file (`.py`).

1.4 Command Line Interface

The `abqcy` command line is used to compile you Cython code into an object (`.obj`) file that can be used by Abaqus. You can use it in the command line or in a Python script with the *`abqcy.cli.abqcy`* object (an *`abqcy.cli.AbqcyCLI`* object).

References

The `abqcy` command (`abqcy.cli.AbqcyCLI`)

```
$ abqcy
NAME
    abqcy - The ``abqcy`` command-line interface.

SYNOPSIS
    abqcy COMMAND

DESCRIPTION
    The ``abqcy`` command-line interface.

COMMANDS
    COMMAND is one of the following:

     compile
       Compile a Cython script to an Abaqus user subroutine as an object file.

     run
       Run Abaqus jobs.
```

The abqcy compile command (abqcy.cli.AbqcyCLI.compile())

```
$ abqcy compile --help
INFO: Showing help with the command 'abqcy compile -- --help'.

NAME
    abqcy compile - Compile a Cython script to an Abaqus user subroutine as an object␣
↪file.

SYNOPSIS
    abqcy compile SCRIPT <flags>

DESCRIPTION
    Compile a Cython script to an Abaqus user subroutine as an object file.

POSITIONAL ARGUMENTS
    SCRIPT
        Type: 'str'
        The path to the Cython script to compile.

FLAGS
    --exclude=EXCLUDE
        Type: Optional['list']
        Default: None
        When passing glob patterns as ``script``, you can exclude certain module names␣
↪explicitly by passing them into the ``exclude`` option.
    -n, --nthreads=NTHREADS
        Type: 'int'
        Default: 0
        The number of concurrent builds for parallel compilation (requires the␣
↪``multiprocessing`` module).
    --aliases=ALIASES
        Type: Optional['dict']
        Default: None
        If you want to use compiler directives like ``# distutils: ...`` but can only␣
↪know at compile time (when running the ``setup.py``) which values to use, you can use␣
↪aliases and pass a dictionary mapping those aliases
    -q, --quiet=QUIET
        Type: 'bool'
        Default: False
        If True, Cython won't print error, warning, or status messages during the␣
↪compilation.
    -f, --force=FORCE
        Type: 'bool'
        Default: False
        Forces the recompilation of the Cython modules, even if the timestamps don't␣
↪indicate that a recompilation is necessary.
    -l, --language=LANGUAGE
        Type: Optional['str']
        Default: None
        To globally enable C++ mode, you can pass ``language='c++'``. Otherwise, this␣
↪will be determined at a per-file level based on compiler directives.  This affects␣
↪only modules found based on file names.  Extension instances passed
```

(continues on next page)

(continued from previous page)

```
    --exclude_failures=EXCLUDE_FAILURES
        Type: 'bool'
        Default: False
        For a broad 'try to compile' mode that ignores compilation failures and simply␣
↪excludes the failed extensions, pass ``exclude_failures=True``. Note that this only␣
↪really makes sense for compiling ``.py`` files which can also be used without␣
↪compilation.
    --annotate=ANNOTATE
        Type: 'bool'
        Default: True
        Whether to generate an HTML file with annotations, by default True.
    Additional flags are accepted.
        Additional keyword arguments to pass to the ``cythonize`` function.

NOTES
    You can also use flags syntax for POSITIONAL ARGUMENTS
```

The `abqcy run` command (`abqcy.cli.AbqcyCLI.run()`)

```
$ abqcy run --help
INFO: Showing help with the command 'abqcy run -- --help'.

NAME
    abqcy run - Run Abaqus jobs.

SYNOPSIS
    abqcy run MODEL <flags>

DESCRIPTION
    Run Abaqus jobs.

POSITIONAL ARGUMENTS
    MODEL
        Type: 'str'
        The path to the input file or a Python script to create the input file.

FLAGS
    -u, --user=USER
        Type: Optional['str']
        Default: None
        The name of the user subroutine, if it is a Cython/Pure Python script, it will␣
↪be compiled to an object file automatically. If a companion ``.pxd`` file is found, it␣
↪will be copied to the output directory along with the Cython/Pure Python script.
    -j, --job=JOB
        Type: Optional['str']
        Default: None
        The name of the job, by default the model name without the extension.
    -o, --output=OUTPUT
        Type: Optional['str']
        Default: None
```

(continues on next page)

(continued from previous page)

```
        The path to the output directory, by default the current directory.
    -p, --post=POST
        Type: Optional['str']
        Default: None
        The Python script to run after finishing the job to post-process the results. In␣
↪the output script, a placeholder ``{odb}`` will be replaced with the path to the␣
↪output database file.
    -v, --visualization=VISUALIZATION
        Type: Optional['str']
        Default: None
        The Python script to run after finishing the job to visualize the results.␣
↪Typically, this script will plot a figure based on the data saved by the post-
↪processing script.
    Additional flags are accepted.
        Additional keyword arguments to pass to the ``abaqus`` command to run the job.

NOTES
    You can also use flags syntax for POSITIONAL ARGUMENTS
```

1.5 API Reference

This page contains auto-generated API reference documentation[1].

abqcy

Submodules

abqcy.cli

Module Contents

Classes

AbqcyCLI	The abqcy command-line interface.

Attributes

abqcy

class AbqcyCLI

The abqcy command-line interface.

[1] Created with sphinx-autoapi

_update_include_lib()

Update the `INCLUDE` and `LIB` environment variables.

compile(*script: str*, *, *exclude: list = None*, *nthreads: int = 0*, *aliases: dict = None*, *quiet: bool = False*, *force: bool = False*, *language: str = None*, *exclude_failures: bool = False*, *annotate: bool = True*, ***kwargs*)

Compile a Cython script to an Abaqus user subroutine as an object file.

Parameters

- **script** (`str`) – The path to the Cython script to compile.
- **exclude** (`list`, *optional*) – When passing glob patterns as `script`, you can exclude certain module names explicitly by passing them into the `exclude` option.
- **nthreads** (`int`, *optional*) – The number of concurrent builds for parallel compilation (requires the `multiprocessing` module).
- **aliases** (`dict`, *optional*) – If you want to use compiler directives like `# distutils: ...` but can only know at compile time (when running the `setup.py`) which values to use, you can use aliases and pass a dictionary mapping those aliases to Python strings when calling `cythonize()`. As an example, say you want to use the compiler directive `# distutils: include_dirs = ../static_libs/include/` but this path isn't always fixed and you want to find it when running the `setup.py`. You can then do `# distutils: include_dirs = MY_HEADERS`, find the value of `MY_HEADERS` in the `setup.py`, put it in a python variable called `foo` as a string, and then call `cythonize(..., aliases={'MY_HEADERS': foo})`.
- **quiet** (`bool`, *optional*) – If True, Cython won't print error, warning, or status messages during the compilation.
- **force** (`bool`, *optional*) – Forces the recompilation of the Cython modules, even if the timestamps don't indicate that a recompilation is necessary.
- **language** (`str`, *optional*) – To globally enable C++ mode, you can pass `language='c++'`. Otherwise, this will be determined at a per-file level based on compiler directives. This affects only modules found based on file names. Extension instances passed into `cythonize()` will not be changed. It is recommended to rather use the compiler directive `# distutils: language = c++` than this option.
- **exclude_failures** (`bool`, *optional*) – For a broad 'try to compile' mode that ignores compilation failures and simply excludes the failed extensions, pass `exclude_failures=True`. Note that this only really makes sense for compiling `.py` files which can also be used without compilation.
- **annotate** (`bool`, *optional*) – Whether to generate an HTML file with annotations, by default True.
- **kwargs** – Additional keyword arguments to pass to the `cythonize` function.

run(*model: str*, *, *user: str = None*, *job: str = None*, *output: str = None*, *post: str = None*, *visualization: str = None*, ***kwargs*)

Run Abaqus jobs.

Parameters

- **model** (`str`) – The path to the input file or a Python script to create the input file.
- **user** (`str`) – The name of the user subroutine, if it is a Cython/Pure Python script, it will be compiled to an object file automatically. If a companion `.pxd` file is found, it will be copied to the output directory along with the Cython/Pure Python script.

- **job** (str, *optional*) – The name of the job, by default the model name without the extension.
- **output** (str, *optional*) – The path to the output directory, by default the current directory.
- **post** (str, *optional*) – The Python script to run after finishing the job to post-process the results. In the output script, a placeholder `{odb}` will be replaced with the path to the output database file.
- **visualization** (str, *optional*) – The Python script to run after finishing the job to visualize the results. Typically, this script will plot a figure based on the data saved by the post-processing script.
- **kwargs** – Additional keyword arguments to pass to the `abaqus` command to run the job.

abqcy

abqcy.subs

Module Contents

STANDARD = ['CREEP', 'DFLOW', 'DFLUX', 'DISP', 'DLOAD', 'FILM', 'FLOW', 'FRIC', 'FRIC_COEF', 'GAPCON',...

EXPLICIT = ['VDFLUX', 'VDISP', 'VDLOAD', 'VEXTERNALDB', 'VFABRIC', 'VFRIC', 'VFRIC_COEF', 'VFRICTION',...

subs

abqcy.version

Module Contents

Functions

_get_version()	Return the version string used for __version__.

Attributes

_default_version
__version__

_default_version = '0.0.0'

_get_version()

Return the version string used for __version__.

__version__

2 Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

E

M

R

S