WrightTools Documentation

Release 0.0.0

WrightTools Developers

Apr 15, 2019

CONTENTS

`attune` is a package for representing, generating, and updating tuning curves.

PyPI	
version	
conda-forge	
version	
Read the Docs	
stable	
latest	

CHAPTER

ONE

CONTENTS

Note: This page contains information that is not (yet) true, as attune is not yet distributed. However, it should be true once a release is made, which is hopefully soon.

1.1 Installation

`attune` requires Python 3.6 or newer.

1.1.1 conda-forge

Conda is a multilingual package/environment manager. It seamlessly handles non-Python library dependencies which many scientific Python tools rely upon. Conda is recommended, especially for Windows users. If you don't have Python yet, start by installing Anaconda or miniconda.

conda-forge is a community-driven conda channel. conda-forge contains an attune feedstock.

```
conda config --add channels conda-forge
conda install attune
```

To upgrade:

```
conda update attune
```

1.1.2 pip

pip is Python's official package manager. attune is hosted on PyPI.

```
pip install attune
```

To upgrade:

```
pip install attune --upgrade
```

1.2 Contributing

Thank you so much for contributing to attune! We really appreciate your help.

If you have any questions at all, please either open an issue on GitHub or email a attune maintainer. The current maintainers can always be found in CONTRIBUTORS.

1.2.1 Preparing

1. fork the attune repository (if you have push access to the main repository you can skip this step)
2. clone attune to your machine:

```
$ git clone <your fork>
```

3. in the cloned directory (note, to install to system python, you may need to use sudo for this command):

```
$ pip install -e .[dev]
```

4. run tests

```
$ python setup.py test
```

1.2.2 Contributing

1. **ensure that the changes you intend to make have corresponding issues on GitHub**
   a) if you aren't sure how to break your ideas into atomic issues, feel free to open a discussion issue
   b) looking for low-hanging fruit? check out the help wanted label for beginner-friendly issues

```
$ # Create the branch, including remote
$ git branch <your branch> --set-upstream-to origin origin/<your branch>
$ git checkout <your branch> # Switch to the newly created branch
```

2. run all tests to ensure that nothing is broken right off the start

```
$ python setup.py test
```

3. make your changes, commiting often

```
$ git status # See which files you have changed/added
$ git diff # See changes since your last commit
$ git add <files you wish to commit>
$ git commit -m "Description of changes" -m "More detail if needed"
```

4. mark your issues as resolved (within your commit message):

```
$ git commit -m "added crazy colormap (resolves #99)"
```

   a. If your commit is related to an issue, but does not resolve it, use addresses #99 in the commit message

5. if appropriate, add tests that address your changes (if you just fixed a bug, it is strongly reccomended that you add a test so that the bug cannot come back unanounced)
6. once you are done with your changes, run your code through flake8 and pydocstyle

```
$ flake8 file.py
$ pydocstyle file.py
```

7. rerun tests
8. add yourself to CONTRIBUTORS
9. push your changes to the remote branch (github)

```
$ git pull # make sure your branch is up to date
$ git push
```

10. make a pull request to the master branch
11. communicate with the maintainers in your pull request, assuming any further work needs to be done
12. celebrate!

1.2.3 Style

Internally we use the following abbreviations:

WrightTools `import WrightTools as wt`

Matplotlib `import matplotlib as mpl`

Pyplot `from matplotlib import pyplot as plt`

NumPy `import numpy as np`

`attune` follows pep8, with the following modifications:

1. Maximum line length from 79 characters to 99 characters.

`attune` also folows numpy Docstring Convention, which is a set of adjustments to pep257. `attune` additionally ignores one guideline:

1. `attune` does not require all magic methods (e.g. `__add__`) to have a docstring.
    a) It remains encourged to add a docstring if there is any ambiguity of the meaning.

We use flake8 for automated code style enforcement, and pydocstyle for automated docstring style checking.

```
$ # These will check the whole directory (recursively)
$ flake8
$ pydocstyle
```

Consider using black for automated code corrections. Black is an opinionated code formatter for unambiguous standardization.

```
$ git commit -m "Describe changes"
$ black file.py
$ git diff # review changes
$ git add file.py
$ git commit -m "black style fixes"
```

We also provide a configuration to use git hooks to automatically apply `black` style to edited files. This hook can be installed using `pre-commit`:

```
$ pre-commit install
```

When committing, it will automatically apply the style, and prevent the commit from completing if changes are made. If that is the case, simply re-add the changed files and then commit again. This prevents noisy commit logs with changes that are purely style conformity.

1.3 attune API

1.3.1 attune.curve package

Module contents

OPA tuning curves.

Curve(setpoints, dependents, name[, ...])	Central object-type for all OPA tuning curves.
Dependent(positions, name[, units, ...])	Container class for dependent arrays.
Setpoints(positions, name[, units])	
TopasCurve(setpoints, dependents, name[, ...])	

attune.curve.Curve

class attune.curve.**Curve**(*setpoints*, *dependents*, *name*, *interaction=None*, *kind='curve'*, *method=<class 'attune.interpolator._linear.Linear'>*, *subcurve=None*, *source_setpoints=None*, *fmt=None*, ***kwargs*)

Central object-type for all OPA tuning curves.

__init__(*setpoints*, *dependents*, *name*, *interaction=None*, *kind='curve'*, *method=<class 'attune.interpolator._linear.Linear'>*, *subcurve=None*, *source_setpoints=None*, *fmt=None*, ***kwargs*)

Create a `Curve` object.

Parameters

- **setpoints** (*attune.Setpoints*) – The setpoint destinations for the curve.
- **dependents** (*list of Dependent objects*) – Dependent positions for each setpoint.
- **name** (*str*) – Name of curve.
- **kind** (*string*) – The kind of curve (for saving).
- **method** (*interpolation class*) – The interpolation method to use.

Methods

__init__(setpoints, dependents, name[, ...])	Create a `Curve` object.
coerce_dependents()	Coerce the dependent positions to lie exactly along the interpolation positions.
convert(units, *[, convert_dependents])	Convert the setpoints to new units.
copy()	Copy the curve object.
get_dependent_positions(setpoint[, units, full])	Get the dependent positions for a destination setpoint.

Continued on next page

Table 2 – continued from previous page

get_limits([units])	Get the edges of the curve.
get_source_setpoint(setpoint[, units])	Get setpoint of source curve.
interpolate([interpolate_subcurve])	Generate the interploator object.
map_setpoints(setpoints[, units])	Map the curve onto new tune points using the curve's own interpolation method.
offset_by(dependent, amount)	Offset a dependent by some ammount.
offset_to(dependent, destination, setpoint)	Offset a dependent such that it evaluates to *destination* at *setpoint*.
plot([autosave, save_path, title])	Plot the curve.
read(filepath[, subcurve])	
save([save_directory, plot, verbose, full])	Save the curve.
sort()	

Attributes

dependent_names	Get dependent names.
dependent_units	Get dependent names.

attune.curve.Dependent

class attune.curve.**Dependent**(*positions*, *name*, *units=None*, *differential=False*, *index=None*)

Container class for dependent arrays.

__init__(*positions*, *name*, *units=None*, *differential=False*, *index=None*)

Create a Dependent object.

Parameters

- **positions** (*1D array*) – Dependent positions.
- **name** (*string*) – Name.

Methods

__init__(positions, name[, units, ...])	Create a Dependent object.
convert(units)	

attune.curve.Setpoints

class attune.curve.**Setpoints**(*positions*, *name*, *units=None*)

__init__(*positions*, *name*, *units=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(positions, name[, units])	Initialize self.
convert(units)	

attune.curve.TopasCurve

class attune.curve.**TopasCurve**(*setpoints*, *dependents*, *name*, *interaction=None*, *kind='curve'*, *method=<class 'attune.interpolator._linear.Linear'>*, *subcurve=None*, *source_setpoints=None*, *fmt=None*, ***kwargs*)

__init__(*setpoints*, *dependents*, *name*, *interaction=None*, *kind='curve'*, *method=<class 'attune.interpolator._linear.Linear'>*, *subcurve=None*, *source_setpoints=None*, *fmt=None*, ***kwargs*)

Create a `Curve` object.

Parameters

- **setpoints** (*attune.Setpoints*) – The setpoint destinations for the curve.
- **dependents** (*list of Dependent objects*) – Dependent positions for each setpoint.
- **name** (*str*) – Name of curve.
- **kind** (*string*) – The kind of curve (for saving).
- **method** (*interpolation class*) – The interpolation method to use.

Methods

__init__(setpoints, dependents, name[, ...])	Create a `Curve` object.
coerce_dependents()	Coerce the dependent positions to lie exactly along the interpolation positions.
convert(units, *[, convert_dependents])	Convert the setpoints to new units.
copy()	Copy the curve object.
get_dependent_positions(setpoint[, units, full])	Get the dependent positions for a destination setpoint.
get_limits([units])	Get the edges of the curve.
get_source_setpoint(setpoint[, units])	Get setpoint of source curve.
interpolate([interpolate_subcurve])	Generate the interploator object.
map_setpoints(setpoints[, units])	Map the curve onto new tune points using the curve's own interpolation method.
offset_by(dependent, amount)	Offset a dependent by some ammount.
offset_to(dependent, destination, setpoint)	Offset a dependent such that it evaluates to *destination* at *setpoint*.
plot([autosave, save_path, title])	Plot the curve.
read(filepaths, interaction_string)	Create a curve object from a TOPAS crv file.
read_all(filepaths)	
save(save_directory[, full])	Save a curve object.
sort()	

Attributes

dependent_names	Get dependent names.
dependent_units	Get dependent names.

1.3.2 attune.interpolator package

Module contents

Interpolator(setpoints, dependent)	
Linear(setpoints, dependent)	Linear interpolation.
Poly(*args, **kwargs)	
Spline(setpoints, dependent)	

attune.interpolator.Interpolator

class attune.interpolator.**Interpolator**(*setpoints*, *dependent*)

__init__(*setpoints*, *dependent*)
Create an Interoplator object.

Parameters

- **setpoints** (*1D array*) – Setpoints.
- **units** (*string*) – Units.
- **dependents** (*list of WrightTools.tuning.curve.Dependent*) – Dependents.

Methods

__init__(setpoints, dependent)	Create an Interoplator object.

attune.interpolator.Linear

class attune.interpolator.**Linear**(*setpoints*, *dependent*)
Linear interpolation.

__init__(*setpoints*, *dependent*)
Create an Interoplator object.

Parameters

- **setpoints** (*1D array*) – Setpoints.
- **units** (*string*) – Units.
- **dependents** (*list of WrightTools.tuning.curve.Dependent*) – Dependents.

Methods

__init__(setpoints, dependent)	Create an Interoplator object.

Attributes

function

attune.interpolator.Poly

class attune.interpolator.**Poly**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)
Create an Interoplator object.

Parameters

- **setpoints** (*1D array*) – Setpoints.
- **units** (*string*) – Units.
- **dependents** (*list of WrightTools.tuning.curve.Dependent*) – Dependents.

Methods

__init__(*args, **kwargs)	Create an Interoplator object.

Attributes

function

attune.interpolator.Spline

class attune.interpolator.**Spline**(*setpoints*, *dependent*)

__init__(*setpoints*, *dependent*)
Create an Interoplator object.

Parameters

- **setpoints** (*1D array*) – Setpoints.
- **units** (*string*) – Units.
- **dependents** (*list of WrightTools.tuning.curve.Dependent*) – Dependents.

Methods

__init__(setpoints, dependent)	Create an Interoplator object.

Attributes

function

1.3.3 attune.workup package

Module contents

Methods for processing OPA 800 tuning data.

intensity(data, channel, dependent[, curve, ...])	Workup a generic intensity plot for a single dependent.
tune_test(data, channel[, curve, level, ...])	Workup a Tune Test.

attune.workup.intensity

attune.workup.**intensity**(*data*, *channel*, *dependent*, *curve=None*, ***, *level=False*, *cutoff_factor=0.1*, *autosave=True*, *save_directory=None*, ***spline_kwargs*)

Workup a generic intensity plot for a single dependent.

Parameters **data** (*wt.data.Data objeect*) – should be in (setpoint, dependent)

Returns New curve object.

Return type curve

attune.workup.tune_test

attune.workup.**tune_test**(*data*, *channel*, *curve=None*, ***, *level=False*, *cutoff_factor=0.01*, *autosave=True*, *save_directory=None*)

Workup a Tune Test.

Parameters

- **data** (*wt.data.Data object*) – should be in (setpoint, detuning)
- **curve** (*attune_curve object*) – tuning curve used to do tune_test
- **channel_nam** (*str*) – name of the signal chanel to evalute
- **level** (*bool (optional)*) – does nothing, default is False
- **cutoff_factor** (*float (optoinal)*) – minimum value for data-point/max(datapoints) for point to be included in the fitting procedure, default is 0.01
- **autosave** (*bool (optional)*) – saves output curve if True, default is True
- **save_directory** (*str*) – directory to save new curve, default is None which uses the data source directory

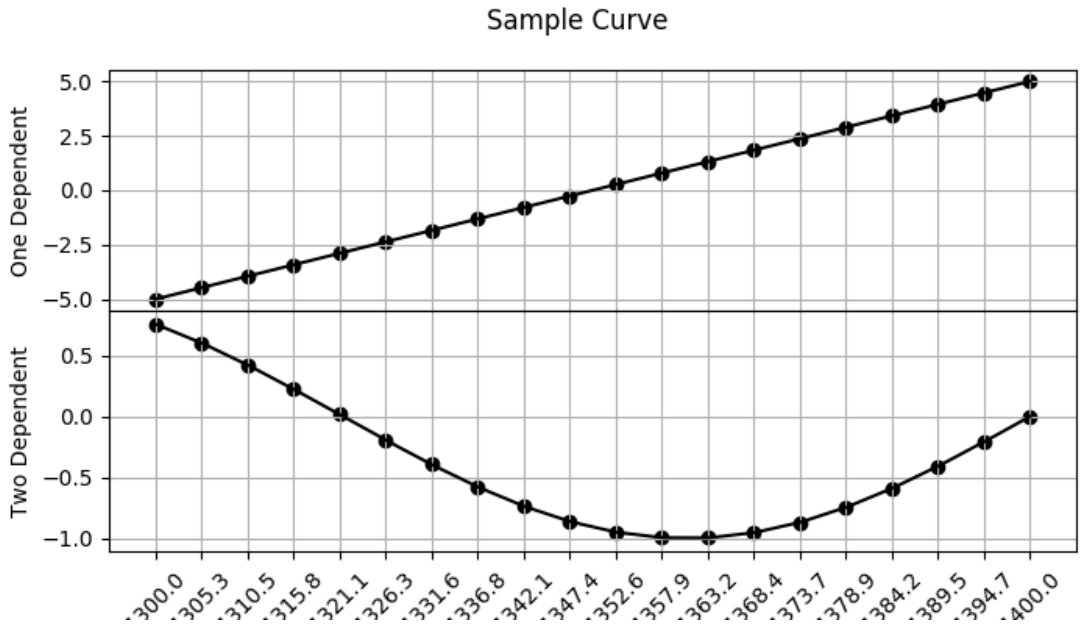
Returns New curve object.

Return type curve

1.4 Gallery

Note: Click *here* to download the full example code

1.4.1 Simple Curve Plot




```
import attune
import numpy as np

d0 = attune.Dependent(np.linspace(-5, 5, 20), "One Dependent")
d1 = attune.Dependent(np.sin(np.linspace(-4, 0, 20)), "Two Dependent")
s = attune.Setpoints(np.linspace(1300, 1400, 20), "Some Setpoints", "wn")
c = attune.Curve(s, [d0, d1], "Sample Curve")

c.plot()
```

Total running time of the script: (0 minutes 0.300 seconds)

CHAPTER

TWO

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

C

D

I

L

P

S

T