taurex Documentation

Release 3.1.4-alpha

Ahmed Al-Refaie

Jul 14, 2023

CONTENTS:

This guide covers general installation, using the standalone `taurex` program and the library

CHAPTER

ONE

INTRODUCTION

TauREx (Tau Retrieval for Exoplanets) is a fully bayesian inverse atmospheric retrieval framework. TauREx is a very extensive retrieval framework with a wide range of functionalities. TauREx3 is the next-generation of the atmospheric retrieval code. It acts as both a retrieval code and as a library that provides functionality relating to atmospheric modelling. The user is free to mix and match and use whatever needed.

The aim of TauREx3 is for anyone to come in an put in their own physics/models/chemistry and then perform a retrieval on it with the minimum of effort.

CHAPTER

TWO

INSTALLATION

TauREx3 only works with Python 3.5+. If you need to use Python 2.7 consider using TauREx2.

2.1 Installing from PyPi

Simply do:

```
pip install taurex
```

To test for correct setup you can do:

```
python -c "import taurex; print(taurex.__version__)"
```

Additionally, to restore the equilbrium chemistry and BHMie from TauREx 3.0 you can run:

```
pip install taurex_ace taurex_bhmie
```

2.2 Installing from git source directly (platform-independent)

You can directly get the most cutting-edge release from the repo:

```
pip install git+https://github.com/ucl-exoplanets/TauREx3_public.git
```

You can also clone TauREx3 from our main git repository:

```
git clone https://github.com/ucl-exoplanets/TauREx3_public.git
```

Move into the TauREx3 folder:

```
cd TauREx3
```

Then, just do:

```
pip install .
```

To test for correct setup you can do:

```
python -c "import taurex; print(taurex.__version__)"
```

If no errors appeared then it was successfuly installed. Additionally the `taurex` program should now be available in the command line:

```
taurex
```

To build documentation do:

```
python setup.py build_sphinx
```

The output can be found in the `doc/build` directory

2.2.1 Dependencies

As TauREx3 is pure python ,there are no prerequisites. Additionally these packages are also download and installed during setup:

- numpy
- numba
- numexpr
- configobj for parsing input files
- nestle for basic retrieval
- h5py for output

TauREx3 also includes 'extras' that can be enabled by installing additional dependancies:

- Lightcurve modelling requires pylightcurve
- Plotting using `taurex-plot` requires matplotlib
- Retrieval using Multinest requires pymultinest
- Retrieval using PolyChord requires pypolychord
  - The dynamic version requires dypolychord as well

CHAPTER

THREE

MPI

The message passing protocol (MPI) is not needed to install, run and perform retrievals.

This is more of a help guide to getting mpi4py working successfully and is not specific to TauREx3

There are some optimizers that can make use of MPI to significantly speed up retrievals. Specifically the Multinest and PolyChord optimizers. Considering most people have difficulty with installing it, this guide has been written to make the experience as smooth as possible.

First you must have an MPI library installed, this may already be installed in your system (such as a cluster) or you can install a library youself. For Mac users the quickest way to install it is through Homebrew:

```
brew install openmpi
```

Now we need to install our python MPI wrapper library mpi4py:

```
pip install mpi4py
```

You can test the installation:

```
mpirun -n 4 python -m mpi4py.bench helloworld
```

Replace *mpirun* with whatever the equivalent is for your system

You should get a similar output like so:

```
Hello, World! I am process 0 of 4 on blahblah.
Hello, World! I am process 1 of 4 on blahblah.
Hello, World! I am process 2 of 4 on blahblah.
Hello, World! I am process 3 of 4 on blahblah.
```

Then you are all set! Theres no need to reinstall TauREx3 as it will now import it successfully when run.

Tip: TauREx3 actually suppresses text output from other processes so running under MPI will actually look its being run serially. In fact if you get multiple of the same outputs this is a surefire way to know that something is wrong with the mpi4py installation!!!

However if you get something like this:

```
Hello, World! I am process 0 of 1 on blahblah.
Hello, World! I am process 0 of 1 on blahblah.
Hello, World! I am process 0 of 1 on blahblah.
Hello, World! I am process 0 of 1 on blahblah.
```

This means mpi4py has not correctly installed. This likely happens in cluster environments with multiple MPI libraries. You can overcome this by re-installing mpi4py with the `MPICC` enviroment set:

```
env MPICC=mpicc pip install --no-cache-dir mpi4py
```

Or:

```
env MPICC=/path/to/mpicc pip install --no-cache-dir mpi4py
```

Now re-run the test. If you get the correct result. Horray! If not, its best to ask your administrator.

Once you have this installed, you can install pymultinest here.

CHAPTER

FOUR

SUPPORTED DATA FORMATS

4.1 Cross-sections

Supported formats are:

- `.pickle` *TauREx2* pickle format
- `.hdf5`, `.h5` New HDF5 format
- `.dat`, ExoTransmit format

More formats can be included through *Plugins*

Tip: For opacities we recommend using hi-res cross-sections (R>7000) from a high temperature linelist. Our recommendation are linelists from the ExoMol project.

4.2 K-Tables

New in version 3.1.

Supported formats are:

- `.pickle` *TauREx2* pickle format
- `.hdf5`, `.h5` petitRADTRANS HDF5 format
- `.kta`, NEMESIS format

More formats can be included through *Plugins*

4.3 Observation

For observations, the following formats supported are:

- Text based 3/4-column data
- `.pickle` Outputs from Iraclis

More formats can be included through *Plugins*

4.4 Collisionally Induced Absorption

Only a few formats are supported

- `.db` *TauREx2* CIA pickle files
- `.cia` HITRAN cia files

CHAPTER

FIVE

THE TAUREX PROGRAM

This section of the documentation deals with using the main `taurex` program is accessed simply by running in the command line:

```
taurex
```

a help can be accessed by doing:

```
taurex --help
```

5.1 Quickstart

To get quickly up to speed lets try an example run using TauREx3. We will be using the `examples/parfiles/quickstart.par` file as a starting point and `examples/parfiles/quickstart.dat` as our observation. Copy these to a new folder somewhere.

5.1.1 Prerequisites

Before reading this you should have a few things on hand. Firstly `H2O` and `CO2` absorption cross sections in one of the *Supported Data Formats* is required. This example assumes cross-sections at *R=10000*. Secondly some collisionally induced absorption (CIA) cross sections are also required for a later part for `H2-He` and `H2-H2`, you can get these from the HITRAN site.

Tip: A starter set of these cross-sections and cia can be found in this dropbox: https://www.dropbox.com/sh/13y33d02vh56jh2/AABxuHdrZI83bSgoLz1Wzb2Fa?dl=0

5.1.2 Setup

In order to begin running forward models we need to tell *TauREx3* where our cross-sections are. We can do this by defining an `xsec_path` for cross sections and `cia_path` for CIA cross-sections under the `[Global]` header in our `quickstart.par` files like so:

```
[Global]
xsec_path = /path/to/xsec
cia_path = /path/to/cia
```

5.1.3 Forward Model

Using our input we can run and plot the forward model by doing:

```
taurex -i quickstart.par --plot
```

And we should get:

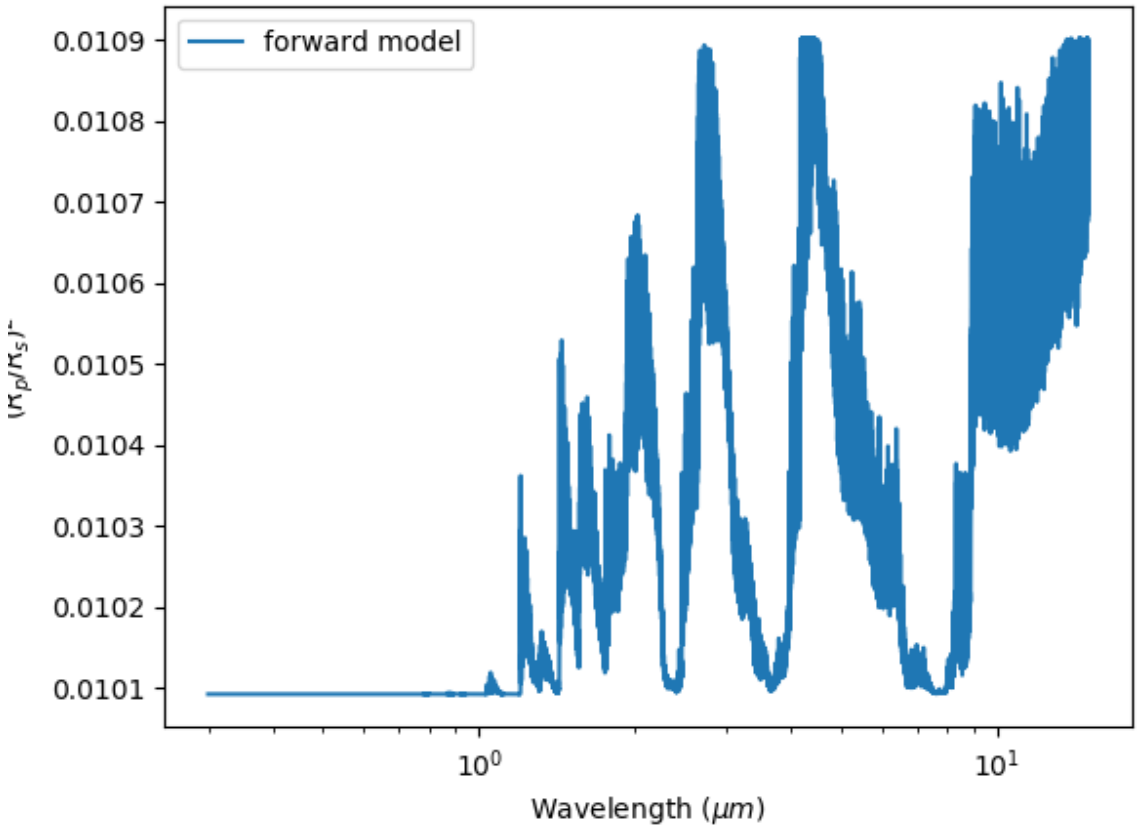


Fig. 1: Our first forward model

Lets try plotting it against our observation. Under the `[Observation]` header we can add in the `observed_spectrum` keyword and point it to our `quickstart.dat` file like so:

```
[Observation]
observed_spectrum = /path/to/quickstart.dat
```

Now the spectrum will be binned to our observation:

You may notice that general structure and peaks don't seem to match up with observation. Our model doesn't seem to do the job and it may be the fault of our choice of molecule. Lets move on to chemistry.

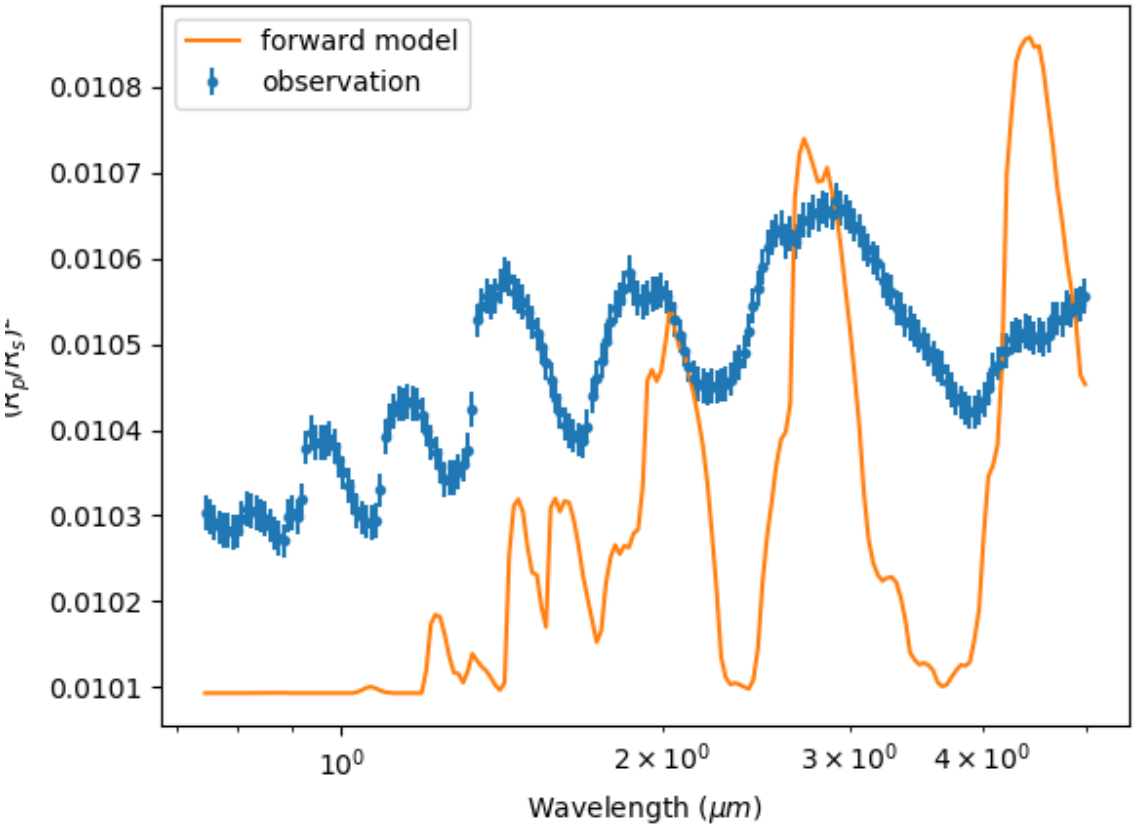


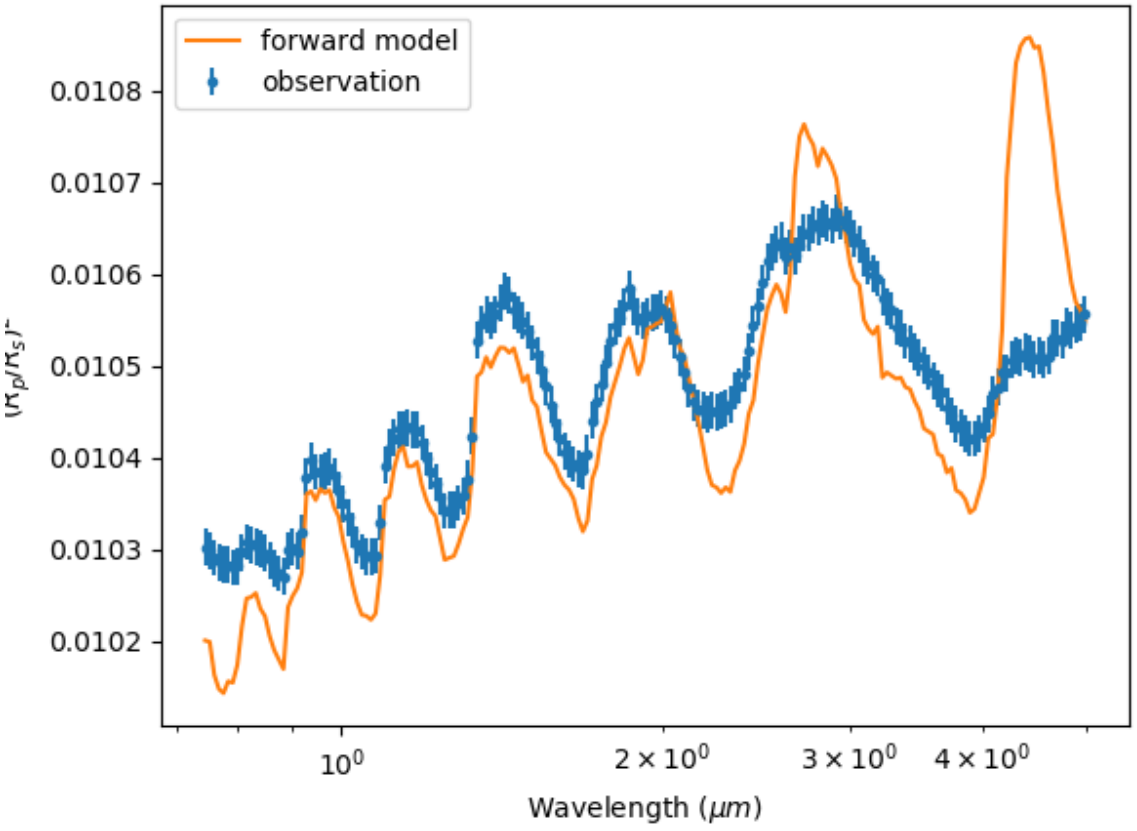
Fig. 2: Our binned observation

5.1.4 Chemistry

As we've seen, `CO2` doesn't fit the observation very well, we should try adding in another molecule. Underneath the `[Chemistry]` section we can add another sub-header with the name of our molecule, for this example we will use a `constant` gas profile which keeps it abundance constant throughout the atmosphere, there are other more complex profiles but for now we'll keep it simple:

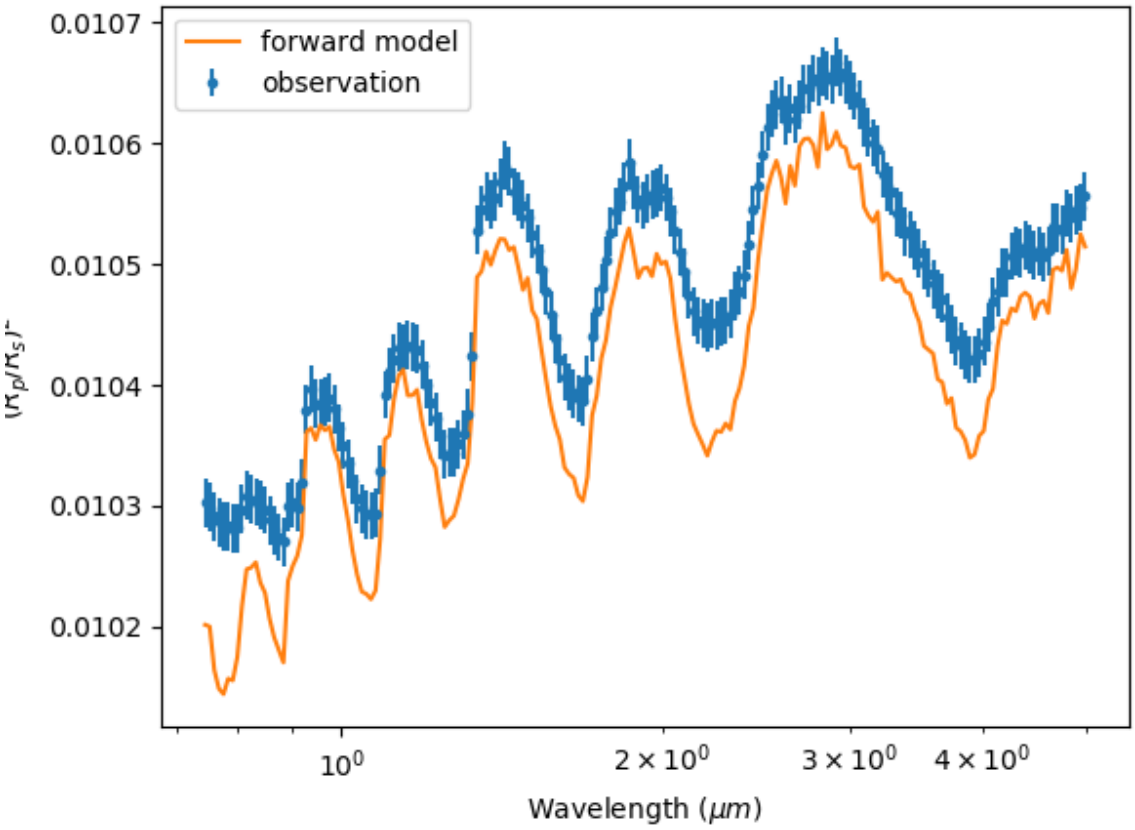
```
[Chemistry]
chemistry_type = taurex
fill_gases = H2,He
ratio=4.8962e-2

    [[H2O]]
    gas_type = constant
    mix_ratio=1.1185e-4

    [[CO2]]
    gas_type=constant
    mix_ratio=1.1185e-4

    [[N2]]
    gas_type = constant
    mix_ratio = 3.00739e-9
```

Plotting it gives:



We're getting there. It looks like H2O is definately there but maybe `CO2` isn't? Lets try it by commenting it out:

```
[Chemistry]
chemistry_type = taurex
fill_gases = H2,He
ratio=4.8962e-2

    [[H2O]]
    gas_type = constant
    mix_ratio=1.1185e-4

    #[[CO2]]
    #gas_type=constant
    #mix_ratio=1.1185e-4

    [[N2]]
    gas_type = constant
    mix_ratio = 3.00739e-9
```



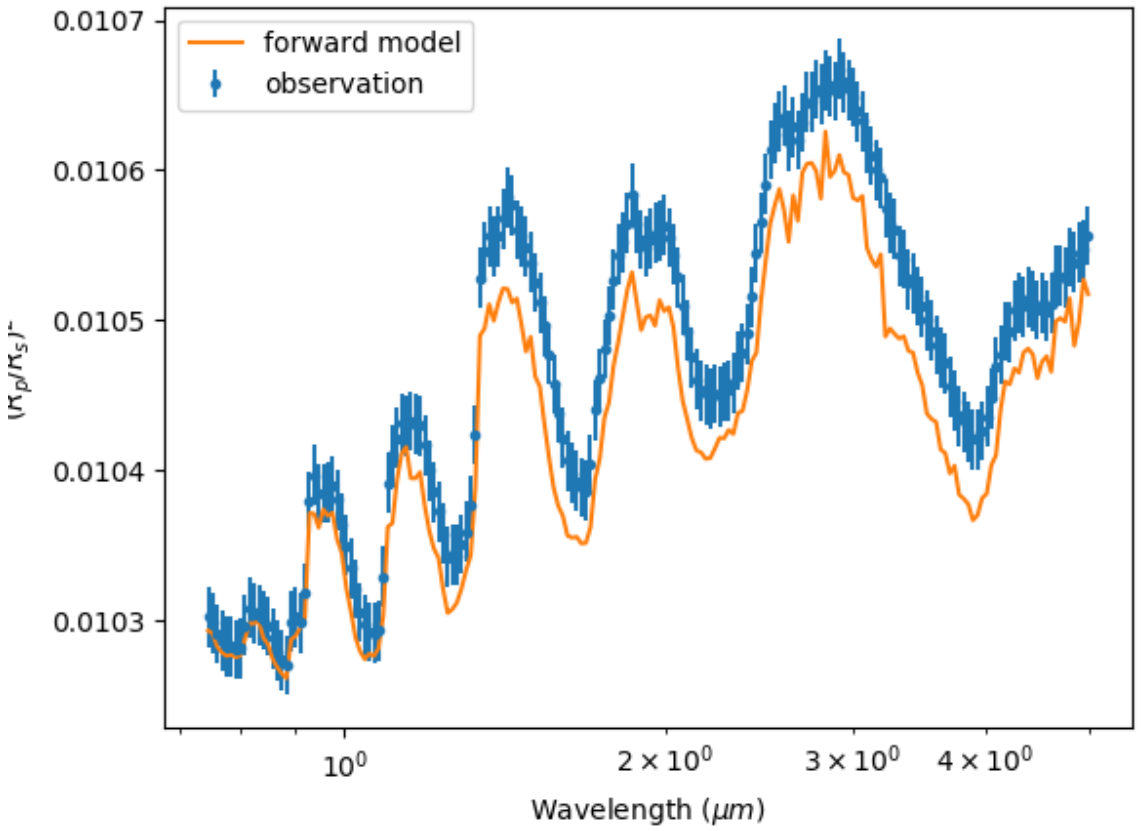
Much much better! We're still missing something though...

5.1.5 Contributions

It seems moelcular absorption is not the only process happening in the atmosphere. Looking at the shorter wavelengths we see the characteristic behaviour of **Rayleigh scattering** and a little from **collisionally induced absorption**. We can

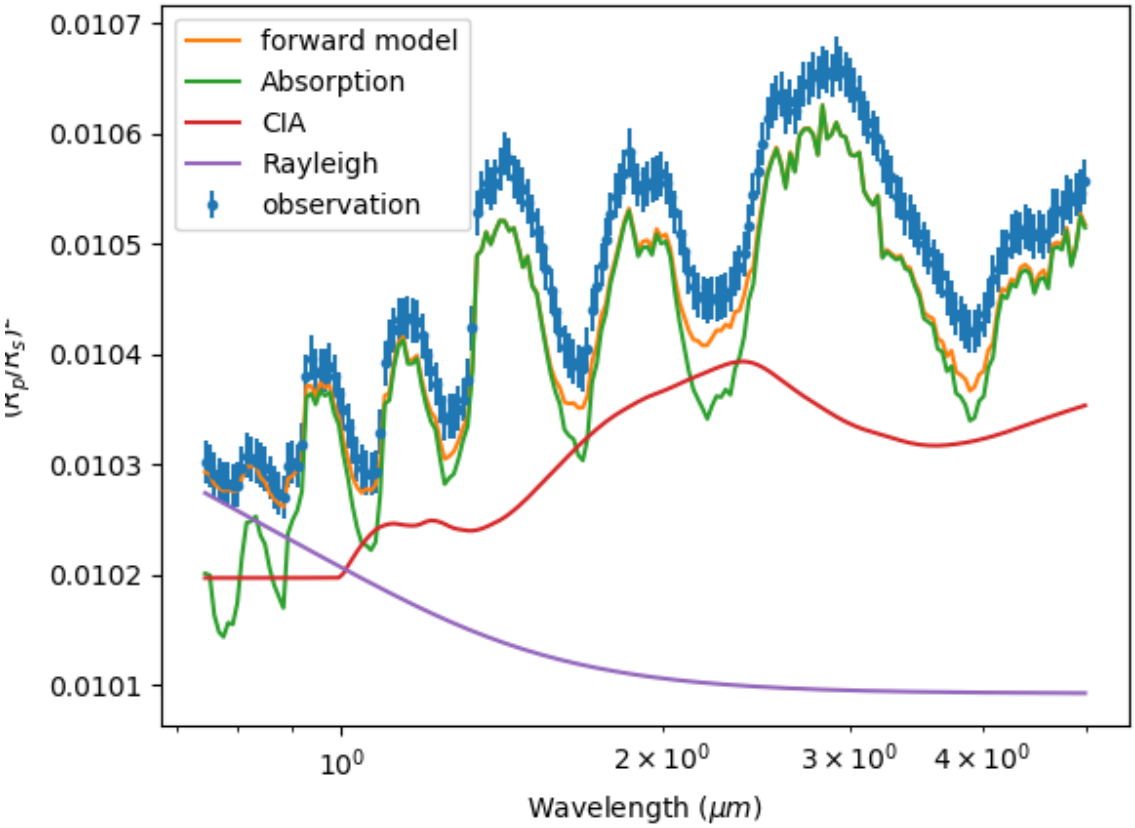
easily add these contributions under the `[Model]` section of the input file. Each *contribution* is represented as a sub-header with additional arguments if necessary. By default we have contributions from molecular `[[Absorption]]` Lets add in some `[[CIA]]` from `H2-H2` and `H2-He` and `[[Rayleigh]]` scattering to the model:

```
[Model]
model_type = transmission

    [[Absorption]]

    [[CIA]]
    cia_pairs = H2-He,H2-H2

    [[Rayleigh]]
```



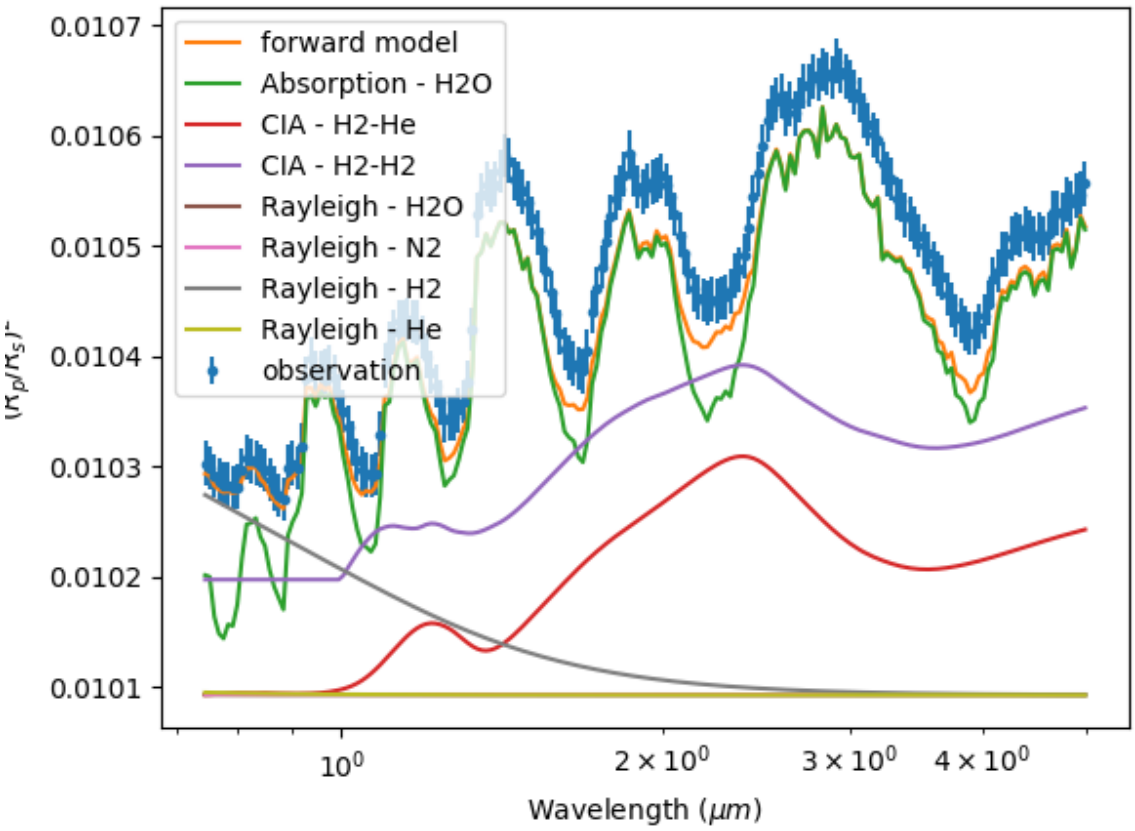
Hey not bad!! It might be worth seeing how each of these processes effect the spectrum. Easy, we can run `taurex` with the `-c` argument which plots the basic contributions:

```
taurex -i quickstart.par --plot -c
```



If you want a more detailed look of the each contribution you can use the `-C` option instead:

```
taurex -i quickstart.par --plot -C
```



Pretty cool. We're almost there. Lets save what we have now to file.

5.1.6 Storage

`Taurex3` uses the HDF5 format to store its state and results. We can accomplish this by using the `-o` output argument:

```
taurex -i quickstart.par -o myfile.hdf5
```

We can use this output to plot profiles spectra and even the optical depth! Try:

```
taurex-plot -i myfile.h5 -o fm_plots/ --all
```

To plot everything:

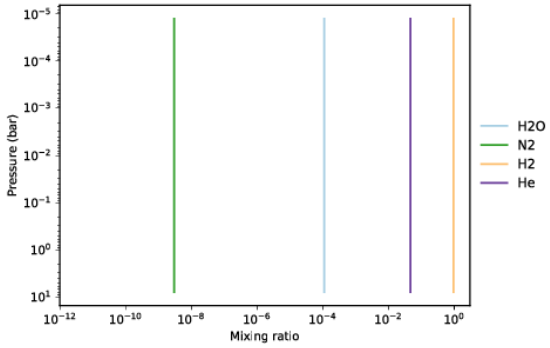


Fig. 3: Chemistry

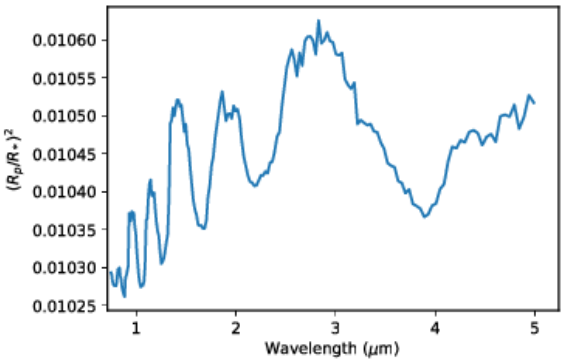


Fig. 4: Spectrum

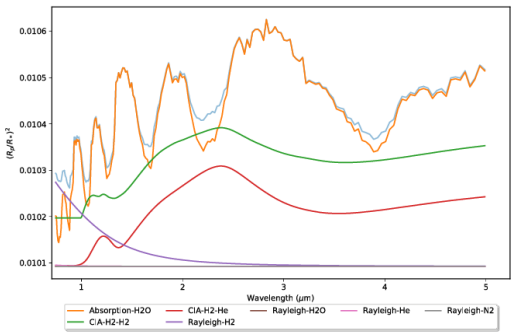


Fig. 5: Contributions

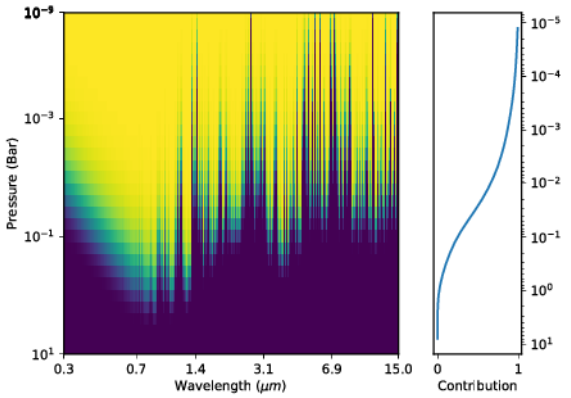


Fig. 6: Optical depth

HDF5 has many viewers such as HDFView or HDFCompass and APIs such as Cpp, FORTRAN and Python. Pick your poison.

5.1.7 Retrieval

So we're close to the observation but not quite there and I suspect its the temperature profile. We should try running a retrieval. We will use nestle as our optimizer of choice but other brands are available. This has already be setup under the `[Optimizer]` section of the input file so we will not worry about it now. We now need to inform the optimizer what parameters we need to fit. The `[Fitting]` section should list all of the parameters in our model that we want (or dont want) to fit and *how* to go about fitting it. By default the `planet_radius` parameter is fit when no section is provided, we should start by creating our `[Fitting]` section and disabling the `planet_radius` fit:

```
[Fitting]
planet_radius:fit = False
```

the syntax is pretty simple, its essentially `parameter_name:option` with `option` being either `fit`, `bounds` and `mode`. `fit` is simply tells the optimizer whether to fit the parameter, `bounds` describes the parameter space to optimize in and `mode` instructs the optimizer to fit in either `linear` or `log` space. The parameter we are interested in is isothermal temperature which is represented as `T`, and we will fit it within *1200 K* and *1400 K*:

```
[Fitting]
planet_radius:fit = False
```

(continues on next page)

(continued from previous page)

```
T:fit = True
T:bounds = 1200.0,1400.0
```

We don't need to include `mode` as by default `T` fits in linear space. Some parameters such as abundances fit in log space by default.

Running taurex like before will just plot our forward model. To run the retrieval we simply add the `--retrieval` keyword like so:

```
taurex -i quickstart.par --plot -o myfile_retrieval.hdf5 --retrieval
```

We should now see something like this pop up:

```
taurex.Nestle - INFO - ------------------------------------
taurex.Nestle - INFO - ------Retrieval Parameters-----------
taurex.Nestle - INFO - ------------------------------------
taurex.Nestle - INFO -
taurex.Nestle - INFO - Dimensionality of fit: 1
taurex.Nestle - INFO -
taurex.Nestle - INFO -
Param      Value    Bound-min    Bound-max
-------  -------  -----------  -----------
T        1265.98         1200         1400


taurex.Nestle - INFO -
it=   393 logz=1872.153686niter: 394
```

It should only take a few minutes to run. Once done we should get an output like this:

```
---Solution 0------
taurex.Nestle - INFO -
Param        MAP    Median
-------  -------  --------
T        1375.97   1371.71
```

So the temperature should have been around *1370 K*, huh, and lets see how it looks. Lets plot the output:

```
taurex-plot -i myfile_retrieval.hdf5 -o retrieval_plots/ --all
```

Our final spectrum looks like:

We can then see the posteriors:

Thats the basics of playing around with TauREx 3. You can try modifying the quickstart to do other things! Take a look at *Input File Format* to see a list of parameters you can change!

5.2 Input File Format

5.2.1 Headers

The input file format is fairly simple to work with. The extension for Taurex3 input files is `.par` however this is generally not enforced by the code. The input is defined in various *headers*, with each header having variables that can be set:

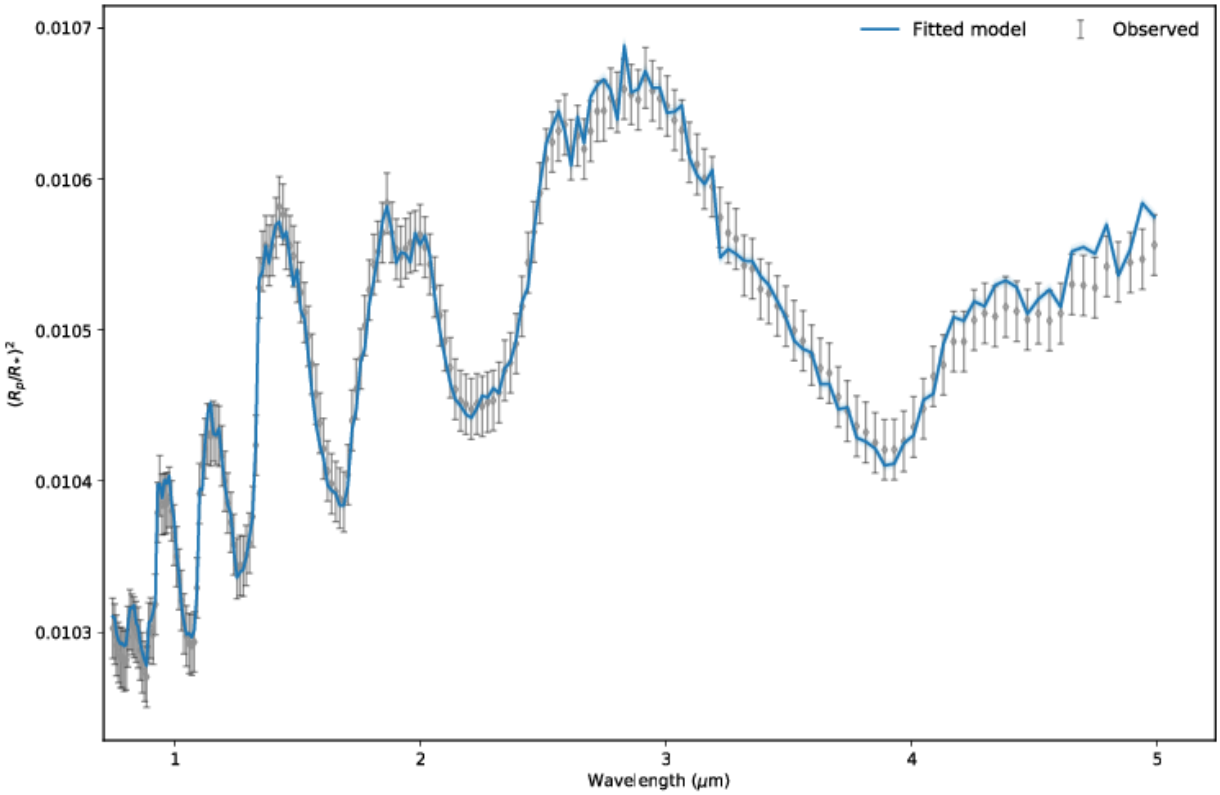


Fig. 7: Final result

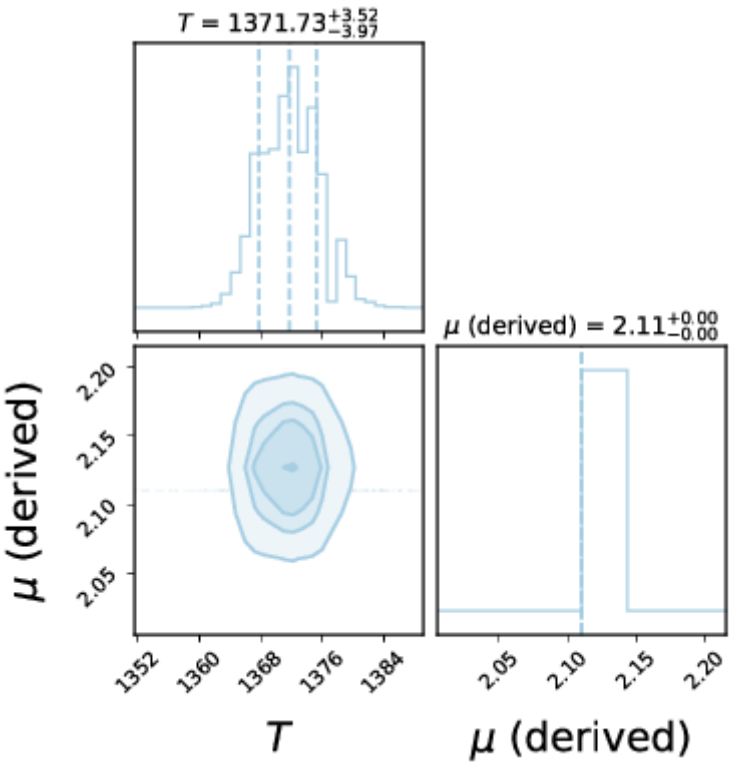


Fig. 8: Posteriors

```
[Header]
parameter1 = value
parameter2 = anothervalue

[Header2]
parameter1 = adifferentvalue
```

Of course comments are handled with #

The available headers are:

- *[Global]*
- *[Chemistry]*
- *[Temperature]*
- *[Pressure]*
- *[Planet]*
- *[Star]*
- *[Model]*
- *[Observation]*
- *[Binning]*
- *[Instrument]*

- *[Optimizer]*
- *[Fitting]*

Not all of these headers are required in an input file. Some will generate default profiles when not present. To perform retrievals, *[Observation]*, *[Optimizer]* and *[Fitting]* *MUST* be present

Some of these may define additional *subheaders* given by the `[[Name]]` notation:

```
[Header]
parameter1 = value
    [[Subheader]]
    parameter2 = anothervalue
```

5.2.2 Variables

String variables take this form:

```
#This is valid
string_variable = Astringvariable
#This is also valid
string_variable_II = "A string variable"
```

Floats and ints are simply:

```
my_int_value = 10
```

And lists/arrays are defined using commas:

```
my_int_list = 10,20,30
my_float_list = 1.1,1.4,1.6,
my_string_list = hello,how,are,you
```

5.2.3 Dynamic variables

The input file is actually a dynamic format and its available variables can change depending on the choice of certain profiles and types. For example lets take the `[Temperature]` header, it contains the variable `profile_type` which describes which temperature profile to use. Setting this to `isothermal` gives us the `T` variable which defines the isothermal temeprature:

```
[Temperature]
profile_type = isothermal
T = 1500.0
```

Now if we change the profile type to `guillot2010` it will use the Guillot 2010 temperature profile which gives access to the variables `T_irr`, `kappa_irr`, `kappa_v1`, `kappa_v2` and `alpha` instead:

```
[Temperature]
profile_type = guillot2010
T_irr=1500
kappa_irr=0.05
kappa_v1=0.05
kappa_v2=0.05
alpha=0.005
```

However setting `T` will throw an error as it doesn't exist anymore:

```
[Temperature]
profile_type = guillot2010
#Error is thrown here
T=1500
kappa_irr=0.05
```

This also applies to fitting parameters, profiles provide certain fitting parameters and changing the model means that these parameters may not exist anymore.

5.2.4 Mixins

New in version 3.1.

Mixins can be applied to any base component through the + operator:

```
[Temperature]
profile_type = mixin1+mixin2+base
```

Where we apply `mixin1` and `mixin2` to a `base`. Including mixins will also include their keywords as well. If `mixin1` has the keyword `param1`, `mixin2` has `param2` and `base` has `another_param` then we can define in the input file:

```
[Temperature]
profile_type = mixin1+mixin2+base
param1 = "Hello"        # From mixin 1
param2 = "World"        # From mixin 2
another_param = 10.0    # From base
```

Mixins are evaluated in reverse, the last must be a *non-mixin* for example if we have a `doubler` mixin that doubles temperature profiles then this is valid:

```
[Temperature]
profile_type = doubler+isothermal
```

but this is *not valid*:

```
[Temperature]
profile_type = isothermal+doubler
```

Additionally we cannot have more than one base class so this is *invalid*:

```
[Temeprature]
profile_type = doubler+isothermal+guillot
```

The reverse evaluation means that the first mixin will be *applied last*. If we have another mixin called `add50` which adds 50 K to the profile, then:

```
[Temperature]
profile_type = doubler+add50+isothermal
T = 1000
```

Will result in a temperature profile of 2100 K. If we instead do this:

```
[Temperature]
profile_type = add50+doubler+isothermal
T = 1000
```

Then the resultant temperature will be 2050 K.

5.3 Custom Types

5.3.1 Direct Method

Across many of the atmospheric parameters/sections you'll come across the `custom` type. These allow you to inject your own code to be used in the forward model and retrieval scheme.

Developing or wrapping your own parameters is discussed in the *Developers guide*.

Lets take a simple example. Imagine you have your own amazing temperature profile and you've written a TauREx 3 class for it:

```
from taurex.temperature import TemperatureProfile
from taurex.core import fitparam
import numpy as np


class RandomTemperature(TemperatureProfile):

    def __init__(self, base_temp=1500.0,
                 random_scale=10.0):
        super().__init__(self.__class__.__name__)

        self._base_temp = base_temp
        self._random_scale = random_scale

    # -----Fitting Parameters--------------

    @fitparam(param_name='rand_scale',param_latex='rand')
    def randomScale(self):
        return self._random_scale

    @randomScale.setter
    def randomScale(self, value):
        self._random_scale = value

    @fitparam(param_name='base_T',param_latex='$T_{base}$')
    def baseTemperature(self):
        return self._base_temp

    @baseTemperature.setter
    def baseTemperature(self, value):
        self._base_temp = value

    # -------Actual calculation -----------

    @property
    def profile(self):
        return self._base_temp + \
                    np.random.rand(self.nlayers) * self._random_scale
```

Ok ok this is a **terrible** temperature profile, essentially it is randomizing around a *base* temperature given but I digress. We can easily include it in taurex by pointing it to the file:

```
[Temperature]
profile_type = custom
python_file = /path/to/rand_temperature.py
```

Thats it!! When you change a type (i.e `profile_type`, `model_type` etc.) to `custom` the new keyword `python_file` is available which should point to the python file with the class you want. We can run TauREx3 with it and see that it has indeed accepted it:

```
taurex -i input.par -o randtemp_test.h5
taurex-plot -i randtemp_test.h5 -o myplots/ --plot-tpprofile
```

Which gives:

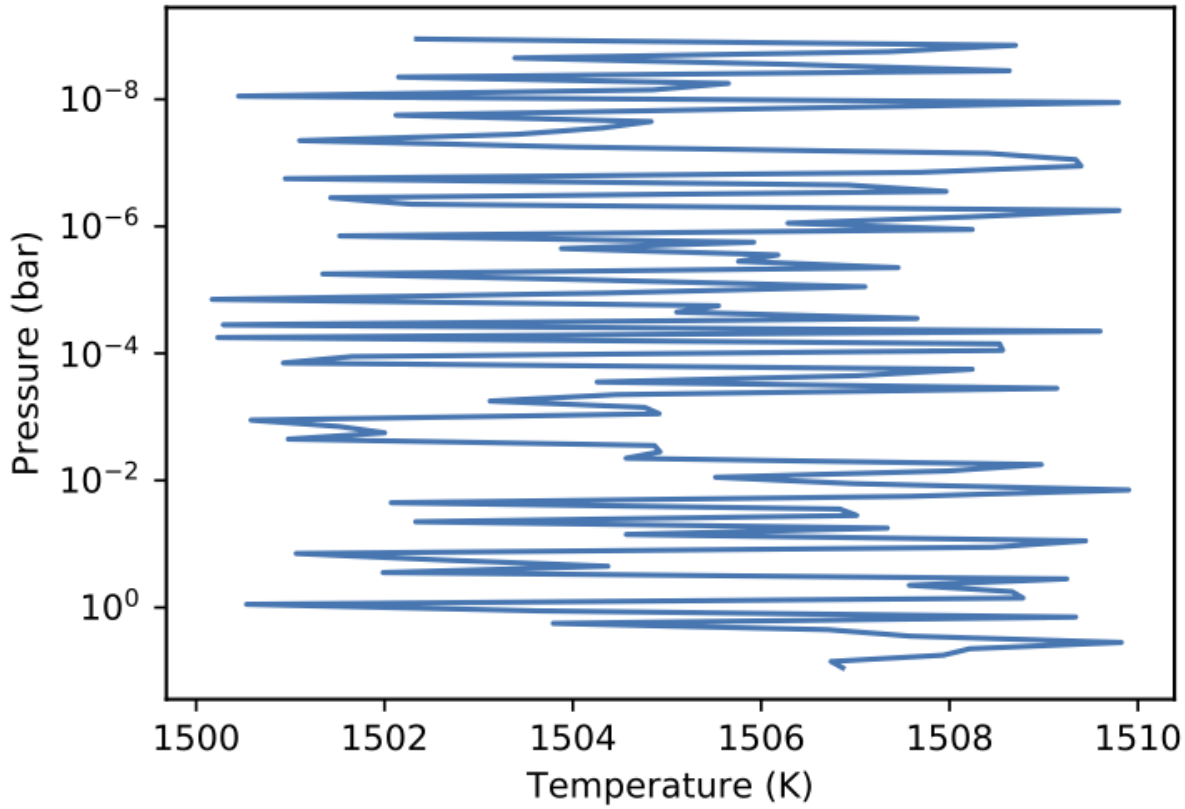


Fig. 9: Truly terrible

Now we can do a little more with this as well. When TauREx3 is given a new class it will scan for initialization keywords and *embed them as new input keywords*. Looking at the class, the initialization keywords are `base_temp` and `random_scale` this means we can put them as parameters in the input file:

```
[Temperature]
profile_type = custom
python_file = /path/to/rand_temperature.py
base_temp = 500.0
random_scale = 100.0
```

And plotting again we see that the profile has now changed to reflect this:

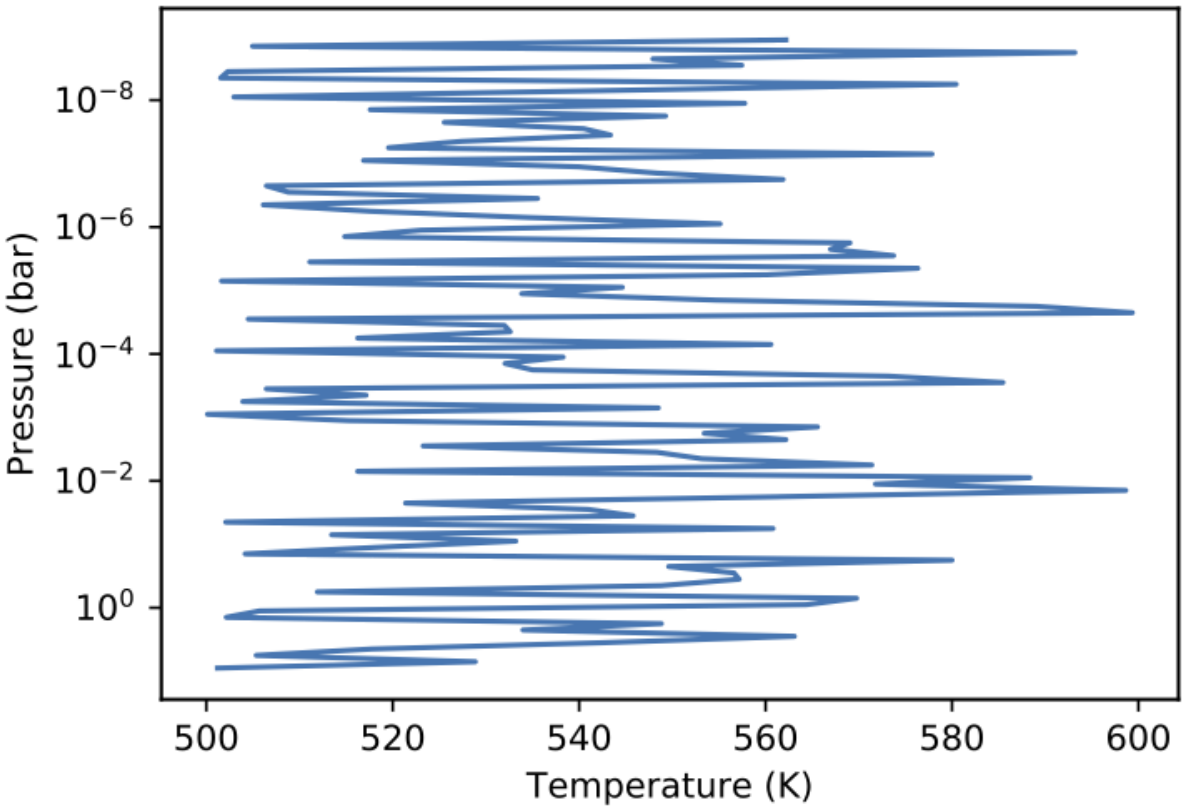


Fig. 10: Truly terrible at 500.0 K

Finally, it is entriely possible to perform retrievals with our new profile, since TauREx3 will also *discover new fitting parameters*. Our profile has the fitting parameters `base_T` and `rand_scale` so we can add them to our `[Fitting]` section:

```
[Fitting]
planet_radius:fit = True
planet_radius:bounds = 0.8, 2.0

base_T:fit = True
base_T:bounds = 500.0, 3000.0
rand_scale:mode = log
rand_scale:fit = True
rand_scale:bounds = 1e-10, 1000.0
```

Of course we get all the benefits of native fitting parameters like the ability to switch between `linear` and `log` scale. Now we can perform a retrieval and plot posteriors like so:

```
taurex -i input.par -o randtemp_retrieval.h5 --retrieval
taurex-plot -i randtemp_retrieval.h5 -o myplots_retrieval/ --plot-posteriors
```

Which correctly adds in the latex parameters as well, it even inserted *log* for us! Of course the retrieval just went ahead and tried to minimize the randomness which makes sense! Almost all parameters have some custom functionality. The ones that do not have this are `[Binning]` and `[Global]`. Try it out!

Here is the full `input.par` file:

```
[Global]
xsec_path = /path/to/xsecfiles
cia_path = /path/to/ciafiles


# ----Forward Model related -----------

[Chemistry]
chemistry_type = taurex
fill_gases = H2,He
ratio = 4.8962e-2

    [[H2O]]
    gas_type = constant
    mix_ratio=1.1185e-4

    [[N2]]
    gas_type = constant
    mix_ratio = 3.00739e-9



[Temperature]
profile_type = custom
python_file = rand_temperature.py
base_temp = 1000.0
random_scale = 100.0

[Pressure]
profile_type = Simple
```

(continues on next page)

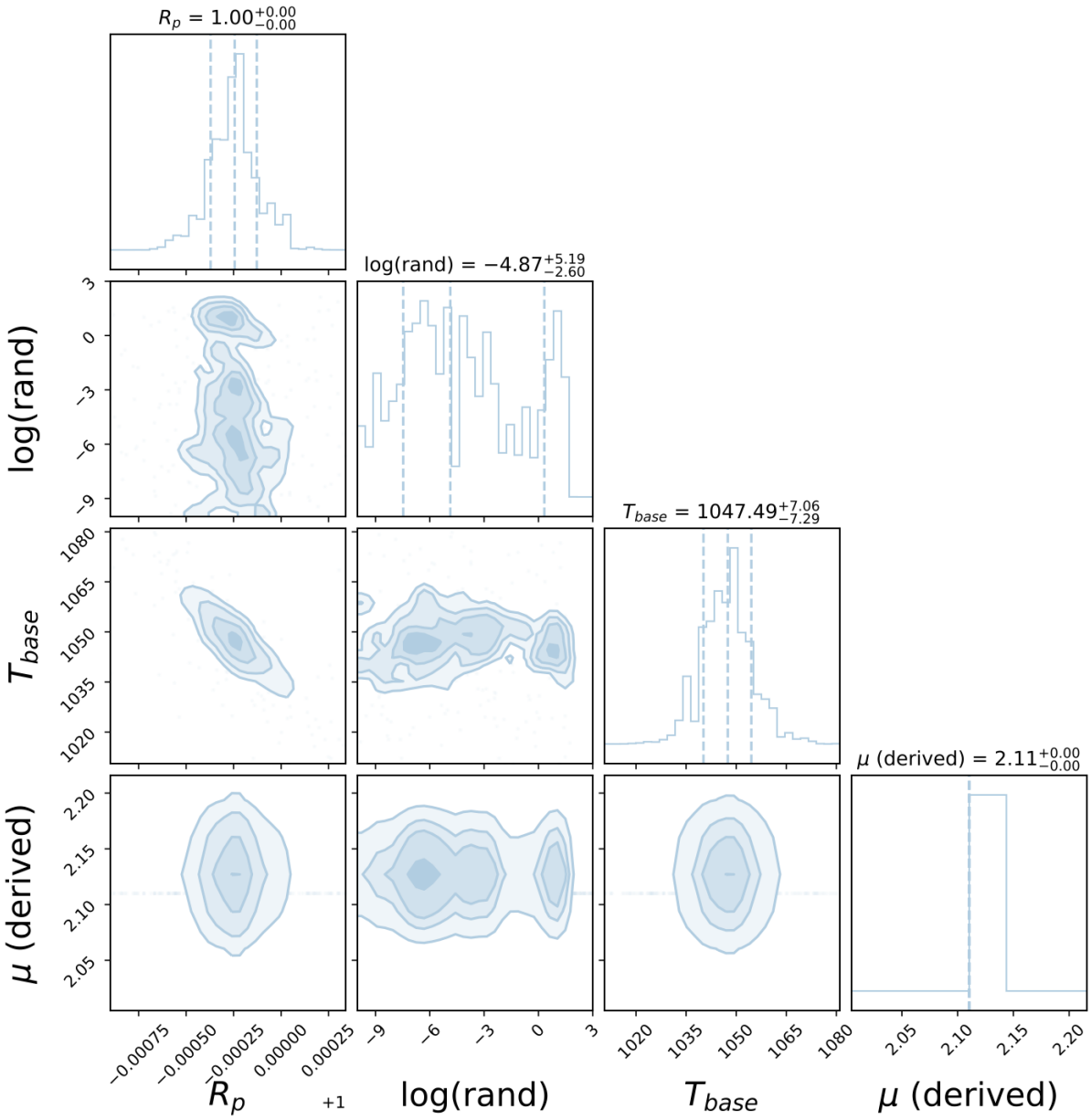


Fig. 11: Truly terrible posteriors

(continued from previous page)

```
atm_min_pressure = 1e-4
atm_max_pressure = 1e6
# Use 10 layers to keep retrieval time down
nlayers = 10

[Planet]
planet_type = Simple
planet_mass = 1.0
planet_radius = 1.0

[Star]
star_type = blackbody

[Model]
model_type = transmission

    [[Absorption]]

    [[CIA]]
    cia_pairs = H2-He,H2-H2

    [[Rayleigh]]

# ---------Creating an observation for retrieval--------
# We use instruments to create an observation
# Rather than passing in a text file

[Binning]
bin_type = manual
accurate = False
wavelength_res = 0.6,4.1,100 # Start end

[Instrument]
instrument = snr
SNR = 20

[Observation]
taurex_spectrum = self

# ------Retrieval related --------------

[Optimizer]
optimizer = nestle
# Use small number of live points to minimize
# retrieval time
num_live_points = 50

[Fitting]
planet_radius:fit = True
planet_radius:factor = 0.8, 2.0

base_T:fit = True
base_T:bounds = 500.0, 3000.0
rand_scale:mode = log
rand_scale:fit = True
rand_scale:bounds = 1e-10, 1000.0
```

5.3.2 Extension Path Method

Another way to include your own code is by setting the `extension_path` variable under *[Global]*. If our python file exists in a folder say:

```
mycodes/
    rand_temperature.py
```

We can set the path to `extension_path` variable to point to the folder:

```
[Global]
extension_path = /path/to/mycodes/
```

We will need to make one small modification and add the `input_keywords` class method to our temperature profile. (See *Basics*):

```
@classmethod
def input_keywords(cls):
    return ['my-random-temperature',]
```

TauREx will now search for `.py` files in the directory, attempt to load them and then automatically integrate them into the TauREx pipeline!!! We can use the value return by `input_keywords` to select our profile:

```
[Temperature]
profile_type = my-random-temperature
base_temp = 1000.0
random_scale = 100.0
```

Cool!!!

5.3.3 Limitations

The custom system is intended for quick development and inclusion of new components or file formats. There are as few limitations when using it.

First each file is loaded in isolation, therefore referencing another python file in the same directory will yield errors, for example if we have this directory:

```
mycodes/
    rand_temperature.py
    util.py
```

And we attempt to import `util` in `rand_temperature.py` then it will fail.

The *Direct Method* does not support loading in *`Opacity`* and *`Contribution`* types.

If you feel like you need more control and flexibility with your extensions or if it is useful to the community as a whole then we suggest trying *Plugin Development*

5.4 `[Global]`

The global section generally handles settings that affect the whole program.

- **`xsec_path`**
  - str or list of str

  - Defines the path(s) that contain molecular cross-sections
  - e.g `xsec_path = path/to/xsec`
- **xsec_interpolation**
  - `exp` or `linear`
  - Defines whether to use exponential or linear interpolation for temperature
  - e.g `xsec_interpolation = exp`
- **in_memory**
  - `True` or `False`
  - For HDF5 opacities. Determines if streamed from file (False) or loaded into memory (True)
  - Default is `True`
  - e.g `in_memory = true`
- **cia_path**
  - str or list of str
  - Defines the path(s) that contain CIA cross-sections
  - e.g `cia_path = path/to/xsec`
- **ktable_path**
  - str or list of str
  - Defines the path(s) that contain k-tables
  - e.g `ktable_path = path/to/ktables`
- **opacity_method**
  - Either `xsec` or `ktables`
  - Choose whether to use molecular cross-sections or correlated k method.
  - e.g `opacity_method = ktables`
- **mpi_use_shared**
  - `True` or `False`
  - Exploit MPI 3.0 shared memory to significantly reduce memory usage per node
  - When running under MPI, will only allocate arrays once in a node rather than each process
  - Works on allocations that use this feature (i.e pickle and HDF5 opacities)
  - e.g `mpi_use_shared = True`

5.5 [Chemistry]

This header describes the chemical composition of the atmosphere. The type of model used is defined by the `chemistry_type` variable.

The available `chemistry_type` are:

- **ace**
  - ACE equlibrium chemistry

  - Class: `ACEChemistry`

- **`taurex`**
  - Free chemistry
  - Class: *`TaurexChemistry`*

- **`custom`**
  - User-type chemistry. See *Custom Types*

5.5.1 ACE Equlibrium Chemistry

Warning: Since version 3.1 this has been removed from the base TauREx package. You can restore this chemical scheme by writing:

```
pip install taurex_ace
```

`chemistry_type = ace` `chemistry_type = equilibrium`

Equilibrium chemistry using the ACE FORTRAN program. *Fortran compiler required*

Keywords

Variable	Type	Description	Default Value
`metallicity`	`float`	Stellar metallicity in solar units	1.0
`co_ratio`	`float`	C/O ratio	0.54951

Fitting Parameters

Parameter	Type	Description
`ace_metallicity`	`float`	Stellar metallicity in solar units
`ace_co`	`float`	C/O ratio

5.5.2 Taurex Chemistry

`chemistry_type = taurex` `chemistry_type = free`

This chemistry type allows you to define individual abundance profiles for each molecule. Molecules are either active or inactive depending on whats available. If no cross-sections are available then the moelcule is not actively absorbing.

Keywords

Variable	Type	Description	Default
`fill_gases`	`str` or `list`	Gas or gases to fill the atmosphere with	H2,He,
`ratio`	`float` or `list`	Ratio between first fill gas and every other fill gas	0.749

Fitting Parameters

On its own, this chemistry model provides fitting parameters relating to the fill gases used. These are only created when more than one fill gas is defined. Here, we use `[Gas-0]` to designate the first gas defined in the fill gas list and `[Gas-(number)]` to designate the nth gas after the main gas. If we have a gas list like:

```
fill_gases = H2,He,CO2,
```

then `[Gas-1]_[Gas-0] == He_H2` and `[Gas-2]_[Gas-0] == CO2_H2`:

Parameter	Type	Description
`[Gas-(n)]_[Gas-0]`	`float`	Ratio of nth fill gas vs first fill gas

However molecules are defined as *subheaders* with the subheader being the name of the molecule. Each molecule can be assigned an abundance profile through the `gas_type` variable. For example, to describe a chemical profile with water in constant abundance in the atmosphere is simply done like so:

```
[Chemistry]
chemistry_type = taurex
fill_gases = H2,He,
ratio = 0.1524

    [[H2O]]
    gas_type = constant
    mix_ratio = 1e-4
```

For each molecule, the available `gas_type` are:

- **`constant`**
  - Constant abundance profile
  - Class: *`ConstantGas`*
- **`twopoint`**
  - Two Point abundance profile
  - Class: `TwoPointGas`
- **`twolayer`**
  - Two layer abundance profile
  - Class: *`TwoLayerGas`*

5.5.3 Gas Profiles

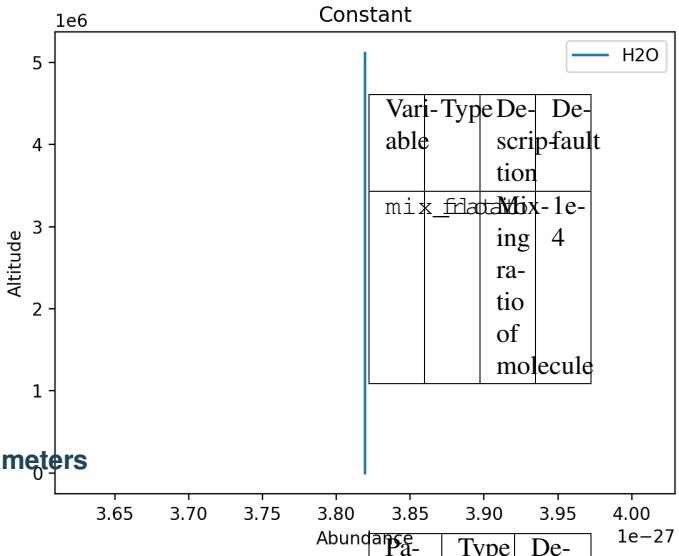
For these profiles, the fitting parameters generated have the name associated with the name of the molecule. For example: `H2O_P`, `CH4_S` etc. Because of this, we will use the moniker: `[Mol]`. Replacing this with the appropriate molecule will give you the correct fitting parameter name. e.g. `[Mol]_surface` should be `H2O_surface` for water etc.

5.5.4 Constant Profile

`gas_type = constant`

An abundance profile that is constant with height of the atmosphere

Keywords



Variable	Type	Description	Default
mix_fraction		Mixing ratio of molecule	1e-4

Fitting Parameters

Parameter	Type	Description
[Mol]	float	Mixing ratio of molecule

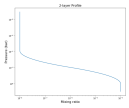
5.5.5 Two Layer Profile

`gas_type = twolayer`

An abundance profile where abundance is defined

on the planet surface and top of the atmosphere with a pressure point determining the boundary between the layers. Smoothing is applied.

Keywords



Variable	Type	Description	Default
`mix_ratio_surface`	`float`	Mixing ratio at BOA	1e-4
`mix_ratio_top`	`float`	Mixing ratio at TOA	1e-8
`mix_ratio_P`	`float`	Pressure boundary (Pa)	1e3
`mix_ratio_smoothing`	`int`	Smoothing window	10

Fitting Parameters

Parameter	Type	Description
`[Mol]_surface`	`float`	Mixing ratio at BOA
`[Mol]_top`	`float`	Mixing ratio at TOA
`[Mol]_P`	`float`	Pressure boundary (Pa)

5.5.6 Chemsitry File

`chemistry_type = file`

Reads a multi-column text file. Order must be from BOA to TOA. Each column must represent a unique molecule.

Keywords

Variable	Type	Description	Default
`filename`	`str`	Path to chemistry file	None
`gases`	`list`	List of all molecules in column order	None

5.6 `[Temperature]`

This header is used to define temperature profiles for the atmosphere. The type of temperature profile is defined by the `profile_type` variable

The available `profile_type` are:

- **`isothermal`**

  - Isothermal temperature profile
  - Class: *Isothermal*
- **guillot2010**
  - TP profile from Guillot 2010, A&A, 520, A27
  - Class: *Guillot2010*
- **npoint**
  - N-point temperature profile
  - Class: *NPoint*
- **rodgers**
  - Layer-by-layer temperature - pressure profile
  - Class: *Rodgers2000*
- **file**
  - Temperature profile from file
  - Class: *TemperatureFile*
- **custom**
  - User-type temperature. See *Custom Types*

    More profiles can also be included using *Plugins*

5.6.1 Isothermal Profile

`profile_type = isothermal`

Constant temperature throughout atmosphere

Keywords

Variable	Type	Description	Default
`T`	`float`	Temperature in Kelvin	1500

Fitting Parameters

Parameter	Type	Description
`T`	`float`	Temperature in Kelvin

Examples

Example isothermal profile:

```
[Temperature]
profile_type
↪= isothermal
T = 1500
```

5.6.2 Guillot 2010 Profile

`profile_type = guillot`

TP profile from Guillot 2010, A&A, 520, A27 (equation 49) Using modified 2stream approx. from Line et al. 2012, ApJ, 749,93 (equation 19)

Keywords

Variable	Type	Description	Default
`T_irr`	`float`	Planet equilibrium temperature (K)	1500
`kappa_ir`	`float`	mean infra-red opacity	0.01
`kappa_v1`	`float`	mean optical opacity one	0.005
`kappa_v2`	`float`	mean optical opacity two	0.005
`alpha`	`float`	ratio between `kappa_v1` and `kappa_v2`	0.5

Fitting Parameters

Parameter	Type	Description
`T_irr`	`float`	Planet equilibrium temperature (K)
`kappa_ir`	`float`	mean infra-red opacity
`kappa_v1`	`float`	mean optical opacity one
`kappa_v2`	`float`	mean optical opacity two
`alpha`	`float`	ratio between `kappa_v1` and `kappa_v2`

Examples

Example Guillot profile:

```
[Temperature]
profile_type
↪= guillot
T_irr = 1500
```

(continues on next page)

(continued from previous page)

```
kappa_ir = 0.01
kappa_
↪v1 = 0.002
kappa_
↪v2 = 0.003
alpha = 0.3
```

5.6.3 N-Point Profile

`profile_type = npoint`

Temperature defined at various heights in the atmosphere. Smoothing is then applied. If no temperature and pressure points are defined, it is equivalent to a 2-point profile. Including 1 makes it a 3-point and so on. Each temperature point must have an associated pressure point and vica versa.

Keywords

Variable	Type	Description	Default
`T_surface`	`float`	Temperature at `P_surface` in K	1500
`T_top`	`float`	Temperature at `P_top` in K	200
`P_surface`	`float`	Pressure at `T_surface` in Pa. Set to -1 for BOA	-1
`P_top`	`float`	Pressure at `T_top` in Pa. Set to -1 for TOA	-1
`temperature_points`	`list`	Temperature points between BOA and TOA	
`pressure_points`	`list`	Pressure in Pa for each temperature point	
`smoothing_window`	`int`	Smoothing width	10

Fitting Parameters

Fitting parameters

are generated for each `temperature_point` and `pressure_point` defined. They start from 1 and have the form `T_point1`, `P_point1`, `T_point2`, `P_point2` etc.

Variable	Type	Description
`T_surface`	`float`	Temperature at `P_surface` in K
`T_top`	`float`	Temperature at `P_top` in K
`P_surface`	`float`	Pressure at `T_surface` in Pa.
`P_top`	`float`	Pressure at `T_top` in Pa.
`T_point(n)`	`float`	Temperature point (n). Starts from 1
`P_point(n)`	`float`	Pressure point (n). Starts from 1

5.6.4 Rodgers 2000 Profile

`profile_type = rodgers`

Layer-by-layer temperature - pressure profile retrieval using dampening factor Introduced in Rodgers (2000): Inverse Methods for Atmospheric Sounding (equation 3.26)

Keywords

Variable	Type	Description	Default
`temperature_layers`	`list`	Temperature in Kelvin for each layer	None
`correlation_length`	`float`	Correlation length	5.0

Fitting Parameters

Warning: For a 100 layer atmosphere, this will create 100 fitting parameters for `T_(n)` which might be very unwieldly to use and fitting them all could lead to a very long sample time.

Parameter	Type	Description
`T_(n)`	`float`	Temperature for layer (n)
`corr_length`	`float`	Correlation length

5.6.5 Temperature File

`profile_type = file`

Reads a text file. Can support multi column files with any units

If a pressure column is provided then it will interpolate the temperature based on the pressure. If no pressure is provided then it will assume index 0 is BOA and the last index is TOA and interpolate according to that.

Keywords

Variable	Type	Description	Default
filename	str	Path to temperature file	None
skiprows	int	No. of rows to ignore	0
temp_col	int	Column number of temperature (0-based)	0
press_col	int	Column number of pressure if available (0-based)	None
temp_units	str	Units of temperature (based on astropy format)	K
press_units	str	Units of pressure (based on astropy format)	Pa
delimiter	str	Delimiter used in file. None means whitespace	None
reverse	bool	False = BOA-TOA, True = TOA-BOA	None

5.7 [Pressure]

The header describes pressure profiles for the atmosphere. Currently only one type of profile is supported, so profile_type=simple or profile_type=hydrostatic must be included. profile_type = custom is also valid, See *Custom Types*

Class
SimplePressureProfile

5.7.1 Keywords

Variable	Type	Description	Default
atm_min_pressure	float	Pressure in Pa at TOA	1e0
atm_max_pressure	float	Pressure in Pa at BOA	1e6
nlayers	int	Number of layers	100

5.7.2 Fitting Parameters

Warning: Whilst included of completeness it is generally not a good idea to fit these parameters as it can drastically alter the scale of the atmosphere.

Variable	Type	Description
`atm_min_pressure`	`float`	Pressure in Pa at TOA
`atm_max_pressure`	`float`	Pressure in Pa at BOA
`nlayers`	`int`	Number of layers

Examples

A basic pressure profile:

```
[Pressure]
profile_type = simple
atm_min_pressure = 1e-3
atm_max_pressure = 1e6
nlayers = 100
```

5.8 `[Planet]`

This header is used to define planetary properties. Currently, only `planet_type = simple` is supported and must be included. `planet_type = custom` is also valid, See *Custom Types*

Class `Planet`

5.8.1 Keywords

Variable	Type	Description	Default
planet_mass	float	Mass in Jupiter mass	1.0
planet_radius	float	Radius in Jupiter radius	1.0
planet_distance	float	Semi-major-axis in AU	1.0
impact_param	float	Impact parameter	0.5
orbital_period	float	Orbital period in days	2.0
albedo	float	Planetary albedo	0.3
transit_time	float	Transit time in seconds	3000.0

5.8.2 Fitting Parameters

Parameter	Type	Description
planet_mass	float	Mass in Jupiter mass
planet_radius	float	Radius in Jupiter radius
planet_distance	float	Semi-major-axis in AU

Examples

Planet with 1.5 Jupiter mass and 1.2 Jupiter radii:

```
[Planet]
planet_
↪type = simple
planet_
↪mass = 1.5
planet_
↪radius = 1.2
```

5.9 [Star]

This header describes the parent star of the exo-planet. The star_type informs the type of spectral emission density (SED) used in the emission and direct image forward model. The star_type available are:

- **blackbody**
  - Star with a blackbody SED
  - Class *BlackbodyStar*
- **phoenix**
  - Uses the PHOENIX library for the SED
  - *PhoenixStar*
- **custom**
  - User-provided star model. See *Custom Types*

5.9.1 Blackbody

`star_type = blackbody`

Star is considered a blackbody.

Keywords

Variable	Type	Description	Default
`temperature`	`float`	Effective temperature in K	5000
`radius`	`float`	Radius in solar radius	1.0
`mass`	`float`	Mass in solar mass	1.0
`distance`	`float`	Distance from Earth in pc	1.0
`metallicity`	`float`	Metallicity in solar units	1.0
`magnitudeK`	`float`	Magnitude in K-band	10.0

Examples

A Sun like star as a black body:

```
[Star]
star_type␣
↪= blackbody
radius = 1.0
temperature␣
↪= 5800
```

5.9.2 PHOENIX

`star_type = phoenix`

Stellar emission spectrum is read from the PHOENIX library `.fits.gz` files and interpolated to the correct temperature. Any temperature outside of the range provided by PHOENIX will use a blackbody SED instead. The `.fits.gz` filenames must include the temperature as the first number. TauREx3 splits the filename in terms of numbers so any text can be included in the beginning of the file name, therefore these are valid:

```
lte05600.
↪fits.gz ␣
↪# 5600 Kelvin
abunchofothertext-
↪andanother-
↪here05660-
↪0.4_0.5.
↪0.8.fits.gz␣
↪#5660 Kelvin
5700-056-034-
↪0434.fits.gz␣
↪#5700 Kelvin
```

Keywords

Variable	Type	Description	Default
`phoenix_path`	`str`	Path to `.fits.gz` files	**Required**
`temperature`	`float`	Effective temperature in K	5000
`radius`	`float`	Radius in solar radius	1.0
`mass`	`float`	Mass in solar mass	1.0
`distance`	`float`	Distance from Earth in pc	1.0
`metallicity`	`float`	Metallicity in solar units	1.0
`magnitudeK`	`float`	Magnitude in K-band	10.0

Examples

A Sun like star using PHOENIX spectra:

```
[Star]
star_type
↪= phoenix
radius = 1.0
temperature
↪= 5800
phoenix_path
↪= /mypath/
↪to/fitsfiles/
```

5.10 [Model]

This header defines the type of forward model (FM) that will be computed by TauREx3. There are only four distinct forward `model_type`:

- **transmission**
  - Transmission forward model
- **emission**
  - Emission forward model
- **directimage**
  - Direct-image forward model
- **custom**
  - User-type forward model, See *Custom Types*

  Both emission and direct image also include an optional keyword `ngauss` which dictates the number of Gaussian quadrate points used in the integration. By default this is set to `ngauss=4`.

5.10.1 Contributions

Contributions define what processes in the atmosphere contribute to the optical depth. These contributions are defined as *subheaders* with the name of the header being the contribution to add into the forward model.Any forward model type can be augmented with these contributions.

Examples

Transmission spectrum with molecular absorption and CIA from `H2-He` and `H2-H2`:

```
[Model]
model_type =␣
↪transmission
␣
↪␣
↪␣
↪␣
↪[[Absorption]]

    [[CIA]]
    cia_pairs␣
↪= H2-He,He-He
```

Emission spectrum with molecular absorption, CIA and Rayleigh scattering:

```
[Model]
model_type␣
↪= emission
ngauss = 4
␣
↪␣
↪␣
↪␣
↪[[Absorption]]
```

(continues on next page)

(continued from previous page)

```
    [[CIA]]
    cia_pairs␣
↪= H2-He,He-He

  ␣
↪ [[Rayleigh]]
```

The following sections give a list of available contributions

5.10.2 Molecular Absorption

`[[Absorption]]`

Adds molecular absorption to the forward model. Here the *active* molecules contribute to absorption. No other keywords are needed. No fitting parameters.

5.10.3 Collisionally Induced Absorption

`[[CIA]]`

Adds collisionally induced absorption to the forward model. Requires `cia_path` to be set. Both *active* and *inactive* molecules can contribute. No fitting parameters

Keywords

Variable	Type	Description
`cia_pairs`	`list`	List of molecular pairs. e.g. `H2-He, H2-H2`

5.10.4 Rayleigh Scattering

`[[Rayleigh]]`

Adds Rayleigh scattering to the forward model. Both *active* and *inactive* molecules can contribute. No keywords or fitting parameters.

5.10.5 Optically thick clouds

`[[SimpleClouds]]`
or
`[[ThickClouds]]`

A simple cloud model that puts a infinitely absorping cloud deck in the atmosphere.

Keywords

Variable	Type	Description
`clouds_pressure`	`float`	Pressure of top of cloud-deck in Pa

Fitting Parameters

Variable	Type	Description
`clouds_pressure`	`float`	Pressure of top of cloud-deck in Pa

5.10.6 Mie scattering (Lee)

`[[LeeMie]]`

Computes Mie scattering contribution to optical depth Formalism taken from: Lee et al. 2013, ApJ, 778, 97

Keywords

Variable	Type	Description
`lee_mie_radius`	`float`	Particle radius in um
`lee_mie_q`	`float`	Extinction coefficient
`lee_mie_mix_ratio`	`float`	Mixing ratio in atmosphere
`lee_mie_bottomP`	`float`	Bottom of cloud deck in Pa
`lee_mie_topP`	`float`	Top of cloud deck in Pa

Fitting Parameters

Parameter	Type	Description
`lee_mie_radius`	`float`	Particle radius in um
`lee_mie_q`	`float`	Extinction coefficient
`lee_mie_mix_ratio`	`float`	Mixing ratio in atmosphere
`lee_mie_bottomP`	`float`	Bottom of cloud deck in Pa
`lee_mie_topP`	`float`	Top of cloud deck in Pa

5.10.7 Mie scattering (BH)

`[[BHMie]]`

Computes a Mie scattering contribution using method given by Bohren & Huffman 2007

Keywords

Variable	Type	Description
`bh_particle_radius`	`float`	Particle radius in um
`bh_cloud_mix`	`float`	Mixing ratio in atmosphere
`bh_clouds_bottomP`	`float`	Bottom of cloud deck in Pa
`bh_clouds_topP`	`float`	Top of cloud deck in Pa
`mie_path`	`str`	Path to molecule scattering parameters
`mie_type`	`cloud` or `haze`	Type of mie cloud

Fitting Parameters

Parameter	Type	Description
`bh_particle_radius`	`float`	Particle radius in um
`bh_cloud_mix`	`float`	Mixing ratio in atmosphere
`bh_clouds_bottomP`	`float`	Bottom of cloud deck in Pa
`bh_clouds_topP`	`float`	Top of cloud deck in Pa

5.10.8 Mie scattering (Flat)

`[[FlatMie]]`

Computes a flat absorbing region of the atmosphere across all wavelengths

Keywords

Variable	Type	Description
`flat_mix_ratio`	`float`	Opacity value
`flat_bottomP`	`float`	Bottom of absorbing region in Pa
`flat_topP`	`float`	Top of absorbing region in Pa

Fitting Parameters

Parameter	Type	Description
`flat_mix_ratio`	`float`	Opacity value
`flat_bottomP`	`float`	Bottom of absorbing region in Pa
`flat_topP`	`float`	Top of absorbing region in Pa

5.11 [Observation]

This header deals with loading in spectral data for retrievals or plotting.

5.11.1 Keywords

Only one of these is required. All accept a string path to a file

Variable	Data format
observed_spectrum	ASCII 3/4-column data with format: Wavelength, depth, error, widths
observed_lightcurve	Lightcurve pickle data
iraclis_spectrum	Iraclis output pickle data
taurex_spectrum	TauREX HDF5 output or self See *taurexspectrum*

5.11.2 Example

An example of loading an ascii data-set:

```
[Observation]
observed_
↪spectrum␣
↪= /path/
↪to/data.dat
```

TauREx Spectrum

The taurex_spectrum has two different modes. The first mode is specifing a filename path of a a TauREx3 HDF5 output. This output must have been run with some form of instrument function (see *[Instrument]*),

for it to be useable as an observation. Another is to set `taurex_spectrum = self`, this will set the current forward model + instrument function as the observation. This type observation is valid of the fitting procedure making it possible to do *self-retrievals*.

5.12 [Binning]

This section deals with the resampling of the forward model.

Binning allows you to change how the forward is sampled. When only running in forward model mode, it affects the final binned spectrum stored in the output and the plotting. It has no effect on retrievals.

The type of binning defined is given by the `bin_type` variable:

The available `bin_type` are:

- **`native`**
  - Spectra is not resampled
  - Default when no *[Observation]* is given
- **`observed`**
  - Resample to observation grid
  - Default when *[Observation]* is given
- **`manual`**
  - Manually defined resample grid

5.12.1 Manual binning

`bin_type = manual`

When set to manual, you can then define the *start*, *end* and *number of points* of the grid using one of these keywords:

Variable	Description
`wavelength_grid`	Equally spaced grid in wavelength (um)
`wavenumber_grid`	Equally spaced grid in wavenumber (cm-1)
`log_wavelength_grid`	Equally log-spaced grid in wavelength (um)
`log_wavenumber_grid`	Equally log-spaced grid in wavenumber (cm-1)

An example, to define an equally spaced wavelength grid at 0.3-5 um:

```
[Binning]
bin_
↪type = manual
wavelength_
↪grid␣
↪= 0.3, 5, 300
```

Or define an equally log spaced wavenumber grid between 400-5000 cm-1:

```
[Binning]
bin_
↪type = manual
log_wavenumber_
↪grid = 400,
↪ 5000, 300
```

Alternativly you can instead define it based on the resolution with the format as *start*, *end*, *resolution*

Variable	Description
`wavelength_res`	Wavelength grid at resolution (um)

We can define a grid

with 1.1-1.7 um at R=50 resolution:

```
[Binning]
bin_
↪type = manual
wavelength_
↪res =␣
↪1.1, 1.7, 50
```

Finally there is an optional parameter `accurate`. When *False*, a simpler histogramming method is used to perform the resampling. When set to `True` a more accurate method is used that takes into account the occupancy of each native sample on the sampling grid.

5.13 [Instrument]

This section deals with passing the forward model through some form of noise model.

The instrument function in TauREx3 serves to generate a spectrum and noise from a forward model.

Including a noise model in the TauREx3 input makes the output file capable of being used as an observation in the retrieval. It is also capable of fitting itself (See *TauREx Spectrum*)

The instrument is defined by the `instrum`

- **`snr`**

  - Signal-to-noise ratio instrument
  - Class: `SNR`
- **`custom`**
  - User-type instrument. See *Custom Types*

5.13.1 SNR

`instrument = snr` A very basic instrument that generates noise based on the forward model spectrum and signal-to-noise ratio value. Uses the native spectrum as the grid, unless a *Manual binning* is defined in which case that is used as the grid.

Keywords

Variable	Type	Description
`SNR`	`float`	Signal-to-noise ratio
`num_observation`	`int`	Number of observations

5.14 [Fitting]

This header deals with controlling the fitting procedure.

The format for altering and controlling fitting parameters is of the form:

```
fit_
↪param:option␣
↪= value
```

Here `fit_param` is the name of the fitting parameter as is given under the *Fitting Parameters* headers in the user documentation.
This also includes any custom fitting parameters provided by a users custom class (See: *Custom Types*) Only parameters that exist within the forward model can be set/altered. Trying to set any other parameter will yield an error.

`option` defines a set of control key words that alter what the fitting parameter does. For example, we can enable fitting of the planet radius using the `fit` option:

```
[Fitting]
planet_radius:
↪fit = True
```

5.14.1 New-style priors

New in version 3.1.

The `prior` option allows you define a prior function for a particular fitting parameter. This replaces the older method by allowing for more control over what type of function to use. They are expandable with new ones implemented through plugins or custom code.

Its syntax is very no similar to creating an object in python, for example to define a uniform prior of bounds 0.8–5.0 Jupiter masses we can do:

```
[Fitting]
planet_radius:fit = True
planet_radius:prior = "Uniform(bounds=(0.8, 5.0))"
```

It is **important** that the prior definition is surrounded by quotation marks. The prior definitions can contain multiple and distinct arguments, and have seperate `Log` forms as well with arguments in log-space:

```
[Fitting]
H2O:fit = True
H2O:prior = "LogUniform(bounds=(-12, -2))"
```

Often these log-forms have extra linear (`lin`) arguments where they are defined in linear space instead, for example, the prior space:

```
[Fitting]
H2O:fit = True
H2O:prior = "LogUniform(lin_bounds=(1e-12, 1e-2))"
```

(continues on next page)

(continued from previous page)

is equivalent to the previous example. The second included prior is the `Gaussian` prior which has mean and standard deviation arguments:

```
planet_
↪radius:
↪prior␣
↪=␣
↪
↪"Gaussian(mean=1.
↪0,std=0.3)"
```

as well as log versions:

```
H2O:
↪prior␣
↪=␣
↪
↪"LogGaussian(mean=-
↪4,std=2)"
```

The mean can be defined in linear space with the `lin_mean` argument:

```
H2O:
↪prior␣
↪=␣
↪
↪"LogGaussian(lin_
↪mean=1e-
↪4,std=2)"
```

5.14.2 Discovery

Refer to the documentation or plugin documentation to find out what fitting parameters are available. You can pass your input file with the `--fitparam` option to list available parameters:

```
> taurex -i␣
↪myinput.par␣
↪--fitparam
```

With the fitting paramaters listed under `Available Retrieval Parameters`:

```
--------
↪-------------
↪-------------
↪-------------
---
↪---Available␣
↪Retrieval␣
↪Parameters-
↪----------
--------
↪-------------
↪-------------
↪-------------


| Param Name  ␣
↪      | Short␣
↪Desc      ␣
↪                ␣
↪                ␣
↪                |

| planet_mass  ␣
↪     | Planet␣
↪mass in␣
↪Jupiter mass␣
↪                ␣
↪                |
|-------------------------------------------------------------------------
| planet_
↪radius␣
↪    | Planet␣
↪radius␣
↪in Jupiter␣
↪radii␣
↪                ␣
↪                |
|-------------------------------------------------------------------------
| planet_
↪distance   |␣
↪Planet semi␣
↪major axis␣
↪from parent␣
↪star (AU)␣
↪          |
|-------------------------------------------------------------------------
| planet_
↪sma        |
↪Planet semi␣
↪major axis␣
↪from parent␣
↪star (AU)␣
↪(ALIAS)  |
```

(continues on next page)

(continued from previous page)

```
atm_
↪min_pressure␣
↪| Minimum␣
↪pressure of␣
↪atmosphere␣
↪(top layer)␣
↪in Pascal |
atm_
↪max_pressure␣
↪| Maximum␣
↪pressure of␣
↪atmosphere␣
↪(surface)␣
↪in Pascal   |
T  ␣
↪              ␣
↪| Isothermal␣
↪temperature␣
↪in Kelvin␣
↪        ␣
↪              |
H2O              ␣
↪         | H2O␣
↪constant mix␣
↪ratio (VMR)␣
↪             ␣
↪              |
CH4              ␣
↪         | CH4␣
↪constant mix␣
↪ratio (VMR)␣
↪             ␣
↪              |
He_H2␣
↪             |␣
↪He/H2 ratio␣
↪(volume)␣
↪      ␣
↪              ␣
↪              |
clouds_
↪pressure␣
↪ | Cloud␣
↪top pressure␣
↪in Pascal␣
↪             ␣
↪              |
```

(continues on next page)

(continued from previous page)

```
                                                                                --------
                                                                               ↪-------------
                                                                               ↪-------------
                                                                               ↪-------------
                                                                               ---
                                                                               ↪---Available␣
                                                                               ↪Computable␣
                                                                               ↪Parameters-
                                                                               ↪---------
                                                                               --------
                                                                               ↪-------------
                                                                               ↪-------------
                                                                               ↪-------------


                                                                               | Param Name␣
                                                                               ↪   | Short␣
                                                                               ↪Desc    ␣
                                                                               ↪              ␣
                                                                               ↪              |

                                                                               | logg       ␣
                                                                               ↪    | Surface␣
                                                                               ↪gravity (m2/
                                                                               ↪s) in log10␣
                                                                               ↪         |
                                                                               |-------------------------------------------------------------------------------
                                                                               | avg_T      ␣
                                                                               ↪    | Average␣
                                                                               ↪temperature␣
                                                                               ↪across␣
                                                                               ↪all layers   |
                                                                               |-------------------------------------------------------------------------------
                                                                               | mu
                                                                               ↪       ␣
                                                                               ↪       | Mean␣
                                                                               ↪molecular␣
                                                                               ↪weight␣
                                                                               ↪at surface␣
                                                                               ↪(amu)  |
                                                                               |-------------------------------------------------------------------------------
                                                                               | C_O_ratio    ␣
                                                                               ↪ | C/O ratio␣
                                                                               ↪(volume)␣
                                                                               ↪        ␣
                                                                               ↪              |
                                                                               |-------------------------------------------------------------------------------
                                                                               | He_H_ratio   ␣
                                                                               ↪| He/H ratio␣
                                                                               ↪(volume)␣
                                                                               ↪       ␣
                                                                               ↪              |
```

5.14.3 Old-Style priors

Warning: It is recommended that the new style priors are used. These are only included for compatability and will be removed in the next major version of TauREx

We can set the prior boundaries between 1.0 - 5.0 Jupiter masses using the `bounds` option:

```
[Fitting]
planet_radius:
↪fit = True
planet_radius:
↪bounds␣
↪= 1.0, 5.0
```

And fit it in log space using the `mode` option:

```
[Fitting]
planet_radius:
↪fit = True
planet_radius:
↪bounds␣
↪= 1.0, 5.0
planet_radius:
↪mode = log
```

Caution: `bounds` *must* be given in linear space. Even if fitting in log space. TauREx3 will automatically convert these bounds to the correct fitting space.

If we have a constant H2O chemistry in the atmosphere we can fit it in linear

space instead of the default log:

```
[Fitting]
planet_radius:fit = True
planet_radius:bounds = 1.0, 5.0
planet_radius:mode = log
H2O:fit = True
H2O:mode = linear
H2O:bounds = 1e-12, 1e-1
```

5.14.4 Deperecated Options table

A summary all valid `option` is given here:

Option	Description	Values
`fit`	Enable or disable fitting	`True` or `False`
`bounds`	Prior boundaries in linear space	*min*, *max*
`factor`	Scaled boundaries in linear space	*sclmin*, *sclmax*
`mode`	Fitting space	`log` or `linear`

5.15 [Derive]

New in version 3.1.

This section deals with post-processed values from a retrieval.

The format for enabling post-processed values are:

```
derived_param:compute = value
```

Only `compute` is available as an option. Setting to

`True` will ask TauREx to generate posteriors at the end of a retrieval for the parameter using the sample points.

By default, the chemistry mean molecular mass (μ) at the surface is computed. We can disable this and instead compute the $log(g)$ of the planet surface and average temperature like so:

```
[Derive]
mu:compute␣
↪= False
logg:compute␣
↪= True
avg_T:compute␣
↪= True
```

Refer to the documentation or plugin documentation to find out what derived parameters are available. You can pass your input file with the `--fitparam` option to list available parameters:

```
> taurex -i␣
↪myinput.par␣
↪--fitparam
```

With the derived paramaters listed under `Available Computable Parameters`:

```
--------
↪-------------
↪-------------
↪-------------
---
↪---Available␣
↪Retrieval␣
↪Parameters-
```

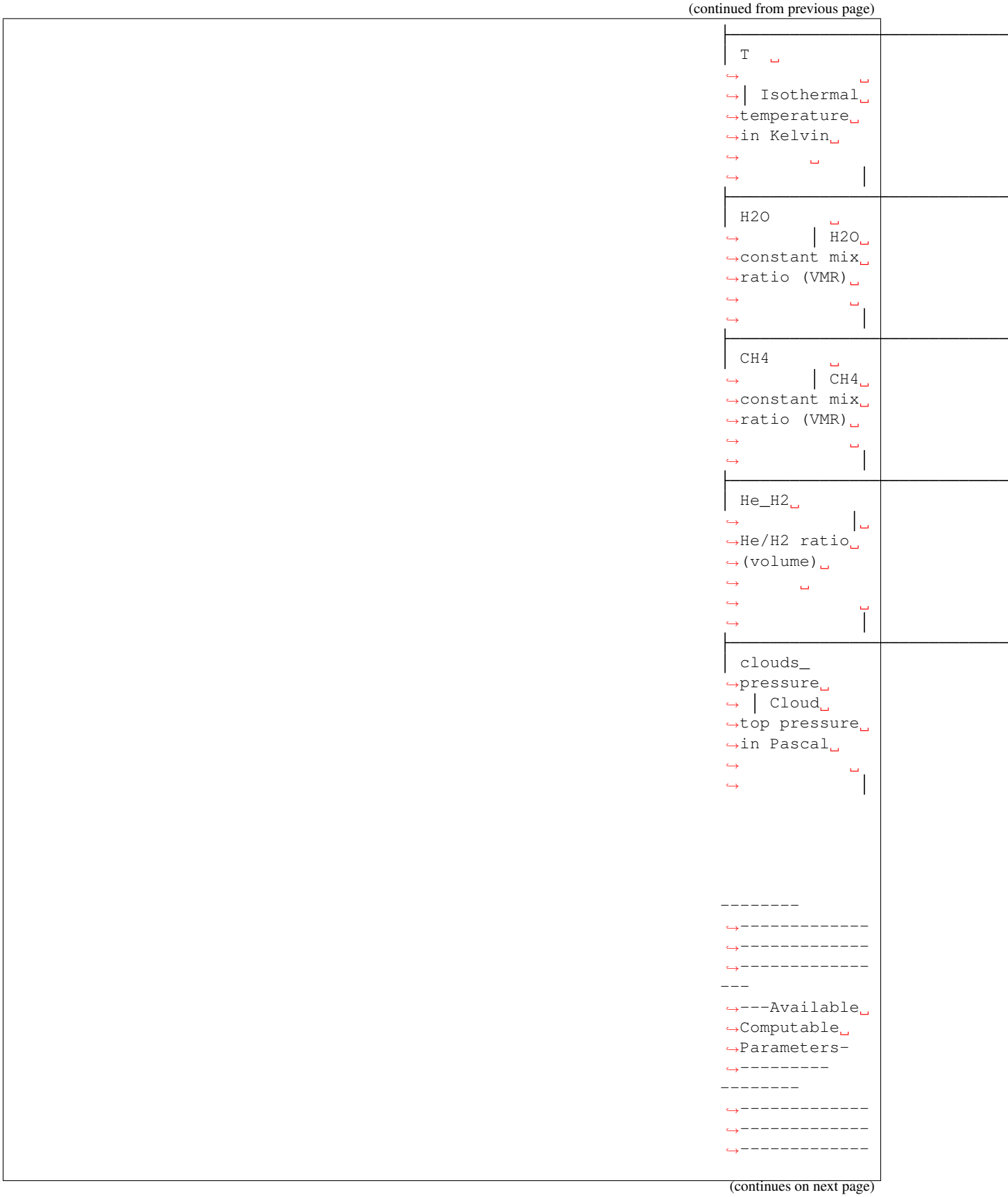
(continues on next page)

(continued from previous page)

```
--------
↪-------------
↪-------------
↪-------------


| Param Name  ␣
↪      | Short␣
↪Desc      ␣
↪                ␣
↪                ␣
↪                |

| planet_mass  ␣
↪     | Planet␣
↪mass in␣
↪Jupiter mass␣
↪                ␣
↪                |
|-------------------------------------------------------------------------
| planet_
↪radius␣
↪    | Planet␣
↪radius␣
↪in Jupiter␣
↪radii␣
↪                ␣
↪                |
|-------------------------------------------------------------------------
| planet_
↪distance   |␣
↪Planet semi␣
↪major axis␣
↪from parent␣
↪star (AU)␣
↪          |
|-------------------------------------------------------------------------
| planet_
↪sma        |␣
↪Planet semi␣
↪major axis␣
↪from parent␣
↪star (AU)␣
↪(ALIAS)  |
|-------------------------------------------------------------------------
| atm_
↪min_pressure␣
↪| Minimum␣
↪pressure of␣
↪atmosphere␣
↪(top layer)␣
↪in Pascal  |
|-------------------------------------------------------------------------
| atm_
↪max_pressure␣
↪| Maximum␣
↪pressure of␣
↪atmosphere␣
↪(surface)␣
↪in Pascal  |
```

(continues on next page)

(continued from previous page)



(continues on next page)

(continued from previous page)

```
| Param Name␣
↪  | Short␣
↪Desc  ␣
↪              ␣
↪                 |

| logg      ␣
↪   | Surface␣
↪gravity (m2/
↪s) in log10␣
↪       |
├──────────────────────────────────────────────────
| avg_T     ␣
↪   | Average␣
↪temperature␣
↪across␣
↪all layers  |
├──────────────────────────────────────────────────
| mu
↪       ␣
↪      | Mean␣
↪molecular␣
↪weight␣
↪at surface␣
↪(amu) |
├──────────────────────────────────────────────────
| C_O_ratio  ␣
↪ | C/O ratio␣
↪(volume)␣
↪       ␣
↪             |
├──────────────────────────────────────────────────
| He_H_ratio  ␣
↪| He/H ratio␣
↪(volume)␣
↪      ␣
↪             |
```

5.16 Mixins

New in version 3.1.

Mixins are lighter components with the sole purpose of giving *all* atmospheric components new abilities and features. For the coding inclined you can see the article here.

5.16.1 makefree

Works under: [Chemistry]

Adds new molecules or forces specific molecules in a chemical scheme to become fittable. Molecules will behave like a *Gas* and will add them as fitting parameters. Molecules can be defined similarly to free. For example if we load a chemistry from file, normally we cannot retrieve any molecule. If add the makefree mixin we can force specific molecules and add in new molecules into the scheme:

```
[Chemistry]
chemistry_type = makefree+file
filename = "mychemistryprofile.dat"
gases = H2O, CH4, CO, CO2

    [CH4]
    gas_type = constant

    [TiO]
    gas_type = constant

[Fitting]
CH4:fit = True
TiO:fit = True
```

Here, CH4 will has become fittable and we injected TiO into the scheme. What happens is

that each time the chemistry will run it will first run the base scheme and then modify or inject the molecule into the profile. Afterwhich the mixing profiles are then renormalized to unity. This can also work for equlibrium schemes, for example using ACE:

```
[Chemistry]
chemistry_type = makefree+ace
metallicity = 1.0

    [CH4]
    gas_type = constant

    [TiO]
    gas_type = constant

[Fitting]
CH4:fit = True
TiO:fit = True
metallicity:fit = True
```

Only the `free` chemical scheme does not work as it is redundant.

5.17 [Optimizer]

This section deals with the type of optimizer used.

TauREx3 includes a few samplers to perform retrievals. These can be set using the `optimizer` keyword:

- **nestle**

  - Nestle sampler
  - Class *NestleOptimizer*

- **multinest**
  - Use the MultiNest sampler
  - Class *MultiNestOptimizer*

- **polychord**
  - PolyChord Optimizer
  - Class PolyChordOptimizer

- **dypolychord**
  - dyPolyChord optimizer
  - Class dyPolyChordOptimizer

- **custom**
  - User-provided star model. See *Custom Types*

5.18 Plotting

Along with the *–plot* argument for taurex, there is also an extra program specifically for plotting to PDF from TauREx 3 HDF5 outputs. It is accessed like this:

```
taurex-plot
```

A summary of the arguments is given here:

Argument	Alternate name	Input	Description
-h	–help		show this help message and exit
-i	–input	INPUT_FILE	TauREx 3 HDF5 output file
-o	–output-dir	OUTPUT_DIR	Directory to store plots
-T	–title	TITLE	Title of plots (optional)
-p	–prefix	PREFIX	Output filename prefix (optional)
-m	–color-map	CMAP	Matplotlib colormap (optional)
-R	–resolution	RESOLUTION	Resample spectra at resolution
-P	–plot-posteriors		Plot posteriors
-x	–plot-xprofile		Plot molecular profiles
-t	–plot-tpprofile		Plot Temperature profiles
-d	–plot-tau		Plot optical depth contribution
-s	–plot-spectrum		Plot spectrum
-c	–plot-contrib		Plot contrib
-C	–full-contrib		Plot detailed contribs
-a	–all		Plot everythiong

CHAPTER

SIX

PLUGINS

New in version 3.1.

Inspired by Flask extensions, plugins are extra packages that add new functionality to TauREx. They allow anyone to improve and expand TauRExs capabilities without modifying the main codebase. For example, new forward models, opacity formats, chemistry and optimizers.

6.1 Finding Plugins

TauREx plugins usually are named as 'taurex_foo' or 'taurex-bar'. The Plugin Catalogue contains a list of plugins developed by us in the *Plugins Catalogue* . You can also search PyPI for packages with keywords taurex.

6.2 Using Plugins

Consult each plugins documentation for installation and usage. Generally TauREx searches for entry points in `taurex.plugins` and adds each component into the correct point in the pipeline

Lets take chemistry for example. Assuming a fresh install, we can see what is available to use in TauREx 3 by writing in the command prompt:

```
taurex --keywords chemistry
```

We get the output:

```
| chemistry_type   | Class           | Source   |
| file / fromfile  | ChemistryFile   | taurex   |
| taurex / free    | TaurexChemistry | taurex   |
```

We only have chemistry from a `file` and `free` chemistry. Supposing we wish to make use of FastChem. In the previous version we could easily load in an output from FastChem but what if we wanted to perform retrievals on the chemistry? We would need to write a wrapper of somekind that loads the C++ library into python before blah blah blah. A considerable amount of effort and likely someone else has solved the problem beforehand. This is what plugins solve!

With 3.1, we can now install the full FastChem chemistry code into TauREx3 with a single command:

```
pip install taurex_fastchem
```

Easy!

Now if we check the available chemistries we see:

```
| chemistry_type    | Class            | Source    |
| file / fromfile   | ChemistryFile    | taurex    |
| fastchem          | FastChem         | fastchem  |
| taurex / free     | TaurexChemistry  | taurex    |
```

We now have FastChem available!!!

Tip: It must be stressed that *downloading and installing FastChem is not necessary*, the plugin includes the precompiled library in the package.

Now we can use FastChem in the input file with retrievals:

```
[Chemistry]
chemistry_type = fastchem
selected_elements = H, He, C, N, O
metallicity = 2

[Fitting]
C_O_ratio:fit = True
C_O_ratio:priors = "LogUniform(-1,2)"
```

6.3 Building Plugins

While PyPI contains a growing list of TauREx plugins, you may not find a plugin that matches your needs. In this case you can try building your own! Read *Plugin Development* to learn how to develop your own and extend TauREx!

CHAPTER

SEVEN

LIBRARY

This section of the documentation deals with using the taurex library to construct your own scripts!

7.1 TauREx 3.0

7.1.1 Setup

Lets setup the notebook. If the plots arent interactive then run this part again

[56]:
```
import matplotlib.pyplot as plt
%matplotlib notebook
from ipywidgets import *
import numpy as np
import sys
```

And lets disable logging

[57]:
```
import taurex.log
taurex.log.disableLogging()
```

7.1.2 Loading cross-sections

We need to point TauREx3 to our cross-sections. This is handled by the caching classes. Once a cross-section is loaded it does not need to be loaded again. First lets import the classes:

[58]:
```
from taurex.cache import OpacityCache,CIACache
```

Now lets point the xsection and cia cachers to our files:

[59]:
```
OpacityCache().clear_cache()
OpacityCache().set_opacity_path("/path/to/xsec")
CIACache().set_cia_path("/path/to/cia")
```

TauREx3 is now ready to use them! For fun lets, try grabbing the H2O cross-section and plotting it. First tell the OpacityCache function to grab it.

[60]:
```
h2o_xsec = OpacityCache()['H2O']
```

Now we can compute the cross-section for any pressure and temperature! Lets try 2000K and 10 Pa.

[61]:
```
h2o_xsec.opacity(2000, 10)
```

[61]:
```
array([8.88161244e-26, 2.87333415e-26, 1.60882258e-26, ...,
       1.56910297e-33, 1.28513545e-33, 1.95851230e-33])
```

But why stop there? We can plot the temperature and pressure *interactively*

[62]:
```
xsec_fig = plt.figure()
xsec_ax = xsec_fig.add_subplot(1,1,1)



xsec, = xsec_ax.plot(10000/h2o_xsec.wavenumberGrid,h2o_xsec.opacity(800,1e0))

def update_cross(temperature=1500.0,pressure=6.7):

    xsec.set_ydata(h2o_xsec.opacity(temperature,10**pressure))
    xsec_ax.relim();
    xsec_ax.autoscale_view()
    xsec_fig.canvas.draw()

interact(update_cross,temperature=(800.0,2000.0,100),pressure=(-1.0,10.0,1));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
interactive(children=(FloatSlider(value=1500.0, description='temperature', max=2000.0,
↪ min=800.0, step=100.0),...
```

7.1.3 Profiles

Now we need to setup our forward model. Lets create a temperature profile, we will use the Guillot profile but other brands are available:

[63]:
```
from taurex.temperature import Guillot2010
guillot = Guillot2010(T_irr=1200.0)
```

Now lets do the same for our planet:

[64]:
```
from taurex.planet import Planet
planet = Planet(planet_radius=1.0,planet_mass=1.0)
```

and the planets star:

[65]:
```
from taurex.stellar import BlackbodyStar

star = BlackbodyStar(temperature=5700.0,radius=1.0)
```

Now we need to define a chemistry profile, first lets setup the chemical model, we're going for the free-type model so we'll use *TaurexChemistry* which allows us to freely add any molecule:

[66]:
```
from taurex.chemistry import TaurexChemistry
chemistry = TaurexChemistry(fill_gases=['H2','He'],ratio=0.172)
```

Adding molecules

Now we need to add some molecules. This is accomplished by the *addGas* function. We can create varius types of gas profiles for each molecule and add them in. Lets try the constant profile for H2O:

[67]:
```
from taurex.chemistry import ConstantGas

h2o = ConstantGas('H2O',mix_ratio=1.2e-4)
chemistry.addGas(h2o)
```

We can also create the gas on the spot as well:

[68]:
```
chemistry.addGas(ConstantGas('N2',mix_ratio=3.00739e-9))
```

And we're done for profiles! Like the cross-sections you can use them in isolation for your own evil deeds. Some require initialization with other profiles like pressure and altitude, you can find how to use them in the API documentation. An easy example are stars. Maybe you want to use the **PHOENIX** star library but don't want to bother coding the function to load and interpolate. Well you can just have *Taurex3* handle it for you!

[69]:
```
from taurex.stellar import PhoenixStar
##
anotherstar = PhoenixStar(phoenix_path='/path/to/phoenix/BT-Settl_M-0.0a+0.0',
↪temperature=5200.0)
star_fig = plt.figure()
star_ax = star_fig.add_subplot(1,1,1)

star_wngrid = np.linspace(0,100000,1000)

anotherstar.initialize(star_wngrid)

pstar, = star_ax.plot(anotherstar.spectralEmissionDensity)

def update_cross(temperature=5200.0):
    anotherstar.temperature=temperature
    anotherstar.initialize(star_wngrid)

    pstar.set_ydata(anotherstar.spectralEmissionDensity)
    star_ax.relim();
    star_ax.autoscale_view()
    star_fig.canvas.draw()

interact(update_cross,temperature=(800.0,8000.0,400));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
interactive(children=(FloatSlider(value=5200.0, description='temperature', max=8000.0,
↪ min=800.0, step=400.0),...
```

After that quick detour lets carry on with our goal.

7.1.4 Building the model

Now we can build our transmission model! Lets first create our transmission model and add our profiles to them:

[70]:
```
from taurex.model import TransmissionModel
tm = TransmissionModel(planet=planet,
                       temperature_profile=guillot,
                       chemistry=chemistry,
                       star=star,
                        atm_min_pressure=1e-0,
                       atm_max_pressure=1e6,
                       nlayers=30)
```

At this point our atmosphere has profiles but no physics! We can add this by including some contributions. Lets add in Absorption:

[71]:
```
from taurex.contributions import AbsorptionContribution
tm.add_contribution(AbsorptionContribution())
```

And some CIA for good measure:

[72]:
```
from taurex.contributions import CIAContribution
tm.add_contribution(CIAContribution(cia_pairs=['H2-H2','H2-He']))
```

And some rayleigh

[73]:
```
from taurex.contributions import RayleighContribution
tm.add_contribution(RayleighContribution())
```

Finally, putting it all together we **build** it to setup all the profiles

[74]:
```
tm.build()
```

Thats it! Our transmission model is complete! We can now run it:

[75]:
```
res = tm.model()
res
```

[75]:
```
(array([  666.61240713,   666.67906837,   666.74573628, ...,
        33326.66766653, 33330.0003333 , 33333.33333333]),
 array([0.01044906, 0.0104756 , 0.01047304, ..., 0.0104743 , 0.01047432,
        0.01047435]),
 array([[0.00000000e+000, 0.00000000e+000, 0.00000000e+000, ...,
         9.04389941e-006, 9.04390199e-006, 9.04377125e-006],
        [0.00000000e+000, 0.00000000e+000, 0.00000000e+000, ...,
         3.53049470e-117, 3.14645981e-117, 2.80401515e-117],
        [0.00000000e+000, 0.00000000e+000, 0.00000000e+000, ...,
         6.96586378e-074, 6.47671633e-074, 6.02165497e-074],
        ...,
        [9.99916988e-001, 9.99834482e-001, 9.99974411e-001, ...,
         9.98343344e-001, 9.98342620e-001, 9.98341897e-001],
        [9.99954235e-001, 9.99907945e-001, 9.99986665e-001, ...,
         9.99079595e-001, 9.99079193e-001, 9.99078791e-001],
        [9.99980310e-001, 9.99960342e-001, 9.99994409e-001, ...,
         9.99602467e-001, 9.99602293e-001, 9.99602120e-001]]),
 None)
```

Nice! The output has four components:

- The wavenumber grid
- The *native* flux

- The optical depth
- Any extra information

Now lets plot it! Lets see the chemistry of the atmosphere:

[76]:
```
plt.figure()

for x,gasname in enumerate(tm.chemistry.activeGases):

    plt.plot(tm.chemistry.activeGasMixProfile[x],tm.pressureProfile/1e5,label=gasname)
for x,gasname in enumerate(tm.chemistry.inactiveGases):

    plt.plot(tm.chemistry.inactiveGasMixProfile[x],tm.pressureProfile/1e5,
↪label=gasname)
plt.gca().invert_yaxis()
plt.yscale("log")
plt.xscale("log")
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Not interesting, but our chemistry model is pretty simple!

And now lets plot the flux

[77]:
```
native_grid, rprs, tau, _ = res

full_fig = plt.figure()
plt.plot(np.log10(10000/native_grid),rprs)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Coool! But lets try binning. We will use a simple but fast binner *SimpleBinner*.

[78]:
```
from taurex.binning import FluxBinner,SimpleBinner
binned_fig = plt.figure()


#Make a logarithmic grid
wngrid = np.sort(10000/np.logspace(-0.4,1.1,1000))
bn = SimpleBinner(wngrid=wngrid)

bin_wn, bin_rprs,_,_  = bn.bin_model(tm.model(wngrid=wngrid))

plt.plot(10000/bin_wn,bin_rprs)
plt.xscale('log')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Cool but the nice thing about TauREx is that you can alter any of the parameters and it will respond to it!

If any of the profiles are altered then model will respond to it. This means that parameters such as temperature and mix ratios and even contributions can be changed on the fly! Try this example out

[79]:
```
wngrid = np.sort(10000/np.logspace(-0.4,1.1,1000))
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

model, = ax.plot(np.log(10000/wngrid),bn.bin_model(tm.model(wngrid))[1])

def update_model(temperature=1500.0,h2o_mix=-4):
    guillot.equilTemperature = temperature
    tm['H2O'] = 10**h2o_mix
    model.set_ydata(bn.bin_model(tm.model(wngrid))[1])
    ax.relim();
    ax.autoscale_view()
    fig.canvas.draw()

interact(update_model,temperature=(800.0,2000.0,100),h2o_mix=(-7.0,-2.0,1));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
interactive(children=(FloatSlider(value=1500.0, description='temperature', max=2000.0,
↪ min=800.0, step=100.0),...
```

You can do some crazy things! If you reuse the same profiles on other models. You essentially couple them! That means you can have multiple model that all alter at the same time! We can see the equivalant Emission and Direct image spectrum like so:

[80]:
```
from taurex.model import EmissionModel, DirectImageModel
em = EmissionModel(planet=planet,
                        temperature_profile=guillot,
                        chemistry=chemistry,
                        star=star,
                         atm_min_pressure=1e-0,
                        atm_max_pressure=1e6,
                        nlayers=30)
di = DirectImageModel(planet=planet,
                        temperature_profile=guillot,
                        chemistry=chemistry,
                        star=star,
                         atm_min_pressure=1e-0,
                        atm_max_pressure=1e6,
                        nlayers=30)

em.add_contribution(AbsorptionContribution())
em.add_contribution(CIAContribution(cia_pairs=['H2-H2','H2-He']))
em.add_contribution(RayleighContribution())

di.add_contribution(AbsorptionContribution())
di.add_contribution(CIAContribution(cia_pairs=['H2-H2','H2-He']))
di.add_contribution(RayleighContribution())

em.build()
di.build()
```

[81]:
```
wngrid = np.sort(10000/np.logspace(-0.4,1.1,1000))

all_fig = plt.figure(figsize=(9,4))
```

(continues on next page)

(continued from previous page)

```
tm_ax = all_fig.add_subplot(1,3,1)
em_ax = all_fig.add_subplot(1,3,2)
di_ax = all_fig.add_subplot(1,3,3)
model_tm, = tm_ax.plot(10000/wngrid,bn.bin_model(tm.model(wngrid))[1])
model_em, = em_ax.plot(10000/wngrid,bn.bin_model(em.model(wngrid))[1])
model_di, = di_ax.plot(10000/wngrid,bn.bin_model(di.model(wngrid))[1])
tm_ax.set_xscale('log')
em_ax.set_xscale('log')
di_ax.set_xscale('log')
tm_ax.set_title('Transmission')
em_ax.set_title('Emission')
di_ax.set_title('Direct Image')
tm_ax.set_xlabel('Wavelength (um)')
em_ax.set_xlabel('Wavelength (um)')
di_ax.set_xlabel('Wavelength (um)')

def update_model(temperature=1500.0,h2o_mix=-4):
    guillot.equilTemperature = temperature
    tm['H2O'] = 10**h2o_mix
    model_tm.set_ydata(bn.bin_model(tm.model(wngrid))[1])
    model_em.set_ydata(bn.bin_model(em.model(wngrid))[1])
    model_di.set_ydata(bn.bin_model(di.model(wngrid))[1])
    tm_ax.relim();
    tm_ax.autoscale_view()

    em_ax.relim();
    em_ax.autoscale_view()

    di_ax.relim();
    di_ax.autoscale_view()

    fig.canvas.draw()

interact(update_model,temperature=(800.0,2000.0,100),h2o_mix=(-7.0,-2.0,1));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
interactive(children=(FloatSlider(value=1500.0, description='temperature', max=2000.0,
↪ min=800.0, step=100.0),...
```

7.1.5 Retreivals

To see what parameters available for retrievals we can list them like so:

[82]:
```
list(tm.fittingParameters.keys())
```

[82]:
```
['planet_mass',
 'planet_radius',
 'planet_distance',
 'atm_min_pressure',
 'atm_max_pressure',
 'T_irr',
 'kappa_irr',
 'kappa_v1',
 'kappa_v2',
```

(continues on next page)

(continued from previous page)

```
 'alpha',
 'H2O',
 'N2',
 'He_H2']
```

The thing is, this is *dynamic*, TauREx 3 figures out what can be fit based on whats in it. Lets throw an isothermal profile instead:

[83]:
```
from taurex.temperature import Isothermal

isothermal = Isothermal(T=1500.0)

tm = TransmissionModel(planet=planet,
                       temperature_profile=isothermal,
                       chemistry=chemistry,
                       star=star,
                       atm_min_pressure=1e-0,
                       atm_max_pressure=1e6,
                       nlayers=30)
tm.add_contribution(AbsorptionContribution())
tm.add_contribution(CIAContribution(cia_pairs=['H2-H2','H2-He']))
tm.add_contribution(RayleighContribution())
tm.build()
```

[84]:
```
tm.model()
```

[84]:
```
(array([  666.61240713,   666.67906837,   666.74573628, ...,
        33326.66766653, 33330.0003333 , 33333.33333333]),
 array([0.01052613, 0.01063063, 0.01056525, ..., 0.01052609, 0.01052612,
        0.01052615]),
 array([[0.00000000e+000, 0.00000000e+000, 0.00000000e+000, ...,
         6.43132652e-006, 6.43132806e-006, 6.43124781e-006],
        [0.00000000e+000, 0.00000000e+000, 0.00000000e+000, ...,
         1.42570291e-110, 1.27917056e-110, 1.14762904e-110],
        [0.00000000e+000, 0.00000000e+000, 0.00000000e+000, ...,
         1.07604140e-069, 1.00473066e-069, 9.38108083e-070],
        ...,
        [9.99796601e-001, 9.95780957e-001, 9.98087394e-001, ...,
         9.98554699e-001, 9.98554068e-001, 9.98553436e-001],
        [9.99887426e-001, 9.97654304e-001, 9.98937920e-001, ...,
         9.99196758e-001, 9.99196407e-001, 9.99196056e-001],
        [9.99951499e-001, 9.98986426e-001, 9.99541444e-001, ...,
         9.99653022e-001, 9.99652870e-001, 9.99652719e-001]]),
 None)
```

[85]:
```
list(tm.fittingParameters.keys())
```

[85]:
```
['planet_mass',
 'planet_radius',
 'planet_distance',
 'atm_min_pressure',
 'atm_max_pressure',
 'T',
 'H2O',
 'N2',
 'He_H2']
```

Here we lost the Guillot parameters like *T_irr* but gained the Isothermal parameter *T* We can also access them directly

from the model using the brackets operator:

[86]:
```
tm['T']
```

[86]:
```
1500.0
```

And set them

[87]:
```
tm['H2O']=1.2e-4
```

Now we'll need an observation. lets use *ObservedSpectrum* to load a text one from **examples/test_data.dat**:

[88]:
```
from taurex.data.spectrum.observed import ObservedSpectrum
obs = ObservedSpectrum('path/to/test_data.dat')
```

Convieniently we have a way of binning our native spectrum down to the observation by calling its *create_binner* method:

[89]:
```
obin = obs.create_binner()
```

And we can now plot

[90]:
```
plt.figure()
plt.errorbar(obs.wavelengthGrid,obs.spectrum,obs.errorBar,label='Obs')
plt.plot(obs.wavelengthGrid,obin.bin_model(tm.model(obs.wavenumberGrid))[1],label='TM
↪')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Ew not good! Lets try a retrieval! We have many optimizers to choose so lets create one using the inbuilt optimizer based on nestle

[91]:
```
from taurex.optimizer.nestle import NestleOptimizer
opt = NestleOptimizer(num_live_points=50)
```

We need to tell it about our forward model and observation:

[92]:
```
opt.set_model(tm)
opt.set_observed(obs)
```

Now lets enable which parameters to fit and their prior boundaries:

[93]:
```
opt.enable_fit('planet_radius')
opt.enable_fit('T')
opt.set_boundary('T',[1000,2000])
opt.set_boundary('planet_radius',[0.8,2.1])
```

Now lets fit!!!!

[94]:
```
solution = opt.fit()
taurex.log.disableLogging()
```

```
it=   663 logz=1867.323389811400
```

```
taurex.Nestle - INFO - Sampling time 85.49567294120789 s
taurex.Nestle - INFO - Generating spectra and profiles
taurex.Nestle - INFO - Computing solution 0
taurex.TransmissionModel - INFO - Computing pressure profile
taurex.ChemistryModel - INFO - Initializing chemistry model
taurex.Absorption - INFO - Recomputing active gas H2O opacity
taurex.Absorption - INFO - Done
taurex.CIA - INFO - Computing CIA
taurex.CIA - INFO - Done
taurex.Rayleigh - INFO - Done
```

```
niter: 664
ncall: 1096
nsamples: 714
logz: 1867.763 +/-  0.478
h: 11.444
```

```
taurex.TransmissionModel - INFO - Computing pressure profile
taurex.ChemistryModel - INFO - Initializing chemistry model
taurex.Absorption - INFO - Recomputing active gas H2O opacity
taurex.Absorption - INFO - Done
taurex.CIA - INFO - Computing CIA
taurex.CIA - INFO - Done
taurex.Rayleigh - INFO - Done
taurex.TransmissionModel - INFO - Computing pressure profile
taurex.ChemistryModel - INFO - Initializing chemistry model
taurex.TransmissionModel - INFO - Modelling each contribution...
taurex.Absorption - INFO - Recomputing active gas H2O opacity
taurex.TransmissionModel - INFO -       Absorption---H2O contribtuion
taurex.CIA - INFO - Computing CIA
taurex.TransmissionModel - INFO -       CIA---H2-H2 contribtuion
taurex.TransmissionModel - INFO -       CIA---H2-He contribtuion
taurex.TransmissionModel - INFO -       Rayleigh---H2O contribtuion
taurex.TransmissionModel - INFO -       Rayleigh---N2 contribtuion
taurex.TransmissionModel - INFO -       Rayleigh---H2 contribtuion
taurex.TransmissionModel - INFO -       Rayleigh---He contribtuion
taurex.TransmissionModel - INFO - Computing pressure profile
taurex.ChemistryModel - INFO - Initializing chemistry model
taurex.Absorption - INFO - Recomputing active gas H2O opacity
taurex.Absorption - INFO - Done
taurex.CIA - INFO - Computing CIA
taurex.CIA - INFO - Done
taurex.Rayleigh - INFO - Done
taurex.Nestle - INFO - ------------Profile generation step------------------
taurex.Nestle - INFO - We are sampling 71 points for the profiles
taurex.Nestle - INFO - I will only iterate through partitioned 71 points (the rest is␣
↪in parallel)
taurex.Nestle - INFO - Done!
taurex.Nestle - INFO -
taurex.Nestle - INFO - ------------------------------------
taurex.Nestle - INFO - ------Final results------------------
taurex.Nestle - INFO - ------------------------------------
taurex.Nestle - INFO -
taurex.Nestle - INFO - Dimensionality of fit: 2
taurex.Nestle - INFO -
taurex.Nestle - INFO -
---Solution 0------
taurex.Nestle - INFO -
```

(continues on next page)

(continued from previous page)

```
Param                 MAP       Median
-------------  ----------  -----------
planet_radius     0.99969     0.999732
T              1443.26     1444.7
```

Lets loop and plot each solution!

[95]:
```
for solution,optimized_map,optimized_value,values in opt.get_solution():
    opt.update_model(optimized_map)
    plt.figure()
    plt.errorbar(obs.wavelengthGrid,obs.spectrum,obs.errorBar,label='Obs')
    plt.plot(obs.wavelengthGrid,obin.bin_model(tm.model(obs.wavenumberGrid))[1],label=
↪'TM')
    plt.legend()
    plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Nice!

- genindex

CHAPTER

EIGHT

DEVELOPERS GUIDE

8.1 Overview

TauREx 3 follows a simply philosophy:

> *I don't care how you do it, just give it to me!*

A relevant analogy: We don't really care *how* a temperature profile is computed, just as long as when we ask for one it gives it to us.

TauREx 3 follow heavily the OOP design to achieve this. Almost everything is split into building blocks that can be mixed and match and combined to form a full atmospheric model and retrieval. Each of these building blocks are a set of *interfaces* or *guarantees* that allow to each part to work together.

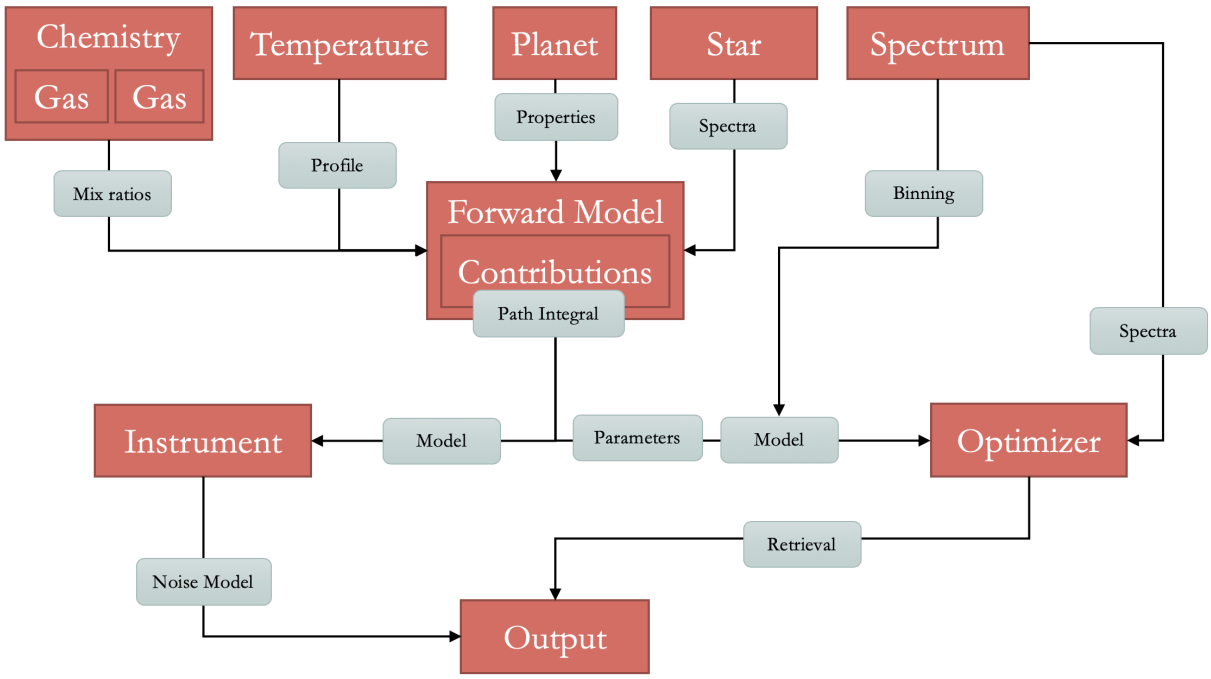


Fig. 1: Simple overview of how blocks connect and what they provide

These come in the form of *abstract* or *skeleton* classes that can be taken and filled out to form a new block in the TauREx 3 architecture.

8.2 Taurex 3 development guidelines

8.2.1 Overview

Here we describe the development guidelines for TauREx 3 and some advice for those wishing to contribute. Since TauREx 3 is open-source, all contributions are welcome!!!

Development on TauREx 3 should be focused on building and improving the framework. New components (i.e. chemistries, profiles etc.) are generally not built directly into the TauREx 3 codebase.

We recommend building new components as Plugins. You can refer to the *Plugin Development* guide.

8.2.2 Documentation

All standalone documentation should be written in plain text (`.rst`) files using reStructuredText for markup and formatting. All docstrings should follow the numpydoc format. New features must include both appropriate docstrings and any necessary standalone documentation

8.2.3 Unit-testing

Unittesting is important in preserving sanity and code integrity. For TauREx 3 we employ pytest. When bugfixing, ensure unittests pass. In the root directory do:

```
pytest tests/
```

To run all unit tests in TauREx3

When building new features, create new unittests and include them in the `test/` directory, any future collaborations from other developers are less likely to break your feature unexpectedly when they have something to easily test against.

Some rules:

- No *extra* files should be included. Instead have the unit test generate them on the spot.
- We recommended hypothesis for bug finding

Coding conventions

Code should follow the PEP8 standard. This can be facilitated with a linter such as flake8

Source control

Git is the source control environment used. In particular, we follow the git-flow branching model internally. In this model, there are two long-lived branches:

- `master`: used for official releases. **Contributors should not need to use it or care about it**
- `develop`: reflects the latest integrated changes for the next release. This is the one that should be used as the base for developing new features or fixing bugs.

For contributions we employ the Fork-and-Pull model:

1. A contributor first forks the TauREx3 repo
2. They then clone their forked branch

3. The contributor then commits and merges their changes into their forked `develop` branch
4. A Pull-Request is created against the official `develop` branch
5. Anyone interest can review and comment on the pull request, and suggest changes. The contributor can continue to commit more changes until it is approved
6. Once approved, the code is considered ready and the pull request is merged into the official develop

8.3 Basics

There are some common rules when developing new components for TauREx 3. These apply to the majority of components in the TauREx pipeline. The only major exception is the *Opacity* related classes that have a different system in place.

8.3.1 Automatic Input Arguments

In TauREx, when loading in a class, it will dynamically parse all `__init__` arguments and make them accessible in the input file. If you build a new temperature class:

```
from taurex.temperature import TemperatureProfile
import numpy as np

class MyNewTemperatureProfile(TemperatureProfile):

    def __init__(self, mykeyword=[1,2,3], another_keyword='A string'):
        super().__init__(name=self.__class__.__name__)
        print('A: ',mykeyword)
        print('B: ',another_keyword)

    @property
    def profile(self):
        T = np.random.rand(self.nlayers)*1000 + 1
        return T

    @classmethod
    def input_keywords(cls):
        return ['myprofile']
```

Then the keyword arguments `mykeyword` and `another_keyword` become arguments in the input file:

```
[Temperature]
profile_type = myprofile
my_keyword = 5,6,7,
another_keyword = "Another string"
```

Which when run will produce:

```
A: [5, 6, 7]
B: Another string
```

We recommend defining all `__init__` arguments as keywords if you intend for your components to be used through the input file. The input file only supports arguments that accept:

- scalars or strings
- **lists of scalars and/or strings**

- i.e `my_arg = 1, 3.14, hello-world!`

8.3.2 Input keywords

Most classes in TauREx include the class method `input_keywords`. This function returns a list of words used to identify the component in the input file. Under most headers in the input file there is a selection keyword (i.e *[Optimizer]* has `optimizer`, *[Chemistry]* has `chemistry_type` etc.) used to select the correct class for the job. This selection is made by searching for the values `input_keywords` from all components of that type until a match is found. So, for example, if we have a new sampler:

```
from taurex.optimizer import Optimizer
class MyOptimizer(Optimizer):
    #....
    @classmethod
    def input_keywords(cls):
        return ['myoptimizer', ]
```

We can select it in the input file as:

```
[Optimizer]
optimizer = myoptimizer
```

You can also alias the class by including multiple words:

```
from taurex.optimizer import Optimizer
class MyOptimizer(Optimizer):
#....
    @classmethod
    def input_keywords(cls):
        return ['myoptimizer', 'my-optimizer',
        'hello-optimizer']
```

We can select the class using one of the three values:

```
[Optimizer]
optimizer = myoptimizer # Valid
optimizer = my-optimizer # Also Valid
optimizer = hello-optimizer # Valid as well
```

Developers implementing this must follow a few rules:

- The values must be *lowercase* only
- *Commas* are not allowed
- They must be *unique*; if two components have the same values, then one may never be selected

---

Tip: This is only necessary if you intend to have your component usable from the input file. If you only indent for it to work when used in a python script, you can omit this.

8.3.3 Logging

Every component has access to *info() warning()*, *debug() error()* and *critical()* methods:

```
from taurex.chemistry import Chemistry
class MyChemistry(Chemistry):

    def do_things(self):
        self.info('I am info')
        self.warning('I am warning!!')
        self.error("I am error!!!")
```

Calling `do_things` will output:

```
taurex.MyChemistry - INFO - I am info
taurex.MyChemistry - WARNING - I am warning!!
taurex.MyChemistry - ERROR - In: do_things()/line:7 - I am error!!!
```

While you can use your own printing methods. We recommend using these built in methods for logging as:

- They can be automatically hidden during retrievals
- They will only output once under MPI
- They automatically include the class, function and line number for *debug()*, *error()* and *critical()*.

8.3.4 Bibliography

New in version 3.1.

It is important to recognise the works involved in each component during a TauREx run. TauREx includes a basic bibliography system that will collect and parse bibtex entries embedded in each component.

Embedding bibliographic information for most cases only requires defining the `BIBTEX_ENTRIES` class variable as a list of bibtex entries:

```
from taurex.temperature import TemperatureProfile
import numpy as np

class MyNewTemperatureProfile(TemperatureProfile):

    def __init__(self, mykeyword=[1,2,3], another_keyword='A string'):
        super().__init__(name=self.__class__.__name__)
        print('A: ',mykeyword)
        print('B: ',another_keyword)

    @property
    def profile(self):
        T = np.random.rand(self.nlayers)*1000 + 1
        return T

    @classmethod
    def input_keywords(cls):
        return ['myprofile']

    BIBTEX_ENTRIES = [
        """
        @article{myprof,
            url = {https://vixra.org/abs/1512.0013},
            year = 2015,
            month = {dec},
            volume = {1512},
```

(continues on next page)

(continued from previous page)

```
            number = {0013},
            author = {Ben S. Dover, Micheal T Hunt, Christopher S Peacock},
            title = {A New Addition to the Stellar Metamorphsis. the Merlin␣
↪Hypothesis},
            journal = {vixra},
        }
        """,
        """
        @misc{vale2014bayesian,
            title={Bayesian Prediction for The Winds of Winter},
            author={Richard Vale},
            year={2014},
            eprint={1409.5830},
            archivePrefix={arXiv},
            primaryClass={stat.AP}
        }
        """

    ]
```

Warning: If your BibTeX entry includes non-Unicode characters, then Python will refuse to run, or your plugin may not be able to load into the TauREx pipeline.

Running TauREx, on program end, we get:

```
A New Addition to the Stellar Metamorphsis. the Merlin Hypothesis
Ben S. Dover, Micheal T Hunt, Christopher S Peacock
vixra, 1512, dec, 2015

Bayesian Prediction for The Winds of Winter
Vale, Richard
arXiv, 1409.5830, 2014
```

Additionally, running `taurex` with `--bibtex mybib.bib` will export the citation as a `.bib` file:

```
@misc{cad6f055,
    author = "Al-Refaie, Ahmed F. and Changeat, Quentin and Waldmann, Ingo P. and␣
↪Tinetti, Giovanna",
    title = "TauREx III: A fast, dynamic and extendable framework for retrievals",
    year = "2019",
    eprint = "1912.07759",
    archivePrefix = "arXiv",
    primaryClass = "astro-ph.IM"
}

@article{6720c2d1,
    author = "Ben S. Dover, Micheal T Hunt, Christopher S Peacock",
    url = "https://vixra.org/abs/1512.0013",
    year = "2015",
    month = "dec",
    volume = "1512",
    number = "0013",
    title = "A New Addition to the Stellar Metamorphsis. the Merlin Hypothesis",
    journal = "vixra"
```

(continues on next page)

(continued from previous page)

```
}

@misc{f55ed081,
    author = "Vale, Richard",
    title = "Bayesian Prediction for The Winds of Winter",
    year = "2014",
    eprint = "1409.5830",
    archivePrefix = "arXiv",
    primaryClass = "stat.AP"
}
```

Bibliographies are additive as well; if we decided to build on top of this class we do not need to redefine the older bibliographic information as all parent bibliographic information is also inherited:

```
class AnotherProfile(MyNewTemperatureProfile):
# ...

    BIBTEX_ENTRIES = [
        """
        @misc{scott2015farewell,
            title={A Farewell to Falsifiability},
            author={Douglas Scott and Ali Frolop and Ali Narimani and Andrei Frolov},
            year={2015},
            eprint={1504.00108},
            archivePrefix={arXiv},
            primaryClass={astro-ph.CO}
            }
        ]
```

Will yield:

```
A Farewell to Falsifiability
Douglas Scott, Ali Frolop, Ali Narimani, Andrei Frolov
arXiv, 1504.00108, 2015

A New Addition to the Stellar Metamorphsis. the Merlin Hypothesis
Ben S. Dover, Micheal T Hunt, Christopher S Peacock
vixra, 1512, dec, 2015

Bayesian Prediction for The Winds of Winter
Vale, Richard
arXiv, 1409.5830, 2014
```

You can get citations from each object through the *citations()* method which will output a `list` of parsed bibtex entries:

```
>>> t = MyNewTemperatureProfile()
>>> t.citations()
[Entry('article',
fields=[
('url', 'https://vixra.org/abs/1512.0013'),
('year', '2015'),
('month', 'dec'),
('volume', '1512'),
('number', '0013'),
('title', 'A New Addi.....etc
```

A printable string can also be generated using the *nice_citation()* method:

```
>>> print(t.nice_citation())
A New Addition to the Stellar Metamorphsis. the Merlin Hypothesis
Ben S. Dover, Micheal T Hunt, Christopher S Peacock
vixra, 1512, dec, 2015

Bayesian Prediction for The Winds of Winter
Vale, Richard
arXiv, 1409.5830, 2014
```

If you're developing a *ForwardModel* then *citations()* should include its own BIBTEX_ENTRIES as well as every component in the model itself (i.e Temperature, Contributions etc.) we have a nice recipe to accomplish this:

```
def citations(self):

    all_citiations = [
        super().citations(),
        self.tp.citations(),
        self.chem.citations(),
        # Other components
        # ...etc...
    ]

    return unique_citiations_only(
    sum(all_citiations,[])
```

Here `self.tp` and `self.chem` are temperature and chemistry components used in our implementation of a forward model. *unique_citations_only()* will remove any repeat bibliography information and `sum(all_citiations,[])` combines all citation lists into a single list.

8.4 Retreival Parameters

8.4.1 Fitting

TauREx 3 employs dynamic discovery of retrieval parameters. When a new profile/chemistry etc is loaded into a forward model, they also advertise the parameters that can be retrieved. The forward model will collect and provide them to the optimizer which it then uses to perform the sampling.

Classes that inherit from *Fittable* are capable of having retrieval parameters. Classes that inherit from this include:

- *TemperatureProfile*
- *Chemistry*
- *Gas*
- *PressureProfile*
- *Star*
- Planet
- *ForwardModel*
- *Contribution*

There are two ways of defining, fitting parameters. The simpler decorator method and programmaticaly

8.4.2 Decorator form

The decorator *fitparam()* decorator acts and behaves almost identically to the `@property` python decorator.

8.4.3 Programmaticaly

The decorator form is useful for describing parameters that always exist in a model in the same form. There are cases where parameters may need to be generated based on some input. One example is the *NPoint* temperature profile. Depending on the number of temperature points input by the user the temperature profile will actually generate *new* fitting parameters for each point (i.e `T_point1`, `T_point2` etc) Another example are the *Gas* profiles. In this case, the fitting parameter names ar

8.5 Components

Warning: This is still under construction as I try to figure out how best to convey this.

Here we present the most basic form of each component. See basic features under *Basics* and retrieval features under retrievaldev

8.5.1 Temperature

The most basic temperature class has this form:

```
from taurex.temperature import Temperature
import numpy as np

class MyTemperature(Temperature):

    def __init__(self):
        super().__init__(self.__class__.__name__)

    def initialize_profile(self, planet=None, nlayers=100,
                           pressure_profile=None):
        self.nlayers = nlayers

        self.myprofile = np.ones(nlayers)*1000.0

    @property
    def profile(self):
        return self.myprofile
```

__init__()

Used to build the component and only called once. Must include `super()` call. Decorator fitting parameters are also collected here automatically. Use keyword arguments to setup the class and load any necessary files. You can also build new fitting parameters here as well.

`initialize_profile()`

Used to initialize and compute the temperature profile. It is run on each *`model()`* call

Arguments:

- `planet`: `Planet`
- `nlayers`: Number of Layers
- **`pressure_profile`: `nlayer` array of pressures.**
  - BOA to TOA
  - Units: Pa

`profile()`

Must be decorated with @property. Must return an array of same shape as `pressure_profile` with units K

8.5.2 Chemistry

We recommend using *`AutoChemistry`* as a base as it greatly simplifies implementation of active and inactive species.

```
from taurex.chemistry import AutoChemistry
import numpy as np

class MyChemistry(AutoChemistry):

    def __init__(self):
        super().__init__(self.__class__.__name__)

        # Perform setup here

        # Populate gases here
        self.mygases = ['H2', 'He', 'H2O', 'CH4', 'NO', 'H2S','TiO',]

        # Call when gases has been populated
        self.determine_active_inactive()

    def initialize_chemistry(self, nlayers=100, temperature_profile=None,
                             pressure_profile=None, altitude_profile=None):

        num_molecules = len(self.gases)


        # We will compute a random profile for each molecule
        self.mixprofile = np.random.rand(num_molecules, nlayers)

        # Make sure each layer sums to unity
        self.mixprofile/= np.sum(self.mixprofile,axis=0)

        # Compute mu profile
        self.compute_mu_profile(nlayers):

    @property
    def gases(self):
        return self.mygases
```

(continues on next page)

(continued from previous page)

```
    @property
    def mixProfile(self):
        return self.mixprofile
```

For chemistry whats important is the the method *determine_active_inactive()* must be called once `gases()` has been populated with the species.

__init__()

Used to build the component and only called once. Must include `super()` call. Decorator fitting parameters are also collected here automatically. Use keyword arguments to setup the class and load any necessary files. You can also build new fitting parameters here as well. We recommend determining your chemical species at this point.

initialize_chemistry()

Used to initialize and compute the chemical model. The μ profile should be computed as well. It is run on each *model()* call

Arguments:

- `nlayers`: Number of Layers
- **temperature_profile: nlayer array of temperature.**
  - BOA to TOA
  - Units: K
- **pressure_profile: nlayer array of pressures.**
  - BOA to TOA
  - Units: Pa

gases()

Must be decorated with @property. Must return a list of species in the chemical model

mixProfile()

Return volume mixing ratios. Must be decorated with @property. Must return an array of shape (number of species, number of layers). The ordering of species must be 1:1 with *gases()*

8.6 Mixins

New in version 3.1.

Mixins are lighter components with the sole purpose of giving *all* atmospheric components new abilities and features. For the coding inclined you can see the article here.

8.6.1 Motivation

To understand this, lets take an viable scenario. Imagine you've come up with an amazing idea. What if all temperature profiles must be doubled to be physically valid? Incredible! So you begin your Nobel Prize winning work and begin defining new temperature profiles for each of the available ones in TauREx3. You create `isothermal_double`, `npoint_double`, `guillot_double` etc and release it to the amazement of the public. Someone comes along and develops a super new temperature profile, lets call it `supernewtemp`. Well now looks like you'll now have to go back and implement a `supernewtemp_double` but no matter, progress comes with sacrifice. Now your colleague suggests that adding 50K also improves the profile, so they implement `isothermal_50`, `npoint_50`, `guillot_50` and `supernewtemp_50`. Now some people say they want to double it and add 50 so someone must create `isothermal_double_50`, `npoint_double_50`, `guillot_double_50` and `supernewtemp_double_50` and other people want to add 50 and double so now we need to build `isothermal_50_double`, `npoint_50_double`, `guillot_50_double` and `supernewtemp_50_double` and oh no someone just created a brand new temperature profile and deeper into the endless abyss you go.

This is what mixins solve, if instead we develop a `doubler` mixin we can instead *add* it to our original profile using the + operator:

```
[Temperature]
profile_type = doubler+isothermal
T = 1000
```

And TauREx will build an isothermal profile that doubles itself for you. Neat We can do the same and build an `add50` mixin:

```
[Temperature]
profile_type = add50+isothermal
```

Now the beauty is that we can stack them together!! If we want to double then add 50 we can write:

```
[Temperature]
profile_type = add50+doubler+isothermal
```

Or add 50 then double:

```
[Temperature]
profile_type = doubler+add50+isothermal
```

The examples given are fairly simple, performing a summation. There are other useful features related to mixins, one mixin sets planet and stellar properties by name! Another splices UV data into stellar models.

8.6.2 Developing Mixins

Each TauREx component has a mixin equivalent, developing mixins means choosing what mixin type to inherit from. (i.e `Star` has *`StarMixin`*). Developing mixins work slightly differently to traditional components. Lets develop two mixins for the stellar models. One that adds noise and another the removes noise (redundant but shows the nature of them) to the spectral emission density. First, unless you know what you are doing, **do not create an __init__ method**. There is a dedicated initialization method we can use, `__init_mixin__`. Lets setup our mixins:

```
from taurex.mixin import StarMixin
import numpy as np
from scipy.signal import medfilt

class Noiser(StarMixin):
```

(continues on next page)

(continued from previous page)

```
    def __init_mixin__(self, noise_level=1.0):
        self.noise_level = noise_level

    @classmethod
    def input_keywords(cls):
        return ['noiser', ]

class Denoiser(StarMixin):

    def __init_mixin__(self, kernel_size=3):
        self.kernel_size = kernel_size

    @classmethod
    def input_keywords(cls):
        return ['denoiser', ]
```

Now `Noiser` will use `numpy.random.randn` to generate gaussian noise and add it to the spectral emission density. And `Denoiser` will use a median filter from `scipy.signal.medfilt` to clean up the spectral emission density.

Since its the spectral emission density we are modify we would need to run and then overwrite the original classes function. We can accomplish this by defining our own `spectralEmissionDensity()` and exploiting super():

```
class Noiser(StarMixin):

    ...

    @property
    def spectralEmissionDensity(self):
        previous_sed = super().spectralEmissionDensity

        new_sed = previous_sed + \
                np.random.randn(*previous_sed.shape)*self.noise_level

        return new_sed


class Denoiser(StarMixin):

    ...

    @property
    def spectralEmissionDensity(self):
        previous_sed = super().spectralEmissionDensity

        new_sed = medfilt(previous_sed, kernel_size=self.kernel_size)

        return new_sed
```

super() is key to mixins. It allows use to evaluate the method of the super class, or in other words. *The class that came before us*. Using this, we can get the original spectrum and then modify it and return it. It can also be chained as well. If we apply two mixins that modify this, then calling super will evaluate the previous mixin which, evaluates the original class. Nicely, this tangent leads us to the next point, how do we actually use the mixin? The *`enhance_class()`* function does exactly this! It takes our *base*, a list of mixins and arguments and generates a new instance of the class! Lets try it for the noiser and modify the black body star:

```
>>> from taurex.mixin import enhance_class
>>> from taurex.stellar import BlackbodyStar
>>> new_star = enhance_class(BlackbodyStar, [Noiser, ], temperature=5800,
                                                        radius=1.0,
                                                        noise_level=1e6)
>>> new_star
<taurex.mixin.core.Noiser+BlackbodyS at 0x7fdb93de4c70>
```

The class is neither a Noiser or Blackbody but a combination of both. What you might notice is arguments from both *BlackbodyStar* and `Noiser` passed in. Each argument is automatically passed to the correct class for you! Anyway lets plot it and see:

```
>>> import matplotlib.pyplot as plt
>>> wngrid = np.linspace(300,30000,10000)
>>> new_star.initialize(wngrid)
>>> plt.figure()
>>> plt.plot(10000/wngrid,new_star.spectralEmissionDensity)
>>> plt.xscale('log')
>>> plt.show()
```

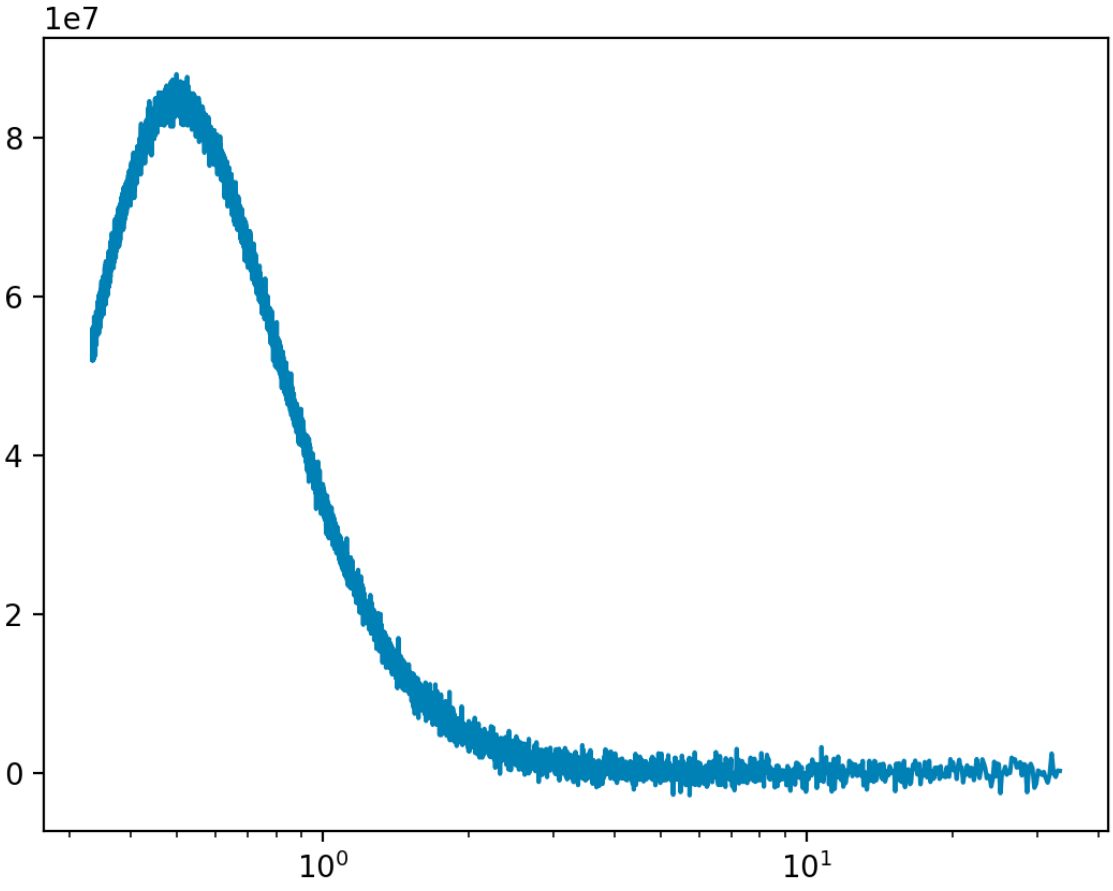


Fig. 2: Blackbody spectrum with `Noiser`

Nice! Now whats makes them special is that we can apply it to the Phoenix model with no additional effort:

```
>>> from taurex.stellar import PhoenixStar
>>> new_star = enhance_class(PhoenixStar, [Noiser, ], temperature=5800, radius=1.0,
                             phoenix_path='/path/to/phoenix', noise_level=1e6)
```

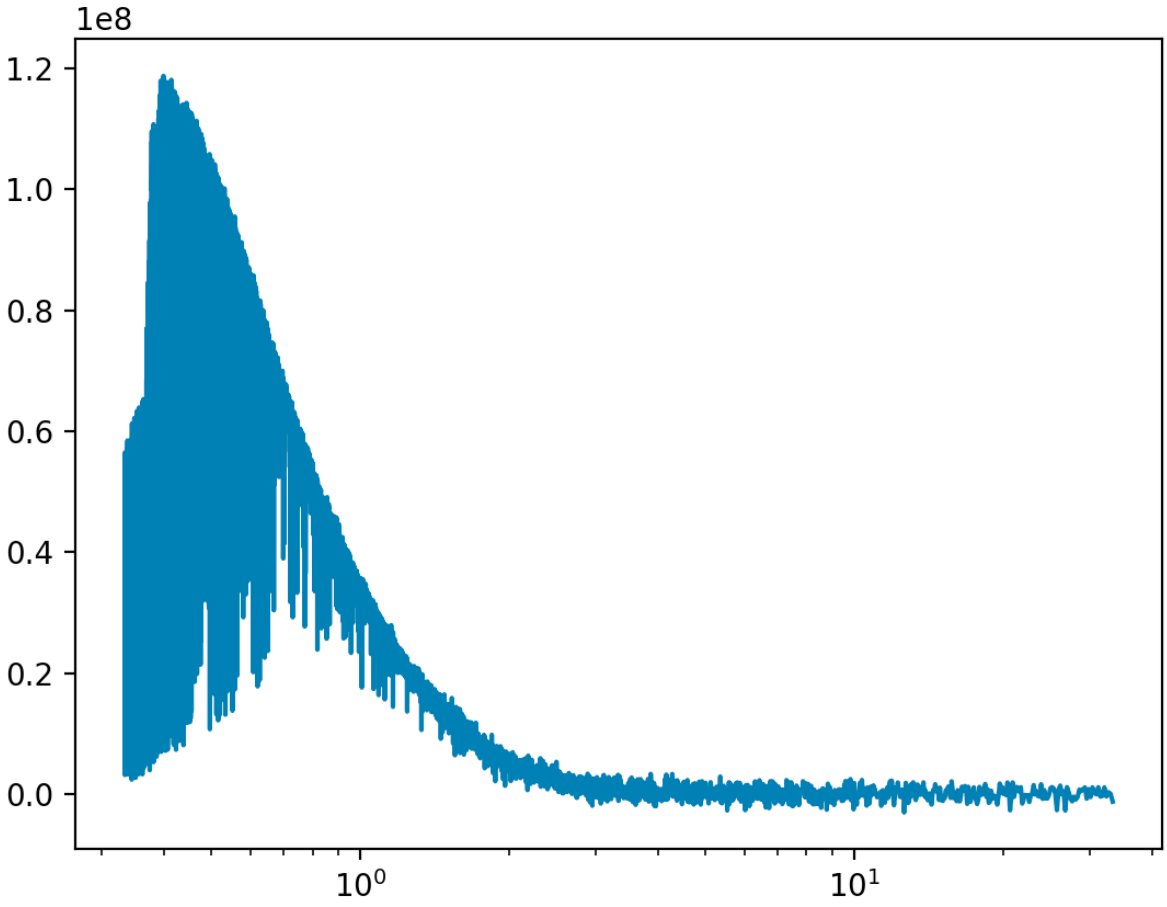


Fig. 3: Phoenix spectrum with `Noiser`

We can do the same with the `Denoiser`:

```
>>> new_star = enhance_class(PhoenixStar, [Denoiser, ], temperature=5800, radius=1.0,
                             phoenix_path='/path/to/phoenix', kernel_size=11)
```

The real magic is combining both!! We could heavily denoise the spectrum and then add noise:

```
>>> new_star = enhance_class(PhoenixStar, [Noiser, Denoiser, ], temperature=5800,␣
↪radius=1.0,
    phoenix_path='/path/to/phoenix', kernel_size=21, noise_level=1e6)
```

OR, add noise and then denoise it:

```
>>> new_star = enhance_class(PhoenixStar, [Denoiser, Noiser, ], temperature=5800,␣
↪radius=1.0,
    phoenix_path='/path/to/phoenix', kernel_size=21, noise_level=1e6)
```

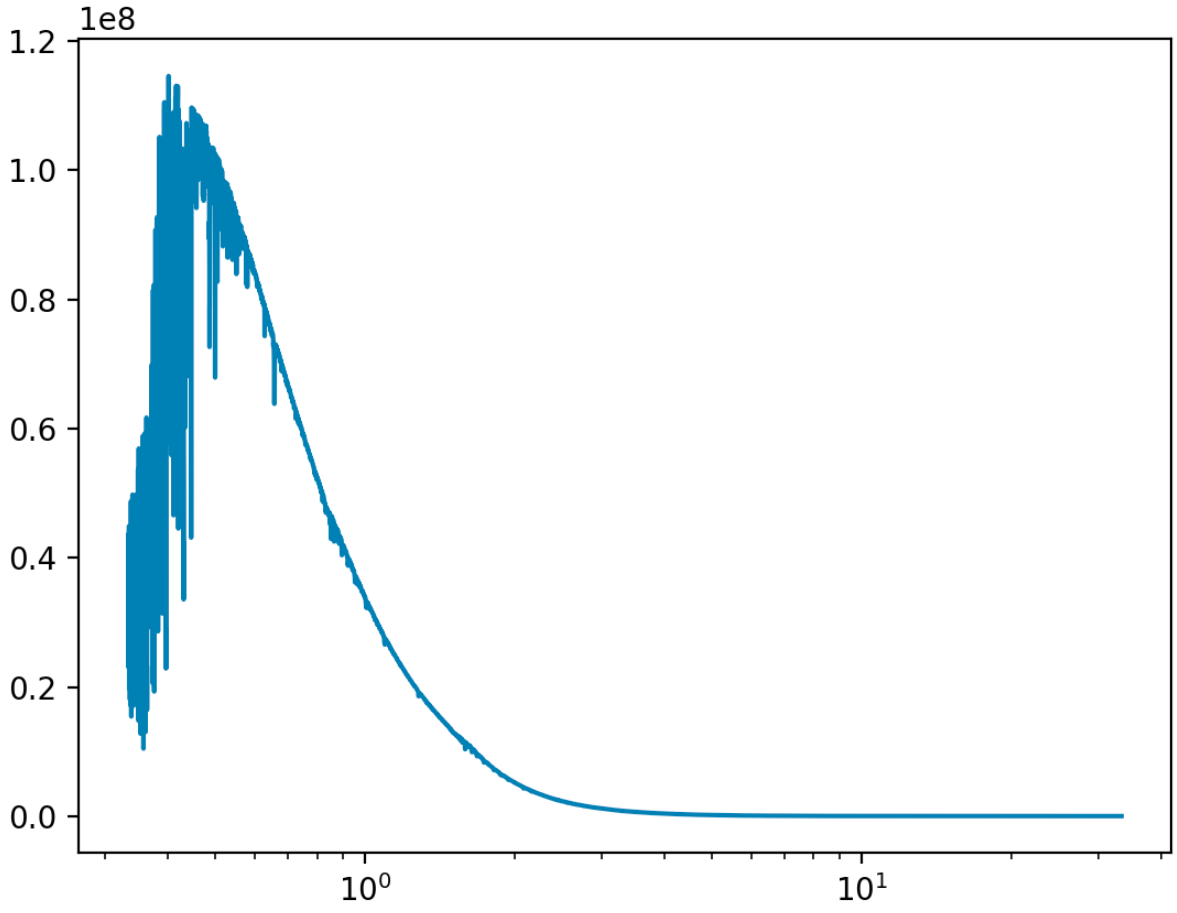


Fig. 4: Phoenix spectrum with `Denoiser`

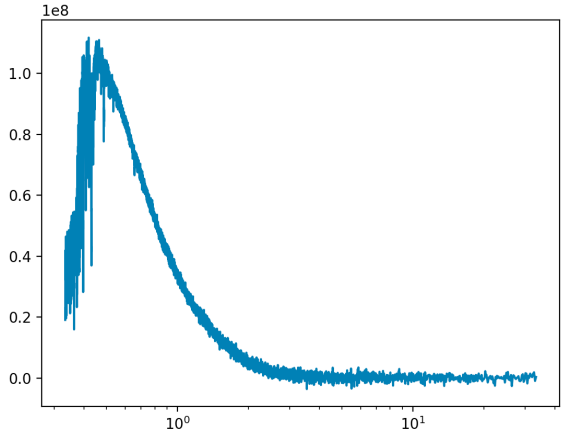


Fig. 5: Denoise then add noise

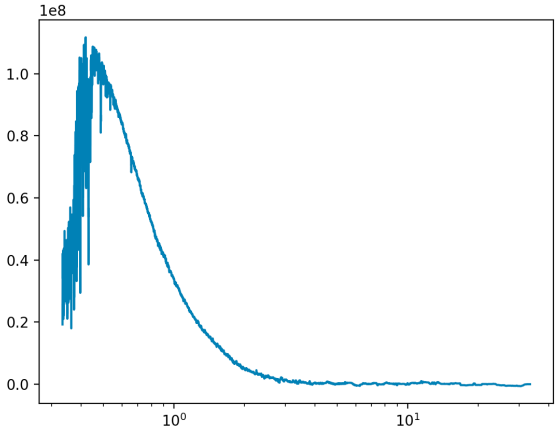


Fig. 6: Add Noise and then denoise

Whats important is the list of mixins is applied in *reverse*. `[Denoiser, Noiser]` does `Noiser` first *and then* `Denoiser`

8.7 Plugin Development

Atmospheric retrievals are not an isolated science. We regularly use different codes and models from various fields and contributors to better characterise exoplanetary systems. Often repetitive steps are needed to make use of an external code, and frequently, these are difficult to share or distribute to a broader audience. Plugins are a new feature in TauREx 3.1 that allows developers to simplify the distribution and usage of their profile/models/chemistry etc., to other users for their retrievals. The plugin system can be used to add the following new components:

- *TemperatureProfile*
- *Chemistry*
- *Gas*
- *PressureProfile*
- *Star*
- Planet
- *ForwardModel*
- *Contribution*
- *Opacity*
- Priors
- *Optimizer*
- *Mixin*

Refer to the *Developers guide* on how to build each individual component. This guide will outline how to package your new components into a TauREx plugin.

8.7.1 Anatomy of a Plugin

Plugins are installable python packages that TauREx will integrate automatically into its pipeline. Plugins can come from existing python libraries or dedicated packages. Dedicated packages only contain TauREx components and generally have the name like `taurex_something`. For example, a package that provides wrappers to the GGchem chemistry code would be called `taurex_ggchem` (which exists btw if you need it).

Importantly, open-source plugins should be registered to PyPI installable with a single command:

```
> pip install taurex_myplugin
```

If they rely on an external FORTRAN/C++ code then they should be packaged into a binary wheel distribution. We recommoned cibuildwheel for building these wheels. After install the plugin will be automatically detected by TauREx:

```
> taurex --plugins
Successfully loaded plugins
---------------------------
myplugin
```

A plugin, in its most basic form, points TauREx to the place where your components exist in your package. This is accomplised through the `entry_point` parameter in `setup.py` of the plugin package:

```
entry_points = {'taurex.plugins': 'myplugin = taurex_myplugin'}

setup(name='taurex_myplugin',
      ..
      entry_points=entry_points,
      ..)
```

What this does is allow TauREx to access `taurex_myplugin` under `taurex.plugins.myplugin`

Plugins can also be defined in existing packages as well. If you have a `coolscience` python library and have built some TauREx components under `coolscience.taurex` then you can add to your `entry_point`:

```
entry_points = {'taurex.plugins': 'coolscience = coolscience.taurex',
                # ... other entrypoints
                }

setup(name='coolscience',
      ...
      entry_points=entry_points,
      ...)
```

The package will still be installable without TauREx. If later on someone installs TauREx then they automatically get the plugin for free! Neat!

8.7.2 TauREx Hello World!

Lets create a first plugin `taurex_helloworld` where we will define a new component: a randomized temperature profile. First we setup our folder structure:

```
taurex_helloworld/
    __init__.py
    randomtemp.py
LICENSE
```

(continues on next page)

(continued from previous page)

```
README.md
setup.py
```

setup.py

The most essential part is the `setup.py` file to install the package and plugin. The following is something you can work with:

```
#!/usr/bin/env python
import setuptools
from setuptools import find_packages
from setuptools import setup

packages = find_packages(exclude=('tests', 'doc'))
provides = ['taurex_helloworld', ]

requires = []

install_requires = ['taurex', ]

entry_points = {'taurex.plugins': 'helloworld = taurex_helloworld'}

setup(name='taurex_helloworld',
      url='http://example.com/taurex_helloworld',
      license='BSD',
      author='Your Name',
      author_email='your-email@example.com',
      description='Very short description',
      long_description=__doc__,
      packages=packages,
      entry_points=entry_points,
      provides=provides,
      requires=requires,
      install_requires=install_requires)
```

randomtemp.py

This is our random temperature profile, we will steal the implementation from *Custom Types* and change it a little:

```
from taurex.temperature import TemperatureProfile
from taurex.core import fitparam
import numpy as np


class RandomTemperature(TemperatureProfile):

    def __init__(self, base_temp=1500.0,
                 random_scale=10.0):
        super().__init__(self.__class__.__name__)

        self._base_temp = base_temp
        self._random_scale = random_scale
```

(continues on next page)

(continued from previous page)

```
    # -----Fitting Parameters--------------

    @fitparam(param_name='rand_scale',param_latex='rand')
    def randomScale(self):
        return self._random_scale

    @randomScale.setter
    def randomScale(self, value):
        self._random_scale = value

    @fitparam(param_name='base_T',param_latex='$T_{base}$')
    def baseTemperature(self):
        return self._base_temp

    @baseTemperature.setter
    def baseTemperature(self, value):
        self._base_temp = value

    # -------Actual calculation -----------

    @property
    def profile(self):
        return self._base_temp + \
                    np.random.rand(self.nlayers) * self._random_scale

    BIBTEX_ENTRIES = [
        """
        @article{myart,
            title={School of Life},
        """
    ]

    # -----Plugin related------------------

    @classmethod
    def input_keywords(cls):
        return ['helloworld', 'helloearth', 'hello-world',]
```

As before a terrible temperature profile we now include two extra parameters. The class method `input_keywords` informs TauREx on how this temperature profile is selected in the input file. It must return a list and can include more than one keyword. If this plugin is installed we can use the profile through one of those keywords:

```
[Temperature]
profile_type = helloworld    # Valid keyword RandomTemperature
# profile_type = helloearth  # Also valid
# profile_type = hello-world # Also valid
```

The `BIBTEX_ENTRIES` parameter is used by TauREx to list relevant publications involved with the atmospheric compoenent. See *Basics* for more information.

__init__.py

We can use `__init__.py` to expose the temperature profile to TauREx by importing it like so:

```
from .randomtemp import RandomTemperature
```

Tip: You could also just point the `entry_point` to `taurex_helloworld.randomtemp`. However we recommend either putting it in an `__init__.py` or defining another python file that includes these imports. This allows you to include components from different files and allows you to be selective on what to expose to TauREx

Using our plugin

To use our plugin we can now do:

```
pip install .
```

Running `taurex --plugins` we see:

```
Successfully loaded plugins
---------------------------
helloworld
```

Our plugin has now been loaded into TauREx! We can also see that our temperature profile was detected as well by doing `taurex --keywords temperature`:

```
| profile_type                          | Class             | Source     |
| file / fromfile                       | TemperatureFile   | taurex     |
| isothermal                            | Isothermal        | taurex     |
| guillot / guillot2010                 | Guillot2010       | taurex     |
| npoint                                | NPoint            | taurex     |
| helloworld / helloearth / hello-world | RandomTemperature | helloworld |
| rodgers / rodgers2010                 | Rodgers2000       | taurex     |
```

Now we can write in the input file:

```
[Temperature]
profile_type = helloworld
base_temp = 500.0
random_scale = 100.0
```

Which gives us

This is a minimal guide to developing plugins but we always recommend looking at plugin projects and seeing how they accomplish their tasks.

8.8 Recipes

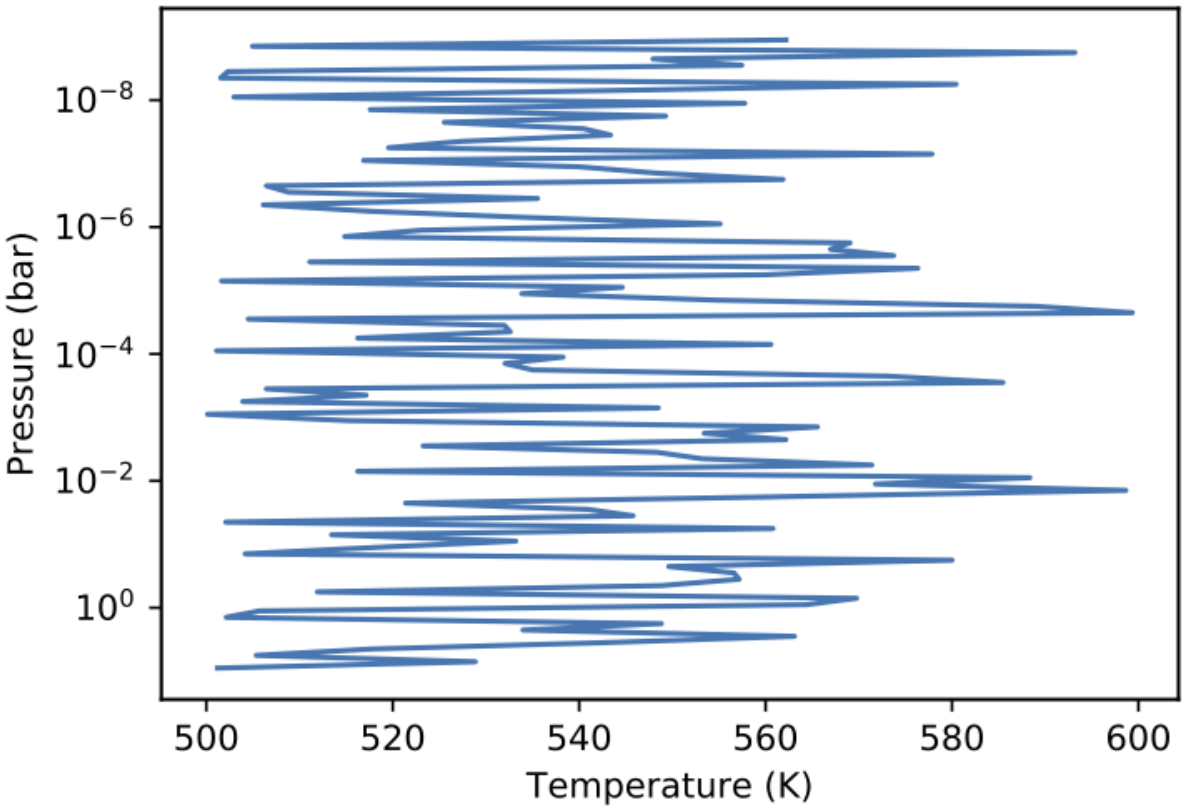


Fig. 7: *Still* terrible

CHAPTER

NINE

PLUGINS CATALOGUE

New in version 3.1.

Here is a list of plugins that can be installed to give TauREx 3 new features and components. Plugins are usually hosted on PyPi and may have precompiled binary wheels for Windows, MacOS and/or manylinux

Name	Description	PyPi	Wheels		
			Win64	MacOS	manylinux
taurex_ace	Equilibrium chemistry using ACE	✓	✓	✓	✓
taurex_fastchem	Equilibrium chemistry using FastChem	✓	✓	✓	✓
taurex_ggchem	Equilibrium chemistry using GGChem	✓	✓	✓	✓
taurex_cuda	CUDA-acceleration of forward models	✓			
taurex_hip	HIP-acceleration of forward models	✓			
taurex_dynesty	Dynesty optimizer	✓	✓	✓	✓
taurex_petitrad	petitRADTRANS forward models and opacity formats	✓	✓	✓	✓

CHAPTER

TEN

TAUREX API DOCUMENTATION

10.1 Core (`taurex.core`)

10.1.1 Retrieval

This module relates to defining fitting parameters in TauREx3

class Fittable

Bases: `object`

A class that manages fitting parameters. Not really used on its own it should really be inherited from to be used properly. It also provides class with the ability to read and write fitting parameters using their params names, for example, if we create a class like this:

```
class Foo(Fittable):

    def __init__(self):
        self.value = 10

    @fitparam(param_name='foobar',param_latex='$Foo^{bar}$',default_bounds=[1,12])
    def bar(self):
        return self.value

    @bar.setter
    def bar(self,value):
        self.value = value
```

We can read and write data in the standard python way like so:

```
>>> foo = Foo()
>>> foo.bar
10
>>> foo.bar = 20
>>> foo.bar
20
```

but we also get this functionality for free:

```
>>> foo['foobar']
20
>>> foo['foobar'] = 30
>>> foo['foobar']
30
```

add_derived_param(*param_name*, *param_latex*, *fget*, *compute*)

add_fittable_param(*param_name*, *param_latex*, *fget*, *fset*, *default_mode*, *default_fit*, *default_bounds*)

Adds a fittable parameter to the internal dictionary. Used during init to add all *fitparam()* decorated methods and can also be utilized by a user to manually add new fitting parameters. This is useful for giving fitting parameters names that depend on certain attributes (e.g. molecule name in a gas profile see *ConstantGas*) or when converting lists into fitting parameters (e.g. Normalization factor in light curves see: LightCurveModel)

Parameters

- **param_name** (*str*) – Nicer name of the parameter. Referenced by the optimizer.
- **param_latex** (*str*) – Latex version of the parameter name, useful in plotting and making figures
- **fget** (*function*) – a function that returns the value of the parameter
- **fset** (*function*) – a function the writes the value of the parameter
- **default_mode** (linear or log) – Defines how the optimizer should read and write the parameter. linear reads/write everything as is. log informs the optimizer to transform from native->log space when read and to transfrom log->native when writing. This also applies to the boundaries
- **default_fit** (*bool*) – Whether this is included in the fit without the user explicity saying so (Default: False)
- **default_bounds** (list) – Default minimum and maximum fitting boundary. Must always be defined in the native space

compile_fitparams()

Loops through and finds all fitting parameters in the class and adds it to the internal dictionary

derived_parameters()

Returns all derived fitting parameters

find_derivedparams()

Finds and returns fitting parameters

Yields method (*function*) – class method that is defined with the *fitparam()* decorator

find_fitparams()

Finds and returns fitting parameters

Yields method (*function*) – class method that is defined with the *fitparam()* decorator

fitting_parameters()

Returns all fitting parameters found as a dictionary

Returns

params – Dictionary with key as the parameter name (param_name) and value as a tuple with:

- parameter name
- parameter name in Latex form
- get function
- set function
- fitting scale

- fit as default
- fitting boundaries

Return type dict

modify_bounds(*parameter*, *new_bounds*)
Modifies the fitting boundary of a parameter

Parameters

- **parameter** (*str*) – Name of parameter (given by param_name in *fitparam()*)
- **new_bounds** (list) – New minimum and maximum fitting boundaries.

derivedparam(*f=None*, *param_name=None*, *param_latex=None*, *compute=False*)
A decorator used in conjunction with *Fittable* to inform which parameters should be derived during retrieval. This allows for posteriors of parameters such as log(g) and mu

Parameters

- **f** (*function*) – Function being passed. Automatically done when used as a decorator
- **param_name** (*str*) – Nicer name of the parameter. Referenced by the optimizer.
- **param_latex** (*str*) – Latex version of the parameter name, useful in plotting and making figures
- **compute** (*bool*) – By default, is this computed?

fitparam(*f=None, param_name=None, param_latex=None, default_mode='linear', default_fit=False, default_bounds=[0.0, 1.0]*)
A decorator used in conjunction with *Fittable* to inform which parameters can be fit and its properties. On its own it acts like the property decorator. When used within a *Fittable* class it serves to tag a property as able to fit and allows the class to compile all parameters that can be fit.

Its usage is simple, simply wrap a method and define its properties:

```
class Foo(Fittable):

    @fitparam(param_name='foobar',param_latex='$Foo^{bar}$')
    def bar(self):
        return 'Foobar'

    @bar.setter
    def bar(self,value):
        self.value = value
```

Parameters

- **f** (*function*) – Function being passed. Automatically done when used as a decorator
- **param_name** (*str*) – Nicer name of the parameter. Referenced by the optimizer.
- **param_latex** (*str*) – Latex version of the parameter name, useful in plotting and making figures
- **default_mode** (linear or log) – Defines how the optimizer should read and write the parameter. linear reads/write everything as is. log informs the optimizer to transform from native->log space when read and to transfrom log->native when writing. This also applies to the boundaries
- **default_fit** (*bool*) – Whether this is included in the fit without the user explicity saying so (Default: False)

- **default_bounds** (list) – Default minimum and maximum fitting boundary. Must always be defined in linear space

10.1.2 Bibliography

class Citable
Bases: object

Defines a class that contains citation information.

BIBTEX_ENTRIES = []
List of bibtext entries

citations()

nice_citation(*prefix=''*, *start_idx=0*, *indent=0*)

cleanup_string(*string*)

construct_nice_printable_string(*entry*, *indent=0*)

doi_to_bibtex

handle_publication(*fields*)

recurse_bibtex(*obj*, *entries*)

stringify_people(*authors*)

to_bibtex(*citations*)

unique_citations_only(*citations*)

10.2 Priors (taurex.core.priors)

class Gaussian(*mean=0.5*, *std=0.25*)
Bases: *taurex.core.priors.Prior*

boundaries()

params()

sample(*x*)

class LogGaussian(*mean=0.5*, *std=0.25*, *lin_mean=None*, *lin_std=None*)
Bases: *taurex.core.priors.Gaussian*

class LogUniform(*bounds=[0.0, 1.0]*, *lin_bounds=None*)
Bases: *taurex.core.priors.Uniform*

class Prior
Bases: *taurex.log.logger.Logger*

Defines a prior function

boundaries()

params()

prior(*value*)

property priorMode

sample(*x*)

class PriorMode

Bases: enum.Enum

Defines the type of prior space

LINEAR = (0,)

LOG = (1,)

class Uniform(*bounds=[0.0, 1.0]*)

Bases: *taurex.core.priors.Prior*

boundaries()

params()

sample(*x*)

set_bounds(*bounds*)

10.3 Binning Module (taurex.binning)

The classes here deal with binning spectra down to different resolutions. These exist within the taurex.binning namespace

10.3.1 Base Module

Module for the base binning class

class Binner

Bases: *taurex.log.logger.Logger*

Abstract class

The binner class deals with binning down spectra to different resolutions. It also provides a method to generate spectrum output format from a forward model result in the form of a dictionary. Using this class does not need to be restricted to TauREx3 results and can be used to bin down any arbitrary spectra.

bin_model(*model_output*)

Bins down a TauREx3 forward model. This automatically splits the output and passes it to the *bindown()* function. Its general usage is of the form:

```
>>> fm = TransmissionModel()
>>> fm.build()
>>> result = fm.model()
>>> binner.bin_model(result)
```

Or in a single line:

```
>>> binner.bin_model(fm.model())
```

Parameters **model_output** (obj:*tuple*) – Result from running a TauREx3 forward model

Returns

Return type See *bindown()*

bindown (*wngrid*, *spectrum*, *grid_width=None*, *error=None*)

Requires implementation

This should handle the binning of a spectrum passed into the function. Parameters given are guidelines on expectation of usage.

Parameters

- **wngrid** (array) – The wavenumber grid of the spectrum to be binned down. Generally the 'native' wavenumber grid
- **spectrum** (array) – The spectra we wish to bin-down. Must be same shape as wngrid.
- **grid_width** (array, optional) – Wavenumber grid full-widths for the spectrum to be binned down. Must be same shape as wngrid. Optional, generally if you require this but the user does not pass it then you must compute it yourself using wngrid. This can be done easily using the function func:*~taurex.util.util.compute_bin_edges*.
- **error** (array, optional) – Associated errors or noise of the spectrum. Must be same shape as wngrid.Optional parameter, when implementing you must deal with the cases where either the error is passed or not passed.

Returns

- **binned_wngrid** (array) – New wavenumber grid
- **spectrum** (array) – Binned spectrum.
- **grid_width** (array) – New grid-widths
- **error** (array or None) – If passed, should be the binned error otherwise None

generate_spectrum_output (*model_output*, *output_size=<OutputSize.heavy: 6>*)

Given a forward model output, generate a dictionary that can be used to store to file. This can include storing the native and binned spectrum. Not necessary for the function of the class but useful for full intergation into TauREx3, especially when storing results from a retrieval. Can be overwritten to store more information.

Parameters

- **model_output** (obj:*tuple*) – Result from running a TauREx3 forward model
- **output_size** (OutputSize) – Size of the output.

Returns Dictionary of spectra

Return type dict

10.3.2 Flux-Binning

class FluxBinner (*wngrid*, *wngrid_width=None*)

Bases: *taurex.binning.binner.Binner*

Bins to a wavenumber grid given by wngrid using a more accurate method that takes into account the amount of contribution from each native bin. This method also handles cases where bins are not continuous and/or overlapping.

Parameters

- **wngrid** (array) – Wavenumber grid

- **wngrid_width** (array, optional) – Must have same shape as wngrid Full bin widths for each wavenumber grid point given in wngrid. If not provided then this is automatically computed from wngrid.

bindown (*wngrid*, *spectrum*, *grid_width=None*, *error=None*)
Bins down spectrum.

Parameters

- **wngrid** (array) – The wavenumber grid of the spectrum to be binned down.
- **spectrum** (array) – The spectra we wish to bin-down. Must be same shape as wngrid.
- **grid_width** (array, optional) – Wavenumber grid full-widths for the spectrum to be binned down. Must be same shape as wngrid. Optional.
- **error** (array, optional) – Associated errors or noise of the spectrum. Must be same shape as wngrid.Optional parameter.

Returns

- **binned_wngrid** (array) – New wavenumber grid
- **spectrum** (array) – Binned spectrum.
- **grid_width** (array) – New grid-widths
- **error** (array or None) – Binned error if given else None

generate_spectrum_output (*model_output*, *output_size=<OutputSize.heavy: 6>*)
Given a forward model output, generate a dictionary that can be used to store to file. This can include storing the native and binned spectrum. Not necessary for the function of the class but useful for full intergation into TauREx3, especially when storing results from a retrieval. Can be overwritten to store more information.

Parameters

- **model_output** (obj:*tuple*) – Result from running a TauREx3 forward model
- **output_size** (OutputSize) – Size of the output.

Returns Dictionary of spectra

Return type dict

10.3.3 Simple-Binning

class SimpleBinner (*wngrid*, *wngrid_width=None*)
Bases: *taurex.binning.binner.Binner*

Bins to a wavenumber grid given by wngrid. The method places flux into the correct bins using histogramming methods. This is fast but can suffer as it assumes that there are no gaps in the wavenumber grid. This can cause weird results and may cause the flux to be higher in the boundary of points between two distinct regions (such as WFC3 + Spitzer)

Parameters

- **wngrid** (array) – Wavenumber grid
- **wngrid_width** (array, optional) – Must have same shape as wngrid Full bin widths for each wavenumber grid point given in wngrid. If not provided then this is automatically computed from wngrid.

bindown (*wngrid*, *spectrum*, *grid_width=None*, *error=None*)
Bins down spectrum.

Parameters

- **wngrid** (array) – The wavenumber grid of the spectrum to be binned down.
- **spectrum** (array) – The spectra we wish to bin-down. Must be same shape as wngrid.
- **grid_width** (array, optional) – Wavenumber grid full-widths for the spectrum to be binned down. Must be same shape as wngrid. Optional.
- **error** (array, optional) – Associated errors or noise of the spectrum. Must be same shape as wngrid.Optional parameter.

Returns

- **binned_wngrid** (array) – New wavenumber grid
- **spectrum** (array) – Binned spectrum.
- **grid_width** (array) – New grid-widths
- **error** (array or None) – Binned error if given else None

generate_spectrum_output (*model_output*, *output_size=<OutputSize.heavy: 6>*)
Given a forward model output, generate a dictionary that can be used to store to file. This can include storing the native and binned spectrum. Not necessary for the function of the class but useful for full intergation into TauREx3, especially when storing results from a retrieval. Can be overwritten to store more information.

Parameters

- **model_output** (obj:*tuple*) – Result from running a TauREx3 forward model
- **output_size** (OutputSize) – Size of the output.

Returns Dictionary of spectra

Return type dict

10.3.4 Lightcurve-Binning

class LightcurveBinner
Bases: *taurex.binning.binner.Binner*

A special class of binning used to generate the correct spectrum output. This is essentially the same as *NativeBinner* but for lightcurve forward models.

bindown (*wngrid*, *spectrum*, *grid_width=None*, *error=None*)
Does nothing, only returns function arguments

generate_spectrum_output (*model_output*, *output_size=<OutputSize.heavy: 6>*)
Accepts only a lightcurve forward model. Stores the lightcurve as well as the spectrum.

Parameters

- **model_output** (obj:*tuple*) – Result from running a TauREx3 lightcurve forward model
- **output_size** (OutputSize) – Size of the output.

Returns Dictionary of spectra containing both lightcurves and spectra.

Return type dict

10.3.5 Native-Binning

class NativeBinner
Bases: *taurex.binning.binner.Binner*

A *do-nothing* binner. This is useful when the pipeline expects a binner but none is given. Simplifies implementation and also handles dictionary writing of the forward model.

bindown (*wngrid*, *spectrum*, *grid_width=None*, *error=None*)
Does nothing, only returns function arguments

generate_spectrum_output (*model_output*, *output_size=<OutputSize.heavy: 6>*)
Given a forward model output, generate a dictionary that can be used to store to file. This can include storing the native and binned spectrum. Not necessary for the function of the class but useful for full intergation into TauREx3, especially when storing results from a retrieval. Can be overwritten to store more information.

Parameters

- **model_output** (obj:*tuple*) – Result from running a TauREx3 forward model
- **output_size** (OutputSize) – Size of the output.

Returns Dictionary of spectra

Return type dict

10.4 Caching Modules (taurex.cache)

10.4.1 Singleton

Just contains a singleton class. Pretty useful

class Singleton
Bases: object

A singleton for your usage. When inheriting do not implement __init__ instead override *init()*

init (**args*, ***kwds*)
Override to act as an init

10.4.2 OpacityCache

Contains caching class for Molecular cross section files

class OpacityCache
Bases: taurex.core.Singleton

Implements a lazy load of opacities. A singleton that loads and caches xsections as they are needed. Calling

```
>>> opt = OpacityCache()
>>> opt2 = OpacityCache()
```

Reveals that:

```
>>> opt == opt2
True
```

Importantly this class will automatically search directories for cross-sections set using the *set_opacity_path()* method:

```
>>> opt.set_opacity_path('path/to/crossections')
```

Multiple paths can be set as well

```
>>> opt.set_opacity_path(['/path/to/crosssections','/another/path/to/crosssections
↪'])
```

To get the cross-section object for a particular molecule use the square bracket operator:

```
>>> opt['H2O']
<taurex.opacity.pickleopacity.PickleOpacity at 0x107a60be0>
```

This returns a *Opacity* object for you to compute H2O cross sections from. When called for the first time, a directory search is performed and, if found, the appropriate cross-section is loaded. Subsequent calls will immediately return the already loaded object:

```
>>> h2o_a = opt['H2O']
>>> h2o_b = opt['H2O']
>>> h2o_a == h2o_b
True
```

If you have any plugins that include new opacity formats, the cache will automatically detect them.

Lastly you can manually add an opacity directly for a molecule into the cache:

```
>>> new_h2o = MyNewOpacityFormat()
>>> new_h2o.molecule
H2O
>>> opt.add_opacity(new_h2o)
>>> opt['H2O']
<MyNewOpacityFormat at 0x107a60be0>
```

Now TauREx3 will use it instead in all calculations!

add_opacity(*opacity*, *molecule_filter=None*)
Adds a *Opacity* object to the cache to then be used by Taurex 3

Parameters

- **opacity** (*Opacity*) – Opacity object to add to the cache
- **molecule_filter** (list of str , optional) – If provided, the opacity object will only be included if its molecule is in the list. Mostly used by the __getitem__() for filtering

clear_cache()
Clears all currently loaded cross-sections

enable_radis(*enable*)
Enables/Disables use of RADIS to fill in missing molecules using HITRAN.

> **Warning:** This is extremely unstable and crashes frequently. It is also very slow as it requires the computation of the Voigt profile for every temperature. We recommend leaving it as False unless necessary.

Parameters **enable** (*bool*) – Whether to enable RADIS functionality (default = False)

find_list_of_molecules()

force_active(*molecules*)
Allows some molecules to be forced as active. Useful when using other radiative codes to do the calculation

Parameters **molecules** (obj:*list*) – List of molecules

init()
Override to act as an init

load_opacity(*opacities=None*, *opacity_path=None*, *molecule_filter=None*)
Main function to use when loading molecular opacities. Handles both cross sections and paths. Handles lists of either so lists of *Opacity* objects or lists of paths can be used to load multiple files/objects

Parameters

- **opacities** (*Opacity* or list of *Opacity* , optional) – Object(s) to include in cache
- **opacity_path** (str or list of str, optional) – search path(s) to look for molecular opacities
- **molecule_filter** (list of str , optional) – If provided, the opacity will only be loaded if its molecule is in this list. Mostly used by the __getitem__() for filtering

load_opacity_from_path(*path*, *molecule_filter=None*)
Searches path for molecular cross-section files, creates and loads them into the cache .pickle will be loaded as *PickleOpacity*

Parameters

- **path** (*str*) – Path to search for molecular cross-section files
- **molecule_filter** (list of str , optional) – If provided, the opacity will only be loaded if its molecule is in this list. Mostly used by the __getitem__() for filtering

set_interpolation(*interpolation_mode*)
Sets the interpolation mode for all currently loaded (and future loaded) cross-sections

Can either be linear for linear interpolation of both temeprature and pressure:

```
>>> OpacityCache().set_interpolation('linear')
```

or exp for natural exponential interpolation of temperature and linear for pressure

```
>>> OpacityCache().set_interpolation('exp')
```

Parameters **interpolation_mode** (*str*) – Either linear for bilinear interpolation or exp for exp-linear interpolation

set_memory_mode(*in_memory*)
If using the HDF5 opacities, whether to stream opacities from file (slower, less memory) or load them into memory (faster, more memory)

Parameters **in_memory** (*bool*) – Whether HDF5 files should be streamed (False) or loaded into memory (True, default)

set_opacity_path(*opacity_path*)
Set the path(s) that will be searched for opacities. Opacities in this path must be of supported types:

- HDF5 opacities
- .pickle opacities

- ExoTransmit opacities.

  Parameters **opacity_path** (str or list of str, optional) – search path(s) to look for molecular opacities

set_radis_wavenumber(*wn_start*, *wn_end*, *wn_points*)

10.4.3 CIACache

Contains caching class for Collisionally Induced Absorption files

class CIACache

Bases: taurex.core.Singleton

Implements a lazy load of collisionally induced absorpiton cross-sections Supports pickle files and HITRAN cia files. Functionally behaves the same as *OpacityCache* except the keys are now cia pairs e.g:

```
>>> CIACache()['H2-H2']
<taurex.cia.picklecia.PickleCIA at 0x107a60be0>
```

Pickle .db and HITRAN .cia files are supported and automatically loaded. with priority given to .db files

add_cia(*cia*, *pair_filter=None*)

Adds a *CIA* object to the cache to then be used by Taurex 3

Parameters

- **cia** (*CIA*) – CIA object to add to the cache
- **pair_filter** (list of str , optional) – If provided, the cia object will only be included if its pairname is in the list. Mostly used by the __getitem__() for filtering

init()

Override to act as an init

load_cia(*cia_xsec=None*, *cia_path=None*, *pair_filter=None*)

Main function to use when loading CIA files. Handles both cross sections and paths. Handles lists of either so lists of *CIA* objects or lists of paths can be used to load multiple files/objects

Parameters

- **cia_xsec** (*CIA* or list of *CIA* , optional) – Object(s) to include in cache
- **cia_path** (str or list of str, optional) – search path(s) to look for cias
- **pair_filter** (list of str , optional) – If provided, the cia will only be loaded if its pair name is in this list. Mostly used by the __getitem__() for filtering

load_cia_from_path(*path*, *pair_filter=None*)

Searches path for CIA files, creates and loads them into the cache .db will be loaded as *PickleCIA* and .cia files will be loaded as *HitranCIA*

Parameters

- **path** (*str*) – Path to search for CIA files
- **pair_filter** (list of str , optional) – If provided, the cia will only be loaded if its pairname is in the list. Mostly used by the __getitem__() for filtering

set_cia_path(*cia_path*)

Sets the path to search for CIA files

Parameters **cia_path** (str or `list` of str) – Either a single path or a list of paths that contain CIA files

10.5 CIA (`taurex.cia`)

10.5.1 Base Class

Contains the abstract class used by all collisionally induced absorption objects.

class CIA(*name*, *pair_name*)

Bases: *taurex.log.logger.Logger*

Abstract class

This is the base class for collisionally induced absorption opacities. To function in Taurex3, it requires concrete implementations of:

- *wavenumberGrid()*
- *compute_cia()*
- *temperatureGrid()*

Parameters

- **name** (*str*) – Name to use for logging
- **pair_name** (*str*) – pair of molecules this class represents. e.g. 'H2-H2' or 'H2-He'

cia(*temperature*, *wngrid=None*)

For a given temperature, computes the appropriate cross section. If wavenumber grid (`wngrid`) is provided then the cross-section is interpolated to it.

Parameters

- **temperature** (*float*) – Temeprature in Kelvin
- **wngrid** (`array` , optional) – Wavenumber grid to interpolate to

Returns CIA cross section at desired temeprature on either its native grid or interpolated on `wngrid` if supplied

Return type `array`

compute_cia(*temperature*)

Computes the collisionaly induced cross-section for a given temeprature

Unimplemented, this must be implemented in any derived class to be considered compatible in Taurex3

The rules are:

1. It must accept temperature in Kelvin (K)
2. If the temperature falls outside of *temperatureGrid()* it must be set to zero
3. The returned array must be of equal size to *wavenumberGrid()*

Parameters **temperature** (*float*) – Temeprature in Kelvin

Returns CIA cross section at desired temeprature on its native grid

Return type `array`

Raises **NotImplementedError** – Only if derived class does not implement this

property pairName
The assigned pair of molecules of this CIA

Returns The pair of molecules of this object in the form: `Molecule1-Molecule2`

Return type str

property pairOne
The name of the first molecule in the pair

Returns First molecule in the pair

Return type str

property pairTwo
The name of the second molecule in the pair

Returns Second molecule in the pair

Return type str

property temperatureGrid
The native temperature grid of the CIA cross-sections.

Returns Native temeprature grid in Kelvin

Return type `array`

Raises **NotImplementedError** – Only if derived class does not implement this

property wavenumberGrid
The native wavenumber grid (cm-1) of the CIA. Must be implemented in derived classes

Returns Native wavenumber grid

Return type `array`

Raises **NotImplementedError** – Only if derived class does not implement this

10.5.2 HITRAN CIA (`.cia`)

Module contains classes that handle loading of HITRAN cia files

exception EndOfHitranCIAException
Bases: `Exception`

An exception that occurs when the end of a HITRAN file is reached

class HitranCIA(*filename*)
Bases: *taurex.cia.cia.CIA*

A class that directly deals with HITRAN cia files and turns them into generic CIA objects that nicely produces cross sections for us. This will handle CIAs that have wavenumber grids split across temperatures by unifying them into single grids.

To use it simply do:

```
>>> h2h2 = HitranCIA('path/to/H2-He.cia')
```

And now you can painlessly compute cross-sections like this:

```
>>> h2h2.cia(400)
```

Or if you have a wavenumber grid, we can also interpolate it:

```
>>> h2h2.cia(400,mywngrid)
```

And all it cost was buying me a beer!

Parameters **filename** (*str*) – Path to HITRAN cia file

compute_cia(*temperature*)
Computes the collisionally induced absorption cross-section using our final native temperature and cross-section grids

Parameters **temperature** (*float*) – Temperature in Kelvin

Returns **out** – Temperature interpolated cross-section

Return type array

compute_final_grid()
Collects all *HitranCiaGrid* objects we've created and unifies them into a single temperature, cross-section and wavenumber grid for us to FINALLY interpolate and produce collisionaly induced cross-sections

fill_gaps(*temperature*)
Fills gaps in temperature grid for all wavenumber grid objects we've created

Parameters **temperature** (*array_like*) – Master temperature grid

find_closest_temperature_index(*temperature*)
Finds the nearest indices for a particular temperature

Parameters **temperature** (*float*) – Temeprature in Kelvin

Returns

- **t_min** (*int*) – index on temprature grid to the left of temperature
- **t_max** (*int*) – index on temprature grid to the right of temperature

interp_linear_grid(*T*, *t_idx_min*, *t_idx_max*)
For a given temperature and indicies. Interpolate the cross-sections linearly from temperature grid to temperature T

Parameters

- **temperature** (*float*) – Temeprature in Kelvin
- **t_min** (*int*) – index on temprature grid to the left of temperature
- **t_max** (*int*) – index on temprature grid to the right of temperature

Returns **out** – Interpolated cross-section

Return type array

load_hitran_file(*filename*)
Handles loading of the HITRAN file by reading and figuring out the wavenumber and temperature grids and matching them up

Parameters **filename** (*str*) – Path to HITRAN cia file

read_header(*f*)
Reads single header in the file

Parameters **f** (*file object*) –

Returns

- **start_wn** (*float*) – Start wavenumber for temperature
- **end_wn** (*float*) – End wavenumber for temperature
- **total_points** (*int*) – total number of points in temperature
- **T** (*float*) – Temperature in Kelvin
- **max_cia** (*float*) – Maximum CIA value in temperature

property temperatureGrid
Unified temperature grid

Returns Native temperature grid in Kelvin

Return type array

property wavenumberGrid
Unified wavenumber grid

Returns Native wavenumber grid

Return type array

class HitranCiaGrid(*wn_min*, *wn_max*)
Bases: *taurex.log.logger.Logger*

Class that handles a particular HITRAN cia wavenumber grid Since temperatures for CIA sometimes have different wavenumber grids this class helps to simplify managing them by only dealing with one at a time. These will help us unify into a single grid eventually

Parameters

- **wn_min** (*float*) – The minimum wavenumber for this grid
- **wn_max** (*float*) – The maximum wavenumber for this grid

add_temperature(*T*, *sigma*)
Adds a temeprature and crossection to this wavenumber grid

Parameters

- **T** (*float*) – Temeprature in Kelvin
- **sigma** (array) – cross-sections for this grid

fill_temperature(*temperatures*)
Here the 'master' temperature grid is passed into here and any gaps in our grid is filled with zero cross-sections to produce our final temperature-crosssection grid that matches with every other wavenumber grid. Temperatures that don't exist in the current grid but are withing the minimum and maximum for us are produced by linear interpolation

Parameters **temperatures** (*array_like*) – Master temperature grid

find_closest_temperature_index(*temperature*)
Finds the nearest indices for a particular temperature

Parameters **temperature** (*float*) – Temeprature in Kelvin

Returns

- **t_min** (*int*) – index on temprature grid to the left of temperature
- **t_max** (*int*) – index on temprature grid to the right of temperature

interp_linear_grid(*T*, *t_idx_min*, *t_idx_max*)
For a given temperature and indicies. Interpolate the cross-sections linearly from temperature grid to temperature `T`

Parameters

- **temperature** (*float*) – Temeprature in Kelvin
- **t_min** (*int*) – index on temprature grid to the left of `temperature`
- **t_max** (*int*) – index on temprature grid to the right of `temperature`

Returns **out** – Interpolated cross-section

Return type `array`

property sigma
Gets the currently loaded crossections for this wavenumber grid

Returns Cross-section grid

Return type `array`

sortTempSigma()
Sorts the temperature-sigma list

property temperature
Gets the current temeprature grid for this wavenumber grid

Returns Temeprature grid in Kelvin

Return type `array`

hashwn(*start_wn*, *end_wn*)
Simple wavenumber hash function

10.5.3 Pickle CIA (`.db`)

class PickleCIA(*filename*, *pair_name=None*)
Bases: *taurex.cia.cia.CIA*

Class for using pickled (`.db`) collisionally induced absorptions Very simple since the format is simple

Parameters

- **filename** (*str*) – Path to pickle
- **pair_name** (*str , optional*) – Whilst the name of the pair is determined by the pickle filename since these can be different you can optionally force the name through this parameter

compute_cia(*temperature*)
Computes the collisionally induced absorption cross-section using our native temperature and cross-section grids

Parameters **temperature** (*float*) – Temperature in Kelvin

Returns **out** – Temperature interpolated cross-section

Return type `array`

find_closest_temperature_index(*temperature*)
Finds the nearest indices for a particular temperature

Parameters **temperature** (*float*) – Temeprature in Kelvin

Returns

- **t_min** (*int*) – index on temprature grid to the left of `temperature`
- **t_max** (*int*) – index on temprature grid to the right of `temperature`

interp_linear_grid(*T*, *t_idx_min*, *t_idx_max*)
For a given temperature and indicies. Interpolate the cross-sections linearly from temperature grid to temperature `T`

Parameters

- **temperature** (*float*) – Temeprature in Kelvin
- **t_min** (*int*) – index on temprature grid to the left of `temperature`
- **t_max** (*int*) – index on temprature grid to the right of `temperature`

Returns **out** – Interpolated cross-section

Return type `array`

property temperatureGrid
returns: Native temperature grid in Kelvin :rtype: `array`

property wavenumberGrid
returns: Native wavenumber grid :rtype: `array`

10.6 Opacities (''taurex.opacity'')

10.6.1 Base

class Opacity(*name*)
Bases: *taurex.log.logger.Logger*, *taurex.data.citation.Citable*

This is the base class for computing opactities

compute_opacity(*temperature*, *pressure*, *wngrid=None*)
Must return in units of cm2

classmethod discover()
Class method, used to discover molecular opacities of this type.

property moleculeName

opacity(*temperature*, *pressure*, *wngrid=None*)

opacityCitation()
Citation for the specific molecular opacity (linelist origin etc)

Returns List of string with reference information

Return type list of str

property pressureGrid

classmethod priority()

property resolution

property temperatureGrid

property wavenumberGrid

10.6.2 Base Interpolator

class InterpolatingOpacity(*name*, *interpolation_mode='linear'*)
Bases: *taurex.opacity.opacity.Opacity*

Provides interpolation methods

compute_opacity(*temperature*, *pressure*, *wngrid=None*)
Must return in units of cm2

find_closest_index(*T*, *P*)

interp_bilinear_grid(*T*, *P*, *t_idx_min*, *t_idx_max*, *p_idx_min*, *p_idx_max*, *wngrid_filter=None*)

interp_pressure_only(*P*, *p_idx_min*, *p_idx_max*, *T*, *filt*)

interp_temp_only(*T*, *t_idx_min*, *t_idx_max*, *P*, *filt*)

property logPressure

property pressureBounds

property pressureMax

property pressureMin

set_interpolation_mode(*interp_mode*)

property temperatureBounds

property temperatureMax

property temperatureMin

property xsecGrid

10.6.3 Pickle Format (`.pickle`)

class PickleOpacity(*filename*, *interpolation_mode='linear'*)
Bases: *taurex.opacity.interpolateopacity.InterpolatingOpacity*

This is the base class for computing opactities

clean_molecule_name()

classmethod discover()
Class method, used to discover molecular opacities of this type.

property moleculeName

property pressureGrid

property resolution

property temperatureGrid

property wavenumberGrid

property xsecGrid

10.6.4 HDF5 Format (`.hdf5`)

class HDF5Opacity(*filename*, *interpolation_mode='exp'*, *in_memory=False*)
Bases: *taurex.opacity.interpolateopacity.InterpolatingOpacity*

This is the base class for computing opactities

BIBTEX_ENTRIES = ['\n @ARTICLE{2021A&A...646A..21C,\n author = {{Chubb}, Katy L. and {

citations()

classmethod discover()
Class method, used to discover molecular opacities of this type.

property moleculeName

opacityCitation()
Citation for the specific molecular opacity (linelist origin etc)

Returns List of string with reference information

Return type list of str

property pressureGrid

classmethod priority()

property resolution

property temperatureGrid

property wavenumberGrid

property xsecGrid

10.6.5 ExoTransmit Format (`.dat`)

class ExoTransmitOpacity(*filename*, *interpolation_mode='linear'*)
Bases: *taurex.opacity.interpolateopacity.InterpolatingOpacity*

BIBTEX_ENTRIES = ['\n @ARTICLE{2017PASP..129d4402K,\n author = {{Kempton}, Eliza M. -R

classmethod discover()
Class method, used to discover molecular opacities of this type.

property moleculeName

property pressureGrid

property resolution

property temperatureGrid

property wavenumberGrid

property xsecGrid

10.7 Contributions (`taurex.contributions`)

Classes related to the computation of the optical depth

10.7.1 Base Contribution

Base contribution classes and functions for computing optical depth

class Contribution(*name*)

Bases: *taurex.data.fittable.Fittable*, *taurex.log.logger.Logger*, *taurex.output.writeable.Writeable*, *taurex.data.citation.Citable*

Abstract class

The base class for modelling contributions to the optical depth. By default this handles contributions from cross-sections. If the type of contribution being implemented is a *sigma*-type like the form given in *contribute_tau()* then To function in Taurex3, it only requires the concrete implementation of:

- *prepare_each()*

Different forms may require reimplementing *contribute()* as well as *prepare()*

Parameters **name** (*str*) – Identifier of the contribution.

build(*model*)

Called during forward model build phase Does nothing by default

Parameters **model** (*ForwardModel*) – Forward model

contribute(*model*, *start_layer*, *end_layer*, *density_offset*, *layer*, *density*, *tau*, *path_length=None*)

Computes an integral for a single layer for the optical depth.

Parameters

- **model** (*ForwardModel*) – A forward model
- **start_layer** (*int*) – Lowest layer limit for integration
- **end_layer** (*int*) – Upper layer limit of integration
- **density_offset** (*int*) – offset in density layer
- **layer** (*int*) – atmospheric layer being computed
- **density** (array) – density profile of atmosphere
- **tau** (array) – optical depth to store result
- **path_length** (array) – integration length

finalize(*model*, *tau*)

Called in the last phase of the calculation, after the optical depth has be completely computed.

classmethod input_keywords()

property name

Name of the contribution. Identifier for plots

property order

Computational order. Lower numbers are given higher priority and are computed first.

Returns Order of computation

Return type int

prepare(*model*, *wngrid*)

Used to prepare the contribution for the calculation. Called before the forward model performs the main optical depth calculation. Default behaviour is to loop through *prepare_each()* and sum all results into a single cross-section.

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

prepare_each (*model*, *wngrid*)

Requires implementation

Used to prepare each component of the contribution. For context when the main taurex program is run with the option each spectra is the component for the contribution. For cross-section based contributions, the components are each molecule Should yield the name of the component and the component itself

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

Yields **component** (tuple of type (str, array)) – Name of component and component itself

property sigma

write (*output*)

Writes contribution class and arguments to file

Parameters **output** (*Output*) –

contribute_tau

Generic cross-section integration function for tau, numba-fied for performance.

This has the form:

$$\tau_\lambda(z) = \int_{z_0}^{z_1} \sigma(z')\rho(z')dz',$$

where z is the layer, z_0 and z_1 are startK and endK respectively. σ is the weighted cross-section sigma. rho is the density and dz' is the integration path length path

Parameters

- **startK** (*int*) – starting layer in integration
- **endK** (*int*) – last layer in integration
- **density_offset** (*int*) – Which part of the density profile to start from
- **sigma** (array) – cross-section
- **density** (*array_like*) – density profile of atmosphere
- **path** (*array_like*) – path-length or altitude gradient
- **nlayers** (*int*) – Total number of layers (unused)
- **ngrid** (*int*) – total number of grid points
- **layer** (*int*) – Which layer we currently on

Returns **tau** – optical depth (well almost you still need to do exp(-tau) yourself)

Return type array_like

10.7.2 Absorption

class AbsorptionContribution

Bases: *taurex.contributions.contribution.Contribution*

build(*model*)
Called during forward model build phase Does nothing by default

Parameters **model** (*ForwardModel*) – Forward model

contribute(*model*, *start_horz_layer*, *end_horz_layer*, *density_offset*, *layer*, *density*, *tau*, *path_length=None*)
Computes an integral for a single layer for the optical depth.

Parameters

- **model** (*ForwardModel*) – A forward model
- **start_layer** (*int*) – Lowest layer limit for integration
- **end_layer** (*int*) – Upper layer limit of integration
- **density_offset** (*int*) – offset in density layer
- **layer** (*int*) – atmospheric layer being computed
- **density** (array) – density profile of atmosphere
- **tau** (array) – optical depth to store result
- **path_length** (array) – integration length

finalize(*model*)
Called in the last phase of the calculation, after the optical depth has be completely computed.

classmethod input_keywords()

prepare(*model*, *wngrid*)
Used to prepare the contribution for the calculation. Called before the forward model performs the main optical depth calculation. Default behaviour is to loop through *prepare_each()* and sum all results into a single cross-section.

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

prepare_each(*model*, *wngrid*)
Requires implementation

Used to prepare each component of the contribution. For context when the main taurex program is run with the option each spectra is the component for the contribution. For cross-section based contributions, the components are each molecule Should yield the name of the component and the component itself

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

Yields **component** (tuple of type (str, array)) – Name of component and component itself

property sigma

10.7.3 CIA

class CIAContribution(*cia_pairs=None*)
Bases: *taurex.contributions.contribution.Contribution*

Computes the contribution to the optical depth occuring from collisionally induced absorption.

Parameters **cia_pairs** (list of str) – list of molecule pairs of the form mol1-mol2 e.g. H2-He

property ciaPairs

Returns list of molecular pairs involved

Returns

Return type list of str

contribute(*model*, *start_layer*, *end_layer*, *density_offset*, *layer*, *density*, *tau*, *path_length=None*)

Computes an integral for a single layer for the optical depth.

Parameters

- **model** (*ForwardModel*) – A forward model
- **start_layer** (*int*) – Lowest layer limit for integration
- **end_layer** (*int*) – Upper layer limit of integration
- **density_offset** (*int*) – offset in density layer
- **layer** (*int*) – atmospheric layer being computed
- **density** (array) – density profile of atmosphere
- **tau** (array) – optical depth to store result
- **path_length** (array) – integration length

classmethod input_keywords()

prepare_each(*model*, *wngrid*)

Computes and weighs cross-section for a single pair of molecules

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

Yields **component** (tuple of type (str, array)) – Molecular pair and the weighted cia opacity.

write(*output*)

Writes contribution class and arguments to file

Parameters **output** (*Output*) –

contribute_cia

Collisionally induced absorption integration function

This has the form:

$$\tau_\lambda(z) = \int_{z_0}^{z_1} \sigma(z')\rho(z')^2 dz',$$

where z is the layer, z_0 and z_1 are startK and endK respectively. σ is the weighted cross-section sigma. rho is the density and dz' is the integration path length path

Parameters

- **startK** (*int*) – starting layer in integration
- **endK** (*int*) – last layer in integration
- **density_offset** (*int*) – Which part of the density profile to start from
- **sigma** (array) – cross-section

- **density** (*array_like*) – density profile of atmosphere
- **path** (*array_like*) – path-length or altitude gradient
- **nlayers** (*int*) – Total number of layers (unused)
- **ngrid** (*int*) – total number of grid points
- **layer** (*int*) – Which layer we currently on

Returns **tau** – optical depth (well almost you still need to do `exp(-tau)` yourself)

Return type array_like

10.7.4 Rayleigh

class RayleighContribution

Bases: *taurex.contributions.contribution.Contribution*

Computes contribution from Rayleigh scattering

BIBTEX_ENTRIES = ['\n @book{cox_allen_rayleigh,\n title={Allen's astrophysical quantit

classmethod input_keywords()

prepare_each(*model*, *wngrid*)

Computes the weighted opacity due to rayleigh scattering for any possible molecules within atmosphere.

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (`array`) – Wavenumber grid

Yields **component** (`tuple` of type (str, `array`)) – Name of scattering molecule and the weighted rayeligh opacity.

10.7.5 SimpleClouds

class SimpleCloudsContribution(*clouds_pressure=1000.0*)

Bases: *taurex.contributions.contribution.Contribution*

Optically thick cloud deck up to a certain height

These have the form:

$$\tau(\lambda, z) = \begin{cases} \infty & \text{if } P(z) >= P_0 \\ 0 & \text{if } P(z) < P_0 \end{cases}$$

Where P_0 is the pressure at the top of the cloud-deck

Parameters **clouds_pressure** (*float*) – Pressure at top of cloud deck

property cloudsPressure

Cloud top pressure in Pascal

contribute(*model*, *start_layer*, *end_layer*, *density_offset*, *layer*, *density*, *tau*, *path_length=None*)

Computes an integral for a single layer for the optical depth.

Parameters

- **model** (*ForwardModel*) – A forward model

- **start_layer** (*int*) – Lowest layer limit for integration
- **end_layer** (*int*) – Upper layer limit of integration
- **density_offset** (*int*) – offset in density layer
- **layer** (*int*) – atmospheric layer being computed
- **density** (array) – density profile of atmosphere
- **tau** (array) – optical depth to store result
- **path_length** (array) – integration length

classmethod input_keywords()

property order

Computational order. Lower numbers are given higher priority and are computed first.

Returns Order of computation

Return type int

prepare_each(*model*, *wngrid*)

Returns an absorbing cross-section that is infinitely absorping up to a certain height

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

Yields **component** (tuple of type (str, array)) – Clouds and opacity array.

write(*output*)

Writes contribution class and arguments to file

Parameters **output** (*Output*) –

10.7.6 Mie Scattering (BH)

Warning: This is no longer available in the base TauREx 3 since version 3.1. To use this you must install the taurex_bhmie plugin.

10.7.7 Mie Scattering (Lee)

class LeeMieContribution(*lee_mie_radius=0.01*, *lee_mie_q=40*, *lee_mie_mix_ratio=1e-10*, *lee_mie_bottomP=-1*, *lee_mie_topP=-1*)

Bases: *taurex.contributions.contribution.Contribution*

Computes Mie scattering contribution to optica depth Formalism taken from: Lee et al. 2013, ApJ, 778, 97

Parameters

- **lee_mie_radius** (*float*) – Particle radius in um
- **lee_mie_q** (*float*) – Extinction coefficient
- **lee_mie_mix_ratio** (*float*) – Mixing ratio in atmosphere
- **lee_mie_bottomP** (*float*) – Bottom of cloud deck in Pa
- **lee_mie_topP** (*float*) – Top of cloud deck in Pa

BIBTEX_ENTRIES = ['\n @article{Lee_2013,\n doi = {10.1088/0004-637x/778/2/97},\n url =

classmethod input_keywords()

property mieBottomPressure
Pressure at bottom of cloud deck in Pa

property mieMixing
Mixing ratio in atmosphere

property mieQ
Extinction coefficient

property mieRadius
Particle radius in um

property mieTopPressure
Pressure at top of cloud deck in Pa

prepare_each(*model*, *wngrid*)
Computes and weights the mie opacity for the pressure regions given

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

Yields **component** (tuple of type (str, array)) – Lee and the weighted mie opacity.

write(*output*)
Writes contribution class and arguments to file

Parameters **output** (*Output*) –

10.7.8 Mie Scattering (Flat)

class FlatMieContribution(*flat_mix_ratio=1e-10*, *flat_bottomP=-1*, *flat_topP=-1*)
Bases: *taurex.contributions.contribution.Contribution*

Computes a flat absorption contribution across all wavelengths to the optical depth

Parameters

- **flat_mix_ratio** (*float*) – Opacity value
- **flat_bottomP** (*float*) – Bottom of absorbing region in Pa
- **flat_topP** (*float*) – Top of absorbing region in Pa

classmethod input_keywords()

property mieBottomPressure
Pressure at bottom of absorbing region in Pa

property mieMixing
Opacity of absorbing region in m2

property mieTopPressure
Pressure at top of absorbing region in Pa

prepare_each(*model*, *wngrid*)
Computes and flat absorbing opacity for the pressure regions given

Parameters

- **model** (*ForwardModel*) – Forward model
- **wngrid** (array) – Wavenumber grid

Yields **component** (tuple of type (str, array)) – Flat and the weighted mie opacity.

write(*output*)
Writes contribution class and arguments to file

Parameters **output** (*Output*) –

10.8 Chemistry Models (taurex.chemistry)

10.8.1 Base

class Chemistry(*name*)
Bases: *taurex.data.fittable.Fittable*, *taurex.log.logger.Logger*, *taurex.output.writeable.Writeable*, *taurex.data.citation.Citable*

Abstract Class

Skeleton for defining chemistry. Must implement methods:

- *activeGases()*
- *inactiveGases()*
- *activeGasMixProfile()*
- *inactiveGasMixProfile()*

Active are those that are actively absorbing in the atmosphere. In technical terms they are molecules that have absorption cross-sections. You can see which molecules are able to actively absorb by doing: You can find out what molecules can actively absorb by doing:

```
>>> avail_active_mols = OpacityCache().find_list_of_molecules()
```

Parameters **name** (*str*) – Name used in logging

property activeGasMixProfile
Requires implementation

Should return profiles of shape (nactivegases,nlayers). Active refers to gases that are actively absorbing in the atmosphere. Another way to put it these are gases where molecular cross-sections are used.

property activeGases
Requires implementation

Should return a list of molecule names

Returns **active** – List of active gases

Return type list

property availableActive
Returns a list of available actively absorbing molecules

Returns **molecules** – Actively absorbing molecules

Return type list

compute_mu_profile(*nlayers*)
Computes molecular weight of atmosphere for each layer

Parameters **nlayers** (*int*) – Number of layers

property condensateMixProfile
Requires implementation

Should return profiles of shape `(ncondensates,nlayers)`.

property condensates
Returns a list of condensates in the atmosphere.

Returns **active** – List of condensates

Return type `list`

property gases

get_condensate_mix_profile(*condensate_name*)
Returns the mix profile of a particular condensate

Parameters **condensate_name** (*str*) – Name of condensate

Returns **mixprofile** – Mix profile of condensate with shape `(nlayer)`

Return type `array`

get_gas_mix_profile(*gas_name*)
Returns the mix profile of a particular gas

Parameters **gas_name** (*str*) – Name of gas

Returns **mixprofile** – Mix profile of gas with shape `(nlayer)`

Return type `array`

get_molecular_mass(*molecule*)

property hasCondensates

property inactiveGasMixProfile
Requires implementation

Should return profiles of shape `(ninactivegases,nlayers)`.

property inactiveGases
Requires implementation

Should return a list of molecule names

Returns **inactive** – List of inactive gases

Return type `list`

initialize_chemistry(*nlayers=100*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)
Requires implementation

Derived classes should implement this to compute the active and inactive gas profiles

Parameters

- **nlayers** (*int*) – Number of layers in atmosphere
- **temperature_profile** (`array`) – Temperature profile in K, must have length `nlayers`
- **pressure_profile** (`array`) – Pressure profile in Pa, must have length `nlayers`

- **altitude_profile** (array) – Altitude profile in m, must have length nlayers

property mixProfile

property mu

property muProfile
Molecular weight for each layer of atmosphere

Returns mix_profile

Return type array

set_star_planet(*star*, *planet*)
Supplies the star and planet to chemistry for photochemistry reasons. Does nothing by default

Parameters

- **star** (*Star*) – A star object
- **planet** (Planet) – A planet object

write(*output*)
Writes chemistry class and arguments to file

Parameters **output** (*Output*) –

10.8.2 Base (Auto)

class AutoChemistry(*name*)
Bases: *taurex.data.profiles.chemistry.chemistry.Chemistry*

Chemistry class that automatically seperates out active and inactive gases

Has a helper function that should be called when

Parameters **name** (*str*) – Name of class

property activeGasMixProfile
Active gas layer by layer mix profile

Returns active_mix_profile

Return type array

property activeGases
Requires implementation

Should return a list of molecule names

Returns **active** – List of active gases

Return type list

compute_mu_profile(*nlayers*)
Computes molecular weight of atmosphere for each layer

Parameters **nlayers** (*int*) – Number of layers

determine_active_inactive()

property gases

property inactiveGasMixProfile
Inactive gas layer by layer mix profile

Returns inactive_mix_profile

Return type array

property inactiveGases

Requires implementation

Should return a list of molecule names

Returns inactive – List of inactive gases

Return type list

property mixProfile

10.8.3 Equilibrium Chemistry (ACE)

Warning: This is no longer available in the base TauREx 3 since version 3.1. To use this you must install the taurex_ace plugin.

10.8.4 Free chemistry

exception InvalidChemistryException

Bases: taurex.exceptions.InvalidModelException

Exception that is called when atmosphere mix is greater than unity

class TaurexChemistry(*fill_gases=['H2', 'He'], ratio=0.17567, derived_ratios=[], base_metallicty=0.013*)

Bases: *taurex.data.profiles.chemistry.autochemistry.AutoChemistry*

The standard chemical model used in Taurex. This allows for the combination of different mixing profiles for each molecule. Lets take an example profile, we want an atmosphere with a constant mixing of H2O but two layer mixing for CH4. First we initialize our chemical model:

```
>>> chemistry = TaurexChemistry()
```

Then we can add our molecules using the *addGas()* method. Lets start with H2O, since its a constant profile for all layers of the atmosphere we thus add the *ConstantGas* object:

```
>>> chemistry.addGas(ConstantGas('H2O',mix_ratio = 1e-4))
```

Easy right? Now the same goes for CH4, we can add the molecule into the chemical model by using the correct profile (in this case *TwoLayerGas*):

```
>>> chemistry.addGas(TwoLayerGas('CH4',mix_ratio_surface=1e-4,
                                  mix_ratio_top=1e-8))
```

Molecular profiles available are:

- *ConstantGas*
- *TwoLayerGas*
- TwoPointGas

Parameters

- **fill_gases** (str or list) – Either a single gas or list of gases to fill the atmosphere with

- **ratio** (float or `list`) – If a bunch of molecules are used to fill an atmosphere, whats the ratio between them? The first fill gas is considered the main one with others defined as `molecule / main_molecule`

addGas(*gas*)
Adds a gas in the atmosphere.

Parameters **gas** (*Gas*) – Gas to add into the atmosphere. Only takes effect on next initialization call.

citations()

compute_elements_mix()

fill_atmosphere(*mixratio_remainder*)

fitting_parameters()
Overrides the fitting parameters to return one with all the gas profile parameters as well

Returns **fit_param**

Return type `dict`

property gases

get_element_ratio(*elem_ratio*)

get_metallicity()

initialize_chemistry(*nlayers=100*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)
Initializes the chemical model and computes the all gas profiles and the mu profile for the forward model

classmethod input_keywords()

isActive(*gas*)
Determines if the gas is active or not (Whether we have cross-sections)

Parameters **gas** (*str*) – Name of molecule

Returns True if active

Return type bool

property metallicity

property mixProfile

setup_derived_params(*ratio_list*)

setup_fill_params()

write(*output*)
Writes chemistry class and arguments to file

Parameters **output** (*Output*) –

10.9 Gas Models (`taurex.chemistry`)

10.9.1 Base

class Gas(*name*, *molecule_name*)
Bases: *taurex.data.fittable.Fittable*, *taurex.log.logger.Logger*, *taurex.output.*

writeable.Writeable, *taurex.data.citation.Citable*

Abstract Class

This class is a base for a single molecule or gas. Its used to describe how it mixes at each layer and combined with `TaurexChemistry` is used to build a chemical profile of the planets atmosphere. Requires implementation of:

- func:*~mixProfile*

Parameters

- **name** (*str*) – Name used in logging
- **molecule_name** (*str*) – Name of molecule

initialize_profile(*nlayers=None*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)

Initializes and computes mix profile

Parameters

- **nlayers** (*int*) – Number of layers in atmosphere
- **temperature_profile** (`array`) – Temperature profile of atmosphere in K. Length must be equal to `nlayers`
- **pressure_profile** (`array`) – Pressure profile of atmosphere in Pa. Length must be equal to `nlayers`
- **altitude_profile** (`array`) – Altitude profile of atmosphere in m. Length must be equal to `nlayers`

property mixProfile

Requires implementation

Should return mix profile of molecule/gas at each layer

Returns **mix** – Mix ratio for molecule at each layer

Return type `array`

property molecule

returns: **molecule_name** – Name of molecule :rtype: str

write(*output*)

Writes class and arguments to file

Parameters **output** (*Output*) –

10.9.2 Constant

class ConstantGas(*molecule_name='H2O'*, *mix_ratio=1e-05*)

Bases: *taurex.data.profiles.chemistry.gas.gas.Gas*

Constant gas profile. Molecular abundace is constant at each layer of the atmosphere

Parameters

- **molecule_name** (*str*) – Name of molecule
- **mix_ratio** (*float*) – Mixing ratio of the molecule

add_active_gas_param()
Adds the mixing ratio as a fitting parameter as the name of the molecule

initialize_profile(*nlayers=None*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)
Initializes and computes mix profile

Parameters

- **nlayers** (*int*) – Number of layers in atmosphere
- **temperature_profile** (array) – Temperature profile of atmosphere in K. Length must be equal to nlayers
- **pressure_profile** (array) – Pressure profile of atmosphere in Pa. Length must be equal to nlayers
- **altitude_profile** (array) – Altitude profile of atmosphere in m. Length must be equal to nlayers

classmethod input_keywords()

property mixProfile
Mixing profile

Returns **mix** – Mix ratio for molecule at each layer

Return type array

write(*output*)
Writes class and arguments to file

Parameters **output** (*Output*) –

10.9.3 Two Layer

class TwoLayerGas(*molecule_name='CH4'*, *mix_ratio_surface=0.0001*, *mix_ratio_top=1e-08*, *mix_ratio_P=1000.0*, *mix_ratio_smoothing=10*)
Bases: *taurex.data.profiles.chemistry.gas.gas.Gas*

Two layer gas profile.

A gas profile with two different mixing layers at the surface of the planet and top of the atmosphere seperated at a defined pressure point and smoothened.

Parameters

- **molecule_name** (*str*) – Name of molecule
- **mix_ratio_surface** (*float*) – Mixing ratio of the molecule on the planet surface
- **mix_ratio_top** (*float*) – Mixing ratio of the molecule at the top of the atmosphere
- **mix_ratio_P** (*float*) – Boundary Pressure point between the two layers
- **mix_ratio_smoothing** (*float , optional*) – smoothing window

BIBTEX_ENTRIES = ['\n @misc{changeat2019complex,\n title={Towards a more complex descr

add_P_param()
Generates pressure fitting parameter. Has the form 'Moleculename_P'

add_surface_param()
Generates surface fitting parameters. Has the form ''Moleculename_surface'

add_top_param()
Generates TOA fitting parameters. Has the form: 'Moleculename_top'

initialize_profile(*nlayers=None*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)
Initializes and computes mix profile

Parameters

- **nlayers** (*int*) – Number of layers in atmosphere
- **temperature_profile** (array) – Temperature profile of atmosphere in K. Length must be equal to nlayers
- **pressure_profile** (array) – Pressure profile of atmosphere in Pa. Length must be equal to nlayers
- **altitude_profile** (array) – Altitude profile of atmosphere in m. Length must be equal to nlayers

classmethod input_keywords()

property mixProfile
returns: **mix** – Mix ratio for molecule at each layer :rtype: array

property mixRatioPressure

property mixRatioSmoothing

property mixRatioSurface
Abundance on the planets surface

property mixRatioTop
Abundance on the top of atmosphere

write(*output*)
Writes class and arguments to file

Parameters **output** (*Output*) –

10.9.4 Array

class TwoLayerGas(*molecule_name='CH4'*, *mix_ratio_surface=0.0001*, *mix_ratio_top=1e-08*, *mix_ratio_P=1000.0*, *mix_ratio_smoothing=10*)
Bases: *taurex.data.profiles.chemistry.gas.gas.Gas*

Two layer gas profile.

A gas profile with two different mixing layers at the surface of the planet and top of the atmosphere seperated at a defined pressure point and smoothened.

Parameters

- **molecule_name** (*str*) – Name of molecule
- **mix_ratio_surface** (*float*) – Mixing ratio of the molecule on the planet surface
- **mix_ratio_top** (*float*) – Mixing ratio of the molecule at the top of the atmosphere
- **mix_ratio_P** (*float*) – Boundary Pressure point between the two layers
- **mix_ratio_smoothing** (*float , optional*) – smoothing window

BIBTEX_ENTRIES = ['\n @misc{changeat2019complex,\n title={Towards a more complex descr

add_P_param()
Generates pressure fitting parameter. Has the form 'Moleculename_P'

add_surface_param()
Generates surface fitting parameters. Has the form ''Moleculename_surface'

add_top_param()
Generates TOA fitting parameters. Has the form: 'Moleculename_top'

initialize_profile(*nlayers=None*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)
Initializes and computes mix profile

Parameters

- **nlayers** (*int*) – Number of layers in atmosphere
- **temperature_profile** (array) – Temperature profile of atmosphere in K. Length must be equal to nlayers
- **pressure_profile** (array) – Pressure profile of atmosphere in Pa. Length must be equal to nlayers
- **altitude_profile** (array) – Altitude profile of atmosphere in m. Length must be equal to nlayers

classmethod input_keywords()

property mixProfile
returns: **mix** – Mix ratio for molecule at each layer :rtype: array

property mixRatioPressure

property mixRatioSmoothing

property mixRatioSurface
Abundance on the planets surface

property mixRatioTop
Abundance on the top of atmosphere

write(*output*)
Writes class and arguments to file

Parameters **output** (*Output*) –

10.10 Temperature (taurex.temperature)

10.10.1 Base

class TemperatureProfile(*name*)
Bases: *taurex.data.fittable.Fittable*, *taurex.log.logger.Logger*, *taurex.output.writeable.Writeable*, *taurex.data.citation.Citable*

Abstract Class

Defines temperature profile for an atmosphere

Must define:

- *profile()*

Parameters **name** (*str*) – Name used in logging

property averageTemperature

initialize_profile(*planet=None*, *nlayers=100*, *pressure_profile=None*)
Initializes the profile

Parameters

- **planet** (Planet) –
- **nlayers** (*int*) – Number of layers in atmosphere
- **pressure_profile** (array) – Pressure at each layer of the atmosphere

classmethod input_keywords()
Return all input keywords

property profile
Must return a temperature profile at each layer of the atmosphere

Returns **temperature** – Temperature in Kelvin

Return type array

write(*output*)

10.10.2 Isothermal

class Isothermal(*T=1500*)
Bases: *taurex.data.profiles.temperature.tprofile.TemperatureProfile*

An isothermal temperature-pressure profile

Parameters **T** (*float*) – Isothermal Temperature to set

classmethod input_keywords()
Return all input keywords

property isoTemperature
Isothermal temperature in Kelvin

property profile
Returns an isothermal temperature profile

Returns: **array** temperature profile

write(*output*)

10.10.3 Two-stream approximation (Guillot)

class Guillot2010(*T_irr=1500*, *kappa_irr=0.01*, *kappa_v1=0.005*, *kappa_v2=0.005*, *alpha=0.5*, *T_int=100*)
Bases: *taurex.data.profiles.temperature.tprofile.TemperatureProfile*

TP profile from Guillot 2010, A&A, 520, A27 (equation 49) Using modified 2stream approx. from Line et al. 2012, ApJ, 749,93 (equation 19)

Parameters

- **T_irr** (*float*) – planet equilibrium temperature (Line fixes this but we keep as free parameter)

- **kappa_ir** (*float*) – mean infra-red opacity
- **kappa_v1** (*float*) – mean optical opacity one
- **kappa_v2** (*float*) – mean optical opacity two
- **alpha** (*float*) – ratio between kappa_v1 and kappa_v2 downwards radiation stream
- **T_int** (*float*) – Internal heating parameter (K)

BIBTEX_ENTRIES = ['\n @article{guillot,\n author = {{Guillot, T.}},\n title = {On the

property equilTemperature
Planet equilibrium temperature

classmethod input_keywords()
Return all input keywords

property internalTemperature
ratio between kappa_v1 and kappa_v2

property meanInfraOpacity
mean infra-red opacity

property meanOpticalOpacity1
mean optical opacity one

property meanOpticalOpacity2
mean optical opacity two

property opticalRatio
ratio between kappa_v1 and kappa_v2

property profile
Returns a guillot temperature temperature profile

Returns temperature_profile

Return type :obj:np.array(float)

write(*output*)

10.10.4 Multi Point

exception InvalidTemperatureException
Bases: `taurex.exceptions.InvalidModelException`

Exception that is called when atmosphere mix is greater than unity

class NPoint(*T_surface=1500.0*, *T_top=200.0*, *P_surface=None*, *P_top=None*, *temperature_points=[]*, *pressure_points=[]*, *smoothing_window=10*, *limit_slope=9999999*)
Bases: *taurex.data.profiles.temperature.tprofile.TemperatureProfile*

A temperature profile that is defined at various heights of the atmopshere and then smoothend.

At minimum, temepratures on both the top `T_top` and surface `T_surface` must be defined. If any intermediate points are given as `temperature_points` then the same number of `pressure_points` must be given as well.

A 2-point temperature profile has `len(temperature_points) == 0` A 3-point temperature profile has `len(temperature_points) == 1`

etc.

Parameters

- **T_surface** (*float*) – Temperature at the planets surface in Kelvin
- **T_top** (*float*) – Temperature at the top of the atmosphere in Kelvin
- **P_surface** (*float , optional*) – Pressure for T_surface (Optional) otherwise uses surface pressure from forward model
- **P_top** (*float , optional*) – Pressure for T_top (Optional) otherwise uses top pressure from forward model
- **temperature_points** (list) – temperature points between T_top and T_surface
- **pressure_points** (list) – pressure points that the each temperature in temperature_points lie on
- **smoothing_window** (*int*) – smoothing window
- **limit_slope** (*int*) –

check_profile(*Ppt*, *Tpt*)

generate_pressure_fitting_params()
Generates the fitting parameters for the pressure points These are given the name P_point(number) for example, if two extra pressure points are defined between the top and surface then the fitting parameters generated are P_point0 and P_point1

generate_temperature_fitting_params()
Generates the fitting parameters for the temeprature points These are given the name T_point(number) for example, if two extra temeprature points are defined between the top and surface then the fitting parameters generated are T_point0 and T_point1

classmethod input_keywords()
Return all input keywords

property pressureSurface
None

property pressureTop
None

property profile
Must return a temperature profile at each layer of the atmosphere

Returns **temperature** – Temperature in Kelvin

Return type array

property temperatureSurface
Temperature at planet surface in Kelvin

property temperatureTop
Temperature at top of atmosphere in Kelvin

write(*output*)

10.10.5 Rodgers

class Rodgers2000(*temperature_layers=[]*, *correlation_length=5.0*, *covariance_matrix=None*)
Bases: *taurex.data.profiles.temperature.tprofile.TemperatureProfile*

Layer-by-layer temperature - pressure profile retrieval using dampening factor Introduced in Rodgers (2000): Inverse Methods for Atmospheric Sounding (equation 3.26). Featured in NEMESIS code (Irwin et al., 2008, J. Quant. Spec., 109, 1136 (equation 19) Used in all Barstow et al. papers.

Parameters

- **temperature_layers** (list) – Temperature in Kelvin per layer of pressure
- **correlation_length** (*float*) – In scaleheights, Line et al. 2013 sets this to 7, Irwin et al sets this to 1.5 may be left as free and Pressure dependent parameter later.
- **covariance_matrix** (array , optional) – User can supply their own covaraince matrix

BIBTEX_ENTRIES = ['\n @MISC{rodger_retrievals,\n author = {{Rodgers}, Clive D.},\n tit

correlate_temp(*cov_mat*)

property correlationLength
Correlation length in scale heights

gen_covariance()
Generate the covariance matrix if None is supplied

generate_temperature_fitting_params()
Generates the temperature fitting parameters for each layer of the atmosphere For a 4 layer atmosphere the fitting parameters generated are T_0, T_1, T_2 and T_3

classmethod input_keywords()
Return all input keywords

property profile
Must return a temperature profile at each layer of the atmosphere

Returns **temperature** – Temperature in Kelvin

Return type array

write(*output*)

10.10.6 Array

class TemperatureArray(*tp_array=[2000, 1000], p_points=None, reverse=False*)
Bases: *taurex.data.profiles.temperature.tprofile.TemperatureProfile*

Temperature profile loaded from array

classmethod input_keywords()
Return all input keywords

property profile
Returns an isothermal temperature profile

Returns: :obj:np.array(float) temperature profile

write(*output*)

10.10.7 File

class TemperatureFile(*filename=None, skiprows=0, temp_col=0, press_col=None, temp_units='K', press_units='Pa', delimiter=None, reverse=False*)
Bases: *taurex.data.profiles.temperature.temparray.TemperatureArray*

A temperature profile read from file

Parameters **filename** (*str*) – File name for temperature profile

classmethod input_keywords()
Return all input keywords

10.11 Pressure Modules (taurex.pressure)

10.11.1 Base

class PressureProfile(*name*, *nlayers*)
Bases: *taurex.data.fittable.Fittable*, *taurex.log.logger.Logger*, *taurex.output.writeable.Writeable*, *taurex.data.citation.Citable*

Abstract Class

Base pressure class. Simple. Defines the layering of the atmosphere. Only requires implementation of:

- *compute_pressure_profile()*
- *profile()*

Parameters

- **name** (*str*) – Name used in logging
- **nlayers** (*int*) – Number of layers in atmosphere

compute_pressure_profile()
Requires implementation

Compute pressure profile and generate pressure array in Pa

Returns **pressure_profile** – Pressure profile array in Pa

Return type array

property nLayers
Number of layers

Returns

Return type int

property nLevels

property profile
Returns pressure at each atmospheric layer (Pascal)

Returns **pressure_profile**

Return type array

write(*output*)

10.11.2 Simple

class SimplePressureProfile(*nlayers=100*, *atm_min_pressure=0.0001*, *atm_max_pressure=1000000.0*)
Bases: *taurex.data.profiles.pressure.pressureprofile.PressureProfile*

A basic pressure profile.

Parameters

- **nlayers** (*int*) – Number of layers in atmosphere
- **atm_min_pressure** (*float*) – minimum pressure in Pascal (top of atmosphere)
- **atm_max_pressure** (*float*) – maximum pressure in Pascal (surface of planet)

compute_pressure_profile()
Sets up the pressure profile for the atmosphere model

classmethod input_keywords()

property maxAtmospherePressure
Maximum pressure of atmosphere (surface) in Pascal

property minAtmospherePressure
Minimum pressure of atmosphere (top layer) in Pascal

property profile
Returns pressure at each atmospheric layer (Pascal)

Returns pressure_profile

Return type array

write(*output*)

10.12 Stellar Models (taurex.stellar)

10.12.1 Base

class Star(*temperature=5000*, *radius=1.0*, *distance=1*, *magnitudeK=10.0*, *mass=1.0*, *metallicity=1.0*)
Bases: *taurex.data.fittable.Fittable*, *taurex.log.logger.Logger*, *taurex.output.writeable.Writeable*, *taurex.data.citation.Citable*

A base class that holds information on the star in the model. Its implementation is a star that has a blackbody spectrum.

Parameters

- **temperature** (*float, optional*) – Stellar temperature in Kelvin
- **radius** (*float, optional*) – Stellar radius in Solar radius
- **metallicity** (*float, optional*) – Metallicity in solar values
- **mass** (*float, optional*) – Stellar mass in solar mass
- **distance** (*float, optional*) – Distance from Earth in pc
- **magnitudeK** (*float, optional*) – Maginitude in K band

initialize(*wngrid*)
Initializes the blackbody spectrum on the given wavenumber grid

Parameters **wngrid** (array) – Wavenumber grid cm-1 to compute black body spectrum

classmethod input_keywords()

property mass

property radius
Radius in metres

Returns R

Return type float

property spectralEmissionDensity
Spectral emmision density

Returns sed

Return type array

property temperature
Blackbody temperature in Kelvin

Returns T

Return type float

write(*output*)

10.12.2 Blackbody

class BlackbodyStar(*temperature=5000*, *radius=1.0*, *distance=1*, *magnitudeK=10.0*, *mass=1.0*, *metallicity=1.0*)
Bases: *taurex.data.stellar.star.Star*

Alias for the base star type

classmethod input_keywords()

10.12.3 PHOENIX

class PhoenixStar(*temperature=5000*, *radius=1.0*, *metallicity=1.0*, *mass=1.0*, *distance=1*, *magnitudeK=10.0*, *phoenix_path=None*, *retro_version_file=None*)
Bases: *taurex.data.stellar.star.BlackbodyStar*

A star that uses the PHOENIX synthetic stellar atmosphere spectrums.

These spectrums are read from `.gits.gz` files in a directory given by `phoenix_path` Each file must contain the spectrum for one temperature

Parameters

- **phoenix_path** (str, **required**) – Path to folder containing phoenix `fits.gz` files
- **temperature** (*float, optional*) – Stellar temperature in Kelvin
- **radius** (*float, optional*) – Stellar radius in Solar radius
- **metallicity** (*float, optional*) – Metallicity in solar values
- **mass** (*float, optional*) – Stellar mass in solar mass
- **distance** (*float, optional*) – Distance from Earth in pc
- **magnitudeK** (*float, optional*) – Maginitude in K band

Raises **Exception** – Raised when no phoenix path is defined

BIBTEX_ENTRIES = ['\n @article{ refId0,\n author = {{Husser, T.-O.} and {Wende-von Ber

compute_logg()
Computes log(surface_G)

find_nearest_file()

get_avail_phoenix()

initialize(*wngrid*)
Initializes and interpolates the spectral emission density to the current stellar temperature and given wavenumber grid

Parameters **wngrid** (array) – Wavenumber grid to interpolate the SED to

classmethod input_keywords()

property mass
Mass of star in solar mass

Returns M

Return type float

read_spectra(*p_file*)

recompute_spectra()

property spectralEmissionDensity
Spectral emmision density

Returns sed

Return type array

property temperature
Effective Temperature in Kelvin

Returns T

Return type float

write(*output*)

10.13 Instruments (taurex.instruments)

10.13.1 Base

class Instrument
Bases: *taurex.log.logger.Logger*, *taurex.data.citation.Citable*

Abstract class

Defines some method that transforms a spectrum and generates noise.

classmethod input_keywords()

model_noise(*model*, *model_res=None*, *num_observations=1*)
Requires implementation

For a given forward model (and optional result) Resample the spectrum and compute noise profile.

Parameters

- **model** (*ForwardModel*) – Forward model to pass.
- **model_res** (tuple, optional) – Result from *model()*
- **num_observations** (*int, optional*) – Number of observations to simulate

10.13.2 Signal-to-Noise

class SNRInstrument(*SNR=10*, *binner=None*)
Bases: *taurex.instruments.instrument.Instrument*

Simple instrument model that, for a given wavelength-independant, signal-to-noise ratio, compute resulting noise from it.

Parameters

- **SNR** (*float*) – Signal-to-noise ratio
- **binner** (*Binner*, optional) – Optional resampler to generate a new spectral grid.

classmethod input_keywords()

model_noise(*model*, *model_res=None*, *num_observations=1*)
Requires implementation

For a given forward model (and optional result) Resample the spectrum and compute noise profile.

Parameters

- **model** (*ForwardModel*) – Forward model to pass.
- **model_res** (tuple, optional) – Result from *model()*
- **num_observations** (*int, optional*) – Number of observations to simulate

10.14 Observations (taurex.spectrum)

10.14.1 Base

Contains the basic definition of an observed spectrum for TauRex 3

class BaseSpectrum(*name*)
Bases: *taurex.log.logger.Logger*, *taurex.data.fittable.Fittable*, *taurex.output.writeable.Writeable*

Abstract class

A base class where spectrums are loaded (or later created). This is used to either plot against the forward model or passed into the optimizer to be used to fit the forward model

Parameters **name** (*str*) – Name to be used in logging

property binEdges
Requires Implementation

Should return the bin edges of the wavenumber grid

Raises **NotImplementedError** –

property binWidths
Requires Implementation

Should return the widths of each bin in the wavenumber grid

Raises **NotImplementedError** –

create_binner()
Creates the appropriate binning object

property derivedParameters

property errorBar
Requires Implementation

Should return the error. *Must* be the same shape as *spectrum()*

Raises **NotImplementedError** –

property fittingParameters

classmethod input_keywords()

property rawData
Requires Implementation

Should return the raw data set.

Raises **NotImplementedError** –

property spectrum
Requires Implementation

Should return the observed spectrum.

Raises **NotImplementedError** –

property wavelengthGrid
Requires Implementation

Should return the wavelength grid of the spectrum in microns. This does not need to necessarily match the shape of *spectrum()*

Raises **NotImplementedError** –

property wavenumberGrid
Wavenumber grid in cm-1

Returns **wngrid**

Return type array

write(*output*)

10.14.2 Array

class ArraySpectrum(*spectrum=None*)
Bases: *taurex.data.spectrum.spectrum.BaseSpectrum*

Loads an observed spectrum from an array and computes bin edges and bin widths. Spectrum shape(nbins, 3-4) with 3-4 columns with ordering:

1. wavelength (um)
2. spectral data
3. error
4. (optional) bin width

If no bin width is present then they are computed.

Parameters **filename** (*string*) – Path to observed spectrum file.

property binEdges
Bin edges

property binWidths
bin widths

property errorBar
Error bars for the spectrum

classmethod input_keywords()

manual_binning()

property rawData
Data read from file

property spectrum
The spectrum itself

property wavelengthGrid
Wavelength grid in microns

property wavenumberGrid
Wavenumber grid in cm-1

10.14.3 Observed

class ObservedSpectrum(*filename=None*)
Bases: *taurex.data.spectrum.array.ArraySpectrum*

Loads an observed spectrum from a text file and computes bin edges and bin widths. Spectrum must be 3-4 columns with ordering:

1. wavelength
2. spectral data
3. error
4. (optional) bin width

If no bin width is present then they are computed.

Parameters **filename** (*string*) – Path to observed spectrum file.

classmethod input_keywords()

10.14.4 Iraclis

class IraclisSpectrum(*filename=None*)
Bases: *taurex.data.spectrum.array.ArraySpectrum*

Loads an observation from Iraclis pickle data

Parameters **filename** (*string*) – Path to observed spectrum file.

classmethod input_keywords()

10.14.5 Taurex

class TaurexSpectrum(*filename=None*)
Bases: *taurex.data.spectrum.array.ArraySpectrum*

Observation is a taurex spectrum from a HDF5 file

An instrument function must have been used for this to work

Parameters **filename** (*string*) – Path to taurex spectrum HDF5 output.

10.14.6 Lightcurves

Module dealing with observed lightcurves

class ObservedLightCurve(*filename=None*)

Bases: *taurex.data.spectrum.spectrum.BaseSpectrum*

Loads an observed lightcurve from a pickle file.

Parameters **filename** (*str*) – Path to pickle file containing lightcurve data

property binEdges

Returns bin edges for wavelength grid

Returns **out**

Return type array

property binWidths

Widths for each bin in wavelength grid

Returns **out**

Return type array

create_binner()

Creates the appropriate binning object

property errorBar

Like *spectrum()* except its the error at each point in the lightcurve spectrum

Returns **err** – Error at each point in lightcurve spectrum

Return type array

classmethod input_keywords()

property rawData

Raw lightcurve data read from file

Returns **lc_data**

Return type array

property spectrum

Returns Light curve spectrum. The lightcurve spectrum comes in the form of multiple lightcurves stuck together into one long spectrum. The number of lightcurves is equal to the number of bins in *wavelengthGrid()*.

Returns **spectrum**

Return type array

property wavelengthGrid

Returns wavelength grid in microns

Returns **wlgrid**

Return type array

write(*output*)

10.15 Forward Models (taurex.model)

10.15.1 Base

class ForwardModel(*name*)

Bases: *taurex.log.logger.Logger*, *taurex.data.fittable.Fittable*, *taurex.output.writeable.Writeable*, *taurex.data.citation.Citable*

A base class for producing forward models

add_contribution(*contrib*)

build()

citations()

compute_error(*samples*, *wngrid=None*, *binner=None*)

defaultBinner()

property derivedParameters

property fittingParameters

generate_profiles()
Must return a dictionary of profiles you want to store after modeling

classmethod input_keywords()

model(*wngrid=None*, *cutoff_grid=True*)
Computes the forward model for a wngrid

model_full_contrib(*wngrid=None*, *cutoff_grid=True*)
Computes the forward model for a wngrid for each contribution

write(*output*)

10.15.2 Basic Model

class SimpleForwardModel(*name*, *planet=None*, *star=None*, *pressure_profile=None*, *temperature_profile=None*, *chemistry=None*, *nlayers=100*, *atm_min_pressure=0.0001*, *atm_max_pressure=1000000.0*)

Bases: *taurex.model.model.ForwardModel*

A 'simple' base model in the sense that its just a fairly standard single profiles model. It will handle settingup and initializing, building collecting parameters from given profiles for you. The only method that requires implementation is:

- *path_integral()*

Parameters

- **name** (*str*) – Name to use in logging
- **planet** (Planet, optional) – Planet model, default planet is Jupiter
- **star** (*Star*, optional) – Star model, default star is Sun-like
- **pressure_profile** (*PressureProfile*, optional) – Pressure model, alternative is to set nlayers, atm_min_pressure and atm_max_pressure

- **temperature_profile** (*TemperatureProfile*, optional) – Temperature model, default is an *Isothermal* profile at 1500 K
- **chemistry** (*Chemistry*, optional) – Chemistry model, default is *TaurexChemistry* with H2O and CH4
- **nlayers** (*int, optional*) – Number of layers. Used if pressure_profile is not defined.
- **atm_min_pressure** (*float, optional*) – Pressure at TOA. Used if pressure_profile is not defined.
- **atm_max_pressure** (*float, optional*) – Pressure at BOA. Used if pressure_profile is not defined.

property altitudeProfile
Atmospheric altitude profile in m

build()
Build the forward model. Must be called at least once before running *model()*

property chemistry
Chemistry model

citations()

collect_derived_parameters()
Collects all derived parameters from all profiles within the forward model

collect_fitting_parameters()
Collects all fitting parameters from all profiles within the forward model

compute_error(*samples*, *wngrid=None*, *binner=None*)
Computes standard deviations from samples

Parameters **samples** –

property densityProfile
Atmospheric density profile in m-3

generate_profiles()
Must return a dictionary of profiles you want to store after modeling

initialize_profiles()
Initializes all profiles

model(*wngrid=None*, *cutoff_grid=True*)
Runs the forward model

Parameters

- **wngrid** (array, optional) – Wavenumber grid, default is to use native grid
- **cutoff_grid** (*bool*) – Run model only on wngrid given, default is True

Returns

- **native_grid** (array) – Native wavenumber grid, clipped if wngrid passed
- **depth** (array) – Resulting depth
- **tau** (array) – Optical depth.
- **extra** (None) – Empty

model_contrib(*wngrid=None*, *cutoff_grid=True*)
Models each contribution seperately

model_full_contrib(*wngrid=None*, *cutoff_grid=True*)
Like *model_contrib()* except all components for each contribution are modelled

property nLayers
Number of layers

property nativeWavenumberGrid
Searches through active molecules to determine the native wavenumber grid

Returns **wngrid** – Native grid

Return type array

Raises **InvalidModelException** – If no active molecules in atmosphere

path_integral(*wngrid*, *return_contrib*)

property planet
Planet model

property pressure
Pressure model

property pressureProfile
Atmospheric pressure profile in Pa

property star
Star model

property temperature
Temperature model

property temperatureProfile
Atmospheric temperature profile in K

write(*output*)

10.15.3 Transmission

class TransmissionModel(*planet=None*, *star=None*, *pressure_profile=None*, *temperature_profile=None*, *chemistry=None*, *nlayers=100*, *atm_min_pressure=0.0001*, *atm_max_pressure=1000000.0*, *new_path_method=False*)
Bases: *taurex.model.simplemodel.SimpleForwardModel*

A forward model for transits

Parameters

- **planet** (Planet, optional) – Planet model, default planet is Jupiter
- **star** (*Star*, optional) – Star model, default star is Sun-like
- **pressure_profile** (*PressureProfile*, optional) – Pressure model, alternative is to set nlayers, atm_min_pressure and atm_max_pressure
- **temperature_profile** (*TemperatureProfile*, optional) – Temperature model, default is an *Isothermal* profile at 1500 K
- **chemistry** (*Chemistry*, optional) – Chemistry model, default is *TaurexChemistry* with H2O and CH4

- **nlayers** (*int, optional*) – Number of layers. Used if `pressure_profile` is not defined.
- **atm_min_pressure** (*float, optional*) – Pressure at TOA. Used if `pressure_profile` is not defined.
- **atm_max_pressure** (*float, optional*) – Pressure at BOA. Used if `pressure_profile` is not defined.

compute_absorption(*tau*, *dz*)

compute_path_length()

compute_path_length_old(*dz*)

classmethod input_keywords()

path_integral(*wngrid*, *return_contrib*)

10.15.4 Emission

class EmissionModel(*planet=None*, *star=None*, *pressure_profile=None*, *temperature_profile=None*, *chemistry=None*, *nlayers=100*, *atm_min_pressure=0.0001*, *atm_max_pressure=1000000.0*, *ngauss=4*)

Bases: *taurex.model.simplemodel.SimpleForwardModel*

A forward model for eclipses

Parameters

- **planet** (`Planet`, optional) – Planet model, default planet is Jupiter
- **star** (*Star*, optional) – Star model, default star is Sun-like
- **pressure_profile** (*PressureProfile*, optional) – Pressure model, alternative is to set `nlayers`, `atm_min_pressure` and `atm_max_pressure`
- **temperature_profile** (*TemperatureProfile*, optional) – Temperature model, default is an *Isothermal* profile at 1500 K
- **chemistry** (*Chemistry*, optional) – Chemistry model, default is *TaurexChemistry* with `H2O` and `CH4`
- **nlayers** (*int, optional*) – Number of layers. Used if `pressure_profile` is not defined.
- **atm_min_pressure** (*float, optional*) – Pressure at TOA. Used if `pressure_profile` is not defined.
- **atm_max_pressure** (*float, optional*) – Pressure at BOA. Used if `pressure_profile` is not defined.
- **ngauss** (*int, optional*) – Number of gaussian quadrature points, default = 4

compute_final_flux(*f_total*)

evaluate_emission(*wngrid*, *return_contrib*)

evaluate_emission_ktables(*wngrid*, *return_contrib*)

classmethod input_keywords()

property logBolometricFlux

partial_model(*wngrid=None*, *cutoff_grid=True*)

path_integral(*wngrid*, *return_contrib*)

set_num_gauss(*value*, *coeffs=None*)

set_quadratures(*mu*, *weight*, *coeffs=None*)

property usingKTables

write(*output*)

contribute_ktau_emission

10.15.5 Direct Image

class DirectImageModel(*planet=None*, *star=None*, *pressure_profile=None*, *temperature_profile=None*, *chemistry=None*, *nlayers=100*, *atm_min_pressure=0.0001*, *atm_max_pressure=1000000.0*, *ngauss=4*)

Bases: *taurex.model.emission.EmissionModel*

A forward model for direct imaging of exo-planets

Parameters

- **planet** (Planet, optional) – Planet model, default planet is Jupiter
- **star** (*Star*, optional) – Star model, default star is Sun-like
- **pressure_profile** (*PressureProfile*, optional) – Pressure model, alternative is to set nlayers, atm_min_pressure and atm_max_pressure
- **temperature_profile** (*TemperatureProfile*, optional) – Temperature model, default is an *Isothermal* profile at 1500 K
- **chemistry** (*Chemistry*, optional) – Chemistry model, default is *TaurexChemistry* with H2O and CH4
- **nlayers** (*int, optional*) – Number of layers. Used if pressure_profile is not defined.
- **atm_min_pressure** (*float, optional*) – Pressure at TOA. Used if pressure_profile is not defined.
- **atm_max_pressure** (*float, optional*) – Pressure at BOA. Used if pressure_profile is not defined.
- **ngauss** (*int, optional*) – Number of gaussian quadrature points, default = 4

compute_final_flux(*f_total*)

classmethod input_keywords()

10.16 Optimizers (taurex.optimizer)

10.16.1 Base

class Optimizer(*name*, *observed=None*, *model=None*, *sigma_fraction=0.1*)

Bases: *taurex.log.logger.Logger*, *taurex.data.citation.Citable*

A base class that handles fitting and optimization of forward models. The class handles the compiling and management of fitting parameters in forward models, in its current form it cannot fit and requires a class derived from it to implement the *compute_fit()* function.

Parameters

- **name** (*str*) – Name to be used in logging
- **observed** (*BaseSpectrum*, optional) – See *set_observed()*
- **model** (*ForwardModel*, optional) – See *set_model()*
- **sigma_fraction** (*float, optional*) – Fraction of weights to use in computing the error. (Default: 0.1)

chisq_trans(*fit_params*, *data*, *datastd*)

Computes the Chi-Squared between the forward model and observation. The steps taken are:

1. Forward model (FM) is updated with *update_model()*
2. FM is then computed at its native grid then binned.
3. Chi-squared between FM and observation is computed

Parameters

- **fit_params** (*list of parameter values*) – List of new parameter values to update the model
- **data** (obj:*ndarray*) – Observed spectrum
- **datastd** (obj:*ndarray*) – Observed spectrum error

Returns chi-squared

Return type float

compile_params()

compute_derived_trace(*solution*)

compute_fit()

Unimplemented. When inheriting this should be overwritten to perform the actual fit.

Raises **NotImplementedError** – Raised when a derived class does override this function

property derived_latex

Returns a list of the current values of a fitting parameter. This respects the `mode` setting

Returns List of each value of a fitting parameter

Return type `list`

property derived_names

Returns a list of the current values of a fitting parameter. This respects the `mode` setting

Returns List of each value of a fitting parameter

Return type `list`

property derived_values

Returns a list of the current values of a fitting parameter. This respects the `mode` setting

Returns List of each value of a fitting parameter

Return type `list`

disable_derived(*parameter*)

disable_fit(*parameter*)

Disables fitting of the parameter

Parameters **parameter** (*str*) – Name of the parameter we do not want to fit

enable_derived(*parameter*)

enable_fit(*parameter*)
Enables fitting of the parameter

Parameters **parameter** (*str*) – Name of the parameter we want to fit

fit(*output_size=<OutputSize.heavy: 6>*)

property fit_boundaries
Returns the fitting boundaries of the parameter

Returns List of boundaries for each fitting parameter. It takes the form of a python `tuple` with the form (`bound_min` , `bound_max`)

Return type `list`

property fit_latex
Returns the names of the parameters in LaTeX format

Returns List of parameter names in LaTeX format

Return type `list`

property fit_names
Returns the names of the model parameters we will be fitting

Returns List of names of parameters that will be fit

Return type `list`

property fit_values
Returns a list of the current values of a fitting parameter. This respects the `mode` setting

Returns List of each value of a fitting parameter

Return type `list`

property fit_values_nomode
Returns a list of the current values of a fitting parameter. Regardless of the `mode` setting

Returns List of each value of a fitting parameter

Return type `list`

generate_profiles(*solution*, *binning*)

generate_solution(*output_size=<OutputSize.heavy: 6>*)
Generates a dictionary with all solutions and other useful parameters

get_samples(*solution_id*)

get_solution()
** Requires implementation **

Generator for solutions and their median and MAP values

Yields

- **solution_no** (*int*) – Solution number
- **map** (`array`) – Map values
- **median** (`array`) – Median values

- **extra** (`list`) – List of tuples of extra information to store. Must be of form `(name, data)`

get_weights(*solution_id*)

classmethod input_keywords()

sample_parameters(*solution*)
Read traces and weights and return a random `sigma_fraction` sample of them

Parameters **solution** – a solution output from sampler

Yields

- **traces** (`array`) – Traces of a particular sample
- **weight** (*float*) – Weight of sample

set_boundary(*parameter*, *new_boundaries*)
Sets the boundary of the parameter

Parameters

- **parameter** (*str*) – Name of the parameter we want to change
- **new_boundaries** (*tuple of float*) – New fitting boundaries, with the form (`bound_min` , `bound_max`). These should not take into account the `mode` setting of a fitting parameter.

set_factor_boundary(*parameter*, *factors*)
Sets the boundary of the parameter based on a factor

Parameters

- **parameter** (*str*) – Name of the parameter we want to change
- **factor** (*tuple of float*) – To be written

set_mode(*parameter*, *new_mode*)
Sets the fitting mode of a parameter

Parameters

- **parameter** (*str*) – Name of the parameter we want to change
- **new_mode** (`linear` or `log`) – Sets whether the parameter is fit in linear or log space

set_model(*model*)

set_observed(*observed*)

set_prior(*parameter*, *prior*)

update_model(*fit_params*)
Updates the model with new parameters

Parameters **fit_params** (`list`) – A list of new values to apply to the model. The list of values are assumed to be in the same order as the parameters given by *fit_names()*

write(*output*)
Creates 'Optimizer' them respectively

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

write_fit(*output*)
Writes basic fitting parameters into output

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

Returns Group (or root) in output file written to

Return type *Output* or *OutputGroup*

write_optimizer(*output*)
Writes optimizer settings under the 'Optimizer' heading in an output file

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

Returns Group (or root) in output file written to

Return type *Output* or *OutputGroup*

compile_params(*fitparams*, *driveparams*, *fit_priors=None*)

10.16.2 Nestle (taurex.optimizer.nestle)

class NestleOptimizer(*observed=None*, *model=None*, *num_live_points=1500*, *method='multi'*, *tol=0.5*, *sigma_fraction=0.1*)
Bases: *taurex.optimizer.optimizer.Optimizer*

An optimizer that uses the nestle library to perform optimization.

Parameters

- **observed** (*BaseSpectrum*, optional) – Observed spectrum to optimize to
- **model** (*ForwardModel*, optional) – Forward model to optimize
- **num_live_points** (*int, optional*) – Number of live points to use in sampling
- **method** (`classic`, `single` or `multi`) – Nested sampling method to use. `classic` uses MCMC exploration, `single` uses a single ellipsoid and `multi` uses multiple ellipsoids (similar to Multinest)
- **tol** (*float*) – Evidence tolerance value to stop the fit. This is based on an estimate of the remaining prior volumes contribution to the evidence.
- **sigma_fraction** (*float, optional*) – Fraction of weights to use in computing the error. (Default: 0.1)

BIBTEX_ENTRIES = ['@misc{nestle,\n\n author = {Kyle Barbary},\n title = {Nestle sampli

compute_fit()
Computes the fit using nestle

get_samples(*solution_idx*)

get_solution()
Generator for solutions and their median and MAP values

Yields

- **solution_no** (*int*) – Solution number (always 0)
- **map** (`array`) – Map values
- **median** (`array`) – Median values
- **extra** (`list`) – Returns Statistics, fitting_params, raw_traces and raw_weights

get_weights(*solution_idx*)

classmethod input_keywords()

property numLivePoints

store_nestle_output(*result*)
This turns the output fron nestle into a dictionary that can be output by Taurex

Parameters **result** (dict) – Result from a nestle sample call

Returns Formatted dictionary for output

Return type dict

property tolerance

write_fit(*output*)
Writes basic fitting parameters into output

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

Returns Group (or root) in output file written to

Return type *Output* or *OutputGroup*

write_optimizer(*output*)
Writes optimizer settings under the 'Optimizer' heading in an output file

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

Returns Group (or root) in output file written to

Return type *Output* or *OutputGroup*

10.16.3 MultiNest (taurex.optimizer.multinest)

class MultiNestOptimizer(*multi_nest_path=None*, *observed=None*, *model=None*, *sampling_efficiency='parameter'*, *num_live_points=1500*, *max_iterations=0*, *search_multi_modes=True*, *num_params_cluster=None*, *maximum_modes=100*, *constant_efficiency_mode=False*, *evidence_tolerance=0.5*, *mode_tolerance=-1e+90*, *importance_sampling=False*, *resume=False*, *n_iter_before_update=100*, *multinest_prefix='1-'*, *verbose_output=True*, *sigma_fraction=0.1*)
Bases: *taurex.optimizer.optimizer.Optimizer*

compute_fit()
Unimplemented. When inheriting this should be overwritten to perform the actual fit.

Raises **NotImplementedError** – Raised when a derived class does override this function

generate_solution(*output_size=<OutputSize.heavy: 6>*)
Generates a dictionary with all solutions and other useful parameters

get_solution()
** Requires implementation **

Generator for solutions and their median and MAP values

Yields

- **solution_no** (*int*) – Solution number
- **map** (array) – Map values
- **median** (array) – Median values
- **extra** (list) – List of tuples of extra information to store. Must be of form (name, data)

write_fit(*output*)
Writes basic fitting parameters into output

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

Returns Group (or root) in output file written to

Return type *Output* or *OutputGroup*

write_optimizer(*output*)
Writes optimizer settings under the 'Optimizer' heading in an output file

Parameters **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to

Returns Group (or root) in output file written to

Return type *Output* or *OutputGroup*

10.16.4 PolyChord (taurex.optimizer.polychord)

10.16.5 dyPolyChord (taurex.optimizer.dypolychord)

10.17 Logging (taurex.log)

10.17.1 Logger

class Logger(*name*)
Bases: object

Standard logging using logger library

Parameters **name** (*str*) – Name used for logging

critical(*message*, **args*, ***kwargs*)
See logging.Logger

debug(*message*, **args*, ***kwargs*)
See logging.Logger

error(*message*, **args*, ***kwargs*)
See logging.Logger

info(*message*, **args*, ***kwargs*)
See logging.Logger

warning(*message*, **args*, ***kwargs*)
See logging.Logger

10.17.2 Module contents

disableLogging()

enableLogging()

setLogLevel(*level*)

10.18 Outputs (taurex.output)

10.18.1 Base

class Output(*name*)
Bases: *taurex.log.logger.Logger*

close()

create_group(*group_name*)

open()

store_dictionary(*dictionary*, *group_name=None*)

class OutputGroup(*name*)
Bases: *taurex.output.output.Output*

write_array(*array_name*, *array*, *metadata=None*)

write_list(*list_name*, *list_array*, *metadata=None*)

write_scalar(*scalar_name*, *scalar*, *metadata=None*)

write_string(*string_name*, *string*, *metadata=None*)

write_string_array(*string_name*, *string_array*, *metadata=None*)

10.18.2 taurex.output.hdf5 module

class HDF5Output(*filename*, *append=False*)
Bases: *taurex.output.output.Output*

close()

create_group(*group_name*)

open()

class HDF5OutputGroup(*entry*)
Bases: *taurex.output.output.OutputGroup*

create_group(*group_name*)

write_array(*array_name*, *array*, *metadata=None*)

write_scalar(*scalar_name*, *scalar*, *metadata=None*)

write_string(*string_name*, *string*, *metadata=None*)

write_string_array(*string_name*, *string_array*, *metadata=None*)

10.18.3 taurex.output.writeable module

class Writeable
Bases: object

write(*output*)

10.18.4 Module contents

10.19 Utilities

10.19.1 Submodules

10.19.2 taurex.util.emission module

Functions related to computing emission spectrums

black_body

black_body_numba

black_body_numba_II

black_body_numexpr(*lamb*, *temp*)

black_body_numpy(*lamb*, *temp*)

integrate_emission_layer(*dtau*, *layer_tau*, *mu*, *BB*)

integrate_emission_numba

10.19.3 taurex.util.math module

Optimized Math functions used in taurex

class OnlineVariance

Bases: `object`

USes the M2 algorithm to compute the variance in a streaming fashion

combine_variance(*averages*, *variance*, *counts*)

parallelVariance()

reset()

property sampleVariance

update(*value*, *weight=1.0*)

property variance

compute_rayleigh_cross_section(*wngrid*, *n*, *n_air=2.6867805e+25*, *king=1.0*)

intepr_bilin

intepr_bilin_double(*x11*, *x12*, *x21*, *x22*, *T*, *Tmin*, *Tmax*, *P*, *Pmin*, *Pmax*)

intepr_bilin_numba

intepr_bilin_numba_II

intepr_bilin_numexpr(*x11*, *x12*, *x21*, *x22*, *T*, *Tmin*, *Tmax*, *P*, *Pmin*, *Pmax*)

intepr_bilin_old(*x11*, *x12*, *x21*, *x22*, *T*, *Tmin*, *Tmax*, *P*, *Pmin*, *Pmax*)

interp_exp_and_lin(*x11*, *x12*, *x21*, *x22*, *T*, *Tmin*, *Tmax*, *P*, *Pmin*, *Pmax*)

2D interpolation

Applies linear interpolation across P and e interpolation across T between Pmin,Pmax and Tmin,Tmax

Parameters

- **x11** (*array*) – Array corresponding to Pmin,Tmin
- **x12** (*array*) – Array corresponding to Pmin,Tmax
- **x21** (*array*) – Array corresponding to Pmax,Tmin
- **x22** (*array*) – Array corresponding to Pmax,Tmax
- **T** (*float*) – Coordinate to exp interpolate to
- **Tmin** (*float*) – Nearest known T coordinate where Tmin < T
- **Tmax** (*float*) – Nearest known T coordinate where T < Tmax
- **P** (*float*) – Coordinate to linear interpolate to
- **Pmin** (*float*) – Nearest known P coordinate where Pmin < P
- **Pmax** (*float*) – Nearest known P coordinate where P < Tmax

interp_exp_and_lin_broken

interp_exp_and_lin_numpy(*x11*, *x12*, *x21*, *x22*, *T*, *Tmin*, *Tmax*, *P*, *Pmin*, *Pmax*)

2D interpolation

Applies linear interpolation across P and e interpolation across T between Pmin,Pmax and Tmin,Tmax

Parameters

- **x11** (*array*) – Array corresponding to Pmin,Tmin
- **x12** (*array*) – Array corresponding to Pmin,Tmax
- **x21** (*array*) – Array corresponding to Pmax,Tmin
- **x22** (*array*) – Array corresponding to Pmax,Tmax
- **T** (*float*) – Coordinate to exp interpolate to
- **Tmin** (*float*) – Nearest known T coordinate where Tmin < T
- **Tmax** (*float*) – Nearest known T coordinate where T < Tmax
- **P** (*float*) – Coordinate to linear interpolate to
- **Pmin** (*float*) – Nearest known P coordinate where Pmin < P
- **Pmax** (*float*) – Nearest known P coordinate where P < Tmax

interp_exp_numba

interp_exp_numpy(*x11*, *x12*, *T*, *Tmin*, *Tmax*)

interp_exp_only(*x11*, *x12*, *T*, *Tmin*, *Tmax*)

interp_lin_numba

interp_lin_numpy(*x11*, *x12*, *P*, *Pmin*, *Pmax*)

interp_lin_only

test_nan(*val*)

10.19.4 taurex.util.util module

General utility functions

bindown(*original_bin*, *original_data*, *new_bin*, *last_point=None*)
This method quickly bins down by taking the mean. The numpy histogram function is exploited to do this quickly

Parameters

- **original_bin** (numpy.array) – The original bins for the that we want to bin down
- **original_data** (numpy.array) – The associated data that will be averaged along the new bins
- **new_bin** (numpy.array) – The new binnings we want to use (must have less points than the original)

Returns Binned mean of original_data

Return type array

calculate_weight(*chem*)

class_for_name(*class_name*)

class_from_keyword(*keyword*, *class_filter=None*)

clip_native_to_wngrid(*native_grid*, *wngrid*)

compute_bin_edges(*wngrid*)

compute_dz(*altitude*)

conversion_factor(*from_unit*, *to_unit*)

create_grid_res(*resolution*, *wave_min*, *wave_max*)

decode_string_array(*f*)
Helper to decode strings from hdf5

ensure_string_utf8(*val*)

find_closest_pair(*arr*, *value) -> (<class 'int'>*, *<class 'int'>*)
Will find the indices that lie to the left and right of the value

arr[left] <= value <= arr[right]

If the value is less than the array minimum then it will always return left=0 and right=1

If the value is above the maximum

Parameters

- **arr** (array) – Array to search, must be sorted
- **value** (*float*) – Value to find in array

Returns

- **left** (*int*)
- **right** (*int*)

get_molecular_weight(*gasname*)
For a given molecule return the molecular weight in atomic units

Parameters **gasname** (*str*) – Name of molecule

Returns molecular weight in amu or 0 if not found

Return type float

has_duplicates(*arr*)

loadtxt2d(*intext*)
Wraps loadtext and either returns a 2d array or 1d array

merge_elements(*elem1*, *elem2*, *factor=1*)

molecule_texlabel(*gasname*)
For a given molecule return its latex form

Parameters **gasname** (*str*) – Name of molecule

Returns Latex form of the molecule or just the passed name if not found

Return type str

movingaverage(*a*, *n=3*)
Computes moving average

Parameters

- **a** (array) – Array to compute average
- **n** (*int*) – Averaging window

Returns Resultant array

Return type array

quantile_corner(*x*, *q*, *weights=None*)

- Taken from corner.py

__author__ = "Dan Foreman-Mackey (danfm@nyu.edu)" __copyright__ = "Copyright 2013-2015 Daniel Foreman-Mackey"

Like numpy.percentile, but:

- Values of q are quantiles [0., 1.] rather than percentiles [0., 100.]
- scalar q not supported (q must be iterable)
- optional weights on x

Parameters

- **x** (array) – Input array or object that can be converted to an array.
- **q** (array or float) – Percentile or sequence of percentiles to compute, which must be between 0 and 1 inclusive.
- **weights** (array or float , optional) – Weights on x

Returns **percentile**

Return type scalar or ndarray

random_int_iter(*total*, *fraction*)

read_error_into_dict(*l*, *d*)
Reads the error into dict?

read_error_line(*l*)
Reads line?

read_table(*txt*, *d=None*, *title=None*)
Yeah whatever i give up

recursively_save_dict_contents_to_output(*output*, *dic*)
Will recursive write a dictionary into output.

Parameters

- **output** (*Output* or *OutputGroup*) – Group (or root) in output file to write to
- **dic** (dict) – Dictionary we want to write

sanitize_molecule_string(*molecule*)
Cleans a molecule string to match up with molecule naming in TauREx3.

e.g:

H2O -> H2O

1H2-16O -> H2O

Parameters **molecule** (*str*) – Molecule to sanitize

Returns Sanitized name

Return type str

split_molecule_elements(*molecule=None*, *tokens=None*)

store_thing(*output*, *key*, *item*)

tokenize_molecule(*molecule*)

weighted_avg_and_std(*values*, *weights*, *axis=None*)
Computes weight average and standard deviation

Parameters

- **values** (array) – Input array
- **weights** (array) – Must be same shape as values

axis [int , optional] axis to perform weighting

wnwidth_to_wlwidth(*wngrid*, *wnwidth*)

10.19.5 Module contents

Common functions that are used and are quite helpful

10.20 taurex.parameter package

10.20.1 Submodules

10.20.2 taurex.parameter.factory module

chemistry_factory(*profile_type*)

create_chemistry(*config*)

create_gas_profile(*config*)

create_instrument(*config*)

create_klass(*config*, *klass*, *is_mixin*)

create_model(*config*, *gas*, *temperature*, *pressure*, *planet*, *star*, *observation=None*)

create_observation(*config*)

create_optimizer(*config*)

create_planet(*config*)

create_pressure_profile(*config*)

create_prior(*prior*)

create_profile(*config*, *factory*, *baseclass=None*, *keyword_type='profile_type'*)

create_star(*config*)

create_temperature_profile(*config*)

detect_and_return_klass(*python_file*, *baseclass*)

determine_klass(*config*, *field*, *factory*, *baseclass=None*)

gas_factory(*profile_type*)

generate_contributions(*config*)

generic_factory(*profile_type*, *baseclass*)

get_keywordarg_dict(*klass*, *is_mixin=False*)

instrument_factory(*instrument*)

mixin_factory(*profile_type*, *baseclass*)

model_factory(*model_type*)

observation_factory(*observation*)

optimizer_factory(*optimizer*)

planet_factory(*planet_type*)

pressure_factory(*profile_type*)

star_factory(*star_type*)

temp_factory(*profile_type*)

10.20.3 taurex.parameter.parameterparser module

class ParameterParser

Bases: *taurex.log.logger.Logger*

create_manual_binning(*config*)

create_snr(*binner*, *config*)

generate_appropriate_model(*obs=None*)

generate_binning()

generate_chemistry_profile()

generate_derived_parameters()

generate_fitting_parameters()

generate_instrument(*binner=None*)

generate_lightcurve()

generate_model(*chemistry=None*, *pressure=None*, *temperature=None*, *planet=None*, *star=None*, *obs=None*)

generate_observation()

generate_optimizer()

generate_planet()

generate_pressure_profile()

generate_star()

generate_temperature_profile()

read(*filename*)

setup_globals()

setup_optimizer(*optimizer: taurex.optimizer.optimizer.Optimizer*)
Setup fitting parameters for optimizer

transform(*section*, *key*)

10.20.4 Module contents

10.21 Mixin (taurex.mixin)

class ChemistryMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class ContributionMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class ForwardModelMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class GasMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class InstrumentMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class Mixin(***kwargs*)
Bases: *taurex.data.fittable.Fittable*, *taurex.data.citation.Citable*

classmethod input_keywords()

class ObservationMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class OptimizerMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class PlanetMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class PressureMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

class StarMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

A mixin that enhances *Star*

class TemperatureMixin(***kwargs*)
Bases: *taurex.mixin.core.Mixin*

build_new_mixed_class(*base_klass*, *mixins*)

determine_mixin_args(*klasses*)

enhance_class(*base_klass*, *mixins*, ***kwargs*)

mixed_init(*self*, ***kwargs*)

class MakeFreeMixin(***kwargs*)
Bases: *taurex.mixin.core.ChemistryMixin*

Provides a *addGas()* method to any chemistry class that will either inject or force a molecule to become a *Gas* object. Allowing them to be freely changed or retrieved.

For example lets enhance ACE:

```
>>> from taurex_ace import ACEChemistry
>>> from taurex.mixin import enhance_class, MakeFreeMixin
>>> old_ace = ACEChemistry()
>>> new_ace = enhance_class(ACEChemistry, MakeFreeMixin)
```

`new_ace` behaves the same as `old_ace`:

```
>>> new_ace.ace_metallicity
1.0
```

And we see the same molecules and fitting parameters exist:

```
>>> old_ace.gases == new_ace.gases
True
>>> new_ace.gases
['CH3COOOH', 'C4H9O', ... 'HNC', 'HON', 'NCN']
>>> new_ace.fitting_parameters().keys()
dict_keys(['ace_metallicity', 'metallicity', 'ace_co', 'C_O_ratio'])
```

`new_ace` is embued with the

```
>>> from taurex.chemistry import ConstantGas
>>> new_ace.addGas(ConstantGas('TiO',1e-8)).addGas(ConstantGas('VO',1e-8))
>>> new_ace.gases == old_ace.gases
False
>>> new_ace.gases
['CH3COOOH', 'C4H9O', ... 'HNC', 'HON', 'NCN', 'TiO', 'VO']
```

And indeed see that they are included. We can also retrieve them:

```
>>> new_ace.fitting_parameters().keys()
dict_keys(['TiO', 'VO', 'ace_metallicity', 'metallicity', 'ace_co', 'C_O_ratio'])
```

Finally we can force an existing molecule like CH4 into becoming a Gas:

```
>>> new_ace.addGas(ConstantGas('CH4',1e-5))
```

And see that it is now a retrieval parameter as well.

```
>>> new_ace.fitting_parameters().keys()
dict_keys(['TiO', 'VO', 'CH4', 'ace_metallicity', 'metallicity', 'ace_co', 'C_O_
↪ratio'])
```

property activeGasMixProfile
Active gas layer by layer mix profile

Returns active_mix_profile

Return type array

property activeGases

addGas(*gas*)
Adds a gas in the atmosphere.

Parameters **gas** (*Gas*) – Gas to add into the atmosphere. Only takes effect on next initialization call.

compute_mu_profile(*nlayers*)
Computes molecular weight of atmosphere for each layer

Parameters **nlayers** (*int*) – Number of layers

determine_new_mix_mask()

fitting_parameters()
Overrides the fitting parameters to return one with all the gas profile parameters as well

Returns fit_param

Return type dict

property gases

property inactiveGasMixProfile
Inactive gas layer by layer mix profile

Returns inactive_mix_profile

Return type array

property inactiveGases

initialize_chemistry(*nlayers=100*, *temperature_profile=None*, *pressure_profile=None*, *altitude_profile=None*)

classmethod input_keywords()

property muProfile

class TempScaler(***kwargs*)
Bases: *taurex.mixin.core.TemperatureMixin*

classmethod input_keywords()

property profile

property scaleFactor
None

10.22 MPI (taurex.mpi)

Module for wrapping MPI functions. Most functions will do nothing if mpi4py is not present.

allgather(*value*)

allocate_as_shared(*arr*, *logger=None*, *force_shared=False*)

Converts a numpy array into an MPI shared memory. This allow for things like opacities to be loaded only once per node when using MPI. Only activates if mpi4py installed and when enabled via the `mpi_use_shared` input:

```
[Global]
mpi_use_shared = True
```

or `force_shared=True` otherwise does nothing and returns the same array back

Parameters

- **arr** (*numpy array*) – Array to convert
- **logger** (*Logger*) – Logger object to print outputs
- **force_shared** (*bool*) – Force conversion to shared memory

Returns If enabled and MPI present, shared memory version of array otherwise the original array

Return type array

allreduce(*value*, *op*)

barrier(*comm=None*)

Waits for all processes to finish. Does nothing if mpi4py not present

Parameters **comm** (*int, optional*) – MPI communicator, default is MPI_COMM_WORLD

broadcast(*array*, *rank=0*)

convert_op(*operation*)

get_rank

Gets rank or returns 0 if mpi is not installed

Parameters **comm** (*int, optional*) – MPI communicator, default is MPI_COMM_WORLD

Returns Rank of process in communitor or 0 if MPI is not installed

Return type int

nprocs

Gets number of processes or returns 1 if mpi is not installed

Returns Rank of process or 1 if MPI is not installed

Return type int

only_master_rank(*f*)

A decorator to ensure only the master MPI rank can run it

shared_comm

Returns the process id within a node. Used for shared memory

shared_rank

CHAPTER

ELEVEN

CITATION

TauREx will output a bibliography at program finish for components used in a run (including plugins) or store a `.bib` file when run with `--bibtex filename.bib`. We also list references for components in the base TauREx3 installation.

11.1 Taurex 3

If you use TauREx 3 in your research and publications, please cite the arXiv preprint or the submitted publication:

```
TauREx III: A fast, dynamic and extendable framework for retrievals
A. F. Al-Refaie, Q. Changeat, I.P. Waldmann and G. Tinetti
ApJ, submitted, 2019
```

11.2 Retrieval

If you make use of any of these samplers then please cite the relevant papers

PyMultiNest and *MultiNest*

```
(PyMultiNest)
X-ray spectral modelling of the AGN obscuring region in the CDFS: Bayesian model␣
↪selection and catalogue
J. Buchner, A. Georgakakis, K. Nandra, L. Hsu, C. Rangel, M. Brightman, A. Merloni, M.
↪ Salvato, J. Donley and D. Kocevski
A&A, 564 (2014) A125
doi: 10.1051/0004-6361/201322971

MultiNest: an efficient and robust Bayesian inference tool for cosmology and particle␣
↪physics
F. Feroz, M.P. Hobson, M. Bridges
Mon. Not. Roy. Astron. Soc. 398: 1601-1614,2009
doi: 10.1111/j.1365-2966.2009.14548.x
```

PolyChord

```
polychord: next-generation nested sampling
W. J. Handley, M. P. Hobson, A. N. Lasenby
Mon. Not. Roy. Astron. Soc. 453: 4384-4398,2015
doi: 10.1093/mnras/stv1911
```

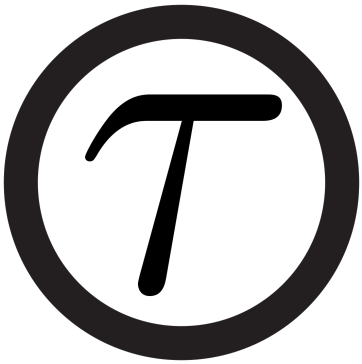
dyPolyChord

```
Dynamic nested sampling: an improved algorithm for parameter estimation and evidence␣
↪calculation
E. Higson, W. Handley, M. Hobson, A. Lasenby
Statistics and Computing volume 29, 891-913, 2019
doi: 10.1007/s11222-018-9844-0
```

CHAPTER

TWELVE

TAUREX 3.1 - A TRUE EXOPLANET RETRIEVAL FRAMEWORK



TauREx 3 (Tau Retrieval for Exoplanets) is an open-source fully bayesian inverse atmospheric retrieval framework licensed under BSDv3. It aims to make exoplanetary atmosphere modelling and retrievals fast, easy and flexible!

TauREx 3 offers a fully customizable framework that allows you to mix and match atmospheric parameters and add in your own to perform modelling and retrievals.

For scientists, the standalone taurex program provides a wealth of parameters to build forward models, simulate instruments and perform retrievals.

For developers, TauREx3 provides a rich library of classes to build your own programs and any new atmospheric parameters you create can be used in the standalone program like it was always there!

TauREx3 can be expanded

If you use TauREx 3 in you work, please see the guide to *Citation*.

Want to install it? Head here: *Installation*

Want to jump into the `taurex` program? Head here: *Quickstart*

Want to try out the library? Try here: *Library*

Release 3.1.4-alpha

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X