qnm Documentation

Release 0.4.0

Leo C. Stein

Sep 24, 2019

Contents:

1 Welcome to qnm **3**
- 1.1 Installation . . . 3
- 1.2 Dependencies . . . 4
- 1.3 Documentation . . . 4
- 1.4 Usage . . . 4
- 1.5 Precision and validation . . . 6
- 1.6 Spherical-spheroidal decomposition . . . 6
- 1.7 Contributing . . . 6
- 1.8 How to cite . . . 7
- 1.9 Credits . . . 7

2 Package API **9**
- 2.1 qnm . . . 9
- 2.2 qnm.angular . . . 10
- 2.3 qnm.cached . . . 13
- 2.4 qnm.contfrac . . . 16
- 2.5 qnm.nearby . . . 19
- 2.6 qnm.radial . . . 21
- 2.7 qnm.spinsequence . . . 24
- 2.8 qnm.schwarzschild . . . 25
- 2.9 qnm.schwarzschild.approx . . . 26
- 2.10 qnm.schwarzschild.overtonesequence . . . 27
- 2.11 qnm.schwarzschild.tabulated . . . 29

3 Indices and tables **33**

Bibliography **35**

Python Module Index **37**

Index **39**

For a quick start, take a look at the usage section of the Welcome node.

CHAPTER 1

Welcome to qnm

`qnm` is an open-source Python package for computing the Kerr quasinormal mode frequencies, angular separation constants, and spherical-spheroidal mixing coefficients. The `qnm` package includes a Leaver solver with the Cook-Zalutskiy spectral approach to the angular sector, and a caching mechanism to avoid repeating calculations.

With this python package, you can compute the QNMs labeled by different (s,l,m,n), at a desired dimensionless spin parameter 0a<1. The angular sector is treated as a spectral decomposition of spin-weighted *spheroidal* harmonics into spin-weighted spherical harmonics. Therefore you get the spherical-spheroidal decomposition coefficients for free when solving for ω and A (*see below for details*).

We have precomputed a large cache of low-lying modes (s=-2 and s=-1, all l<8, all n<7). These can be automatically installed with a single function call, and interpolated for good initial guesses for root-finding at some value of a.

1.1 Installation

1.1.1 PyPI

qnm is available through PyPI:

```
pip install qnm
```

1.1.2 From source

```
git clone https://github.com/duetosymmetry/qnm.git
cd qnm
python setup.py install
```

If you do not have root permissions, replace the last step with `python setup.py install --user`. Instead of using `setup.py` manually, you can also replace the last step with `pip install .` or `pip install --user .`.

1.2 Dependencies

All of these can be installed through pip or conda.

- numpy
- scipy
- numba
- tqdm (just for `qnm.download_data()` progress)
- pathlib2 (backport of `pathlib` to pre-3.4 python)

1.3 Documentation

Automatically-generated API documentation is available on Read the Docs: qnm.

1.4 Usage

The highest-level interface is via `qnm.cached.KerrSeqCache`, which loads cached *spin sequences* from disk. A spin sequence is just a mode labeled by (s,l,m,n), with the spin a ranging from a=0 to some maximum, e.g. 0.9995. A large number of low-lying spin sequences have been precomputed and are available online. The first time you use the package, download the precomputed sequences:

```
import qnm

qnm.download_data() # Only need to do this once
# Trying to fetch https://duetosymmetry.com/files/qnm/data.tar.bz2
# Trying to decompress file /<something>/qnm/data.tar.bz2
# Data directory /<something>/qnm/data contains 860 pickle files
```

Then, use `qnm.modes_cache` to load a `qnm.spinsequence.KerrSpinSeq` of interest. If the mode is not available, it will try to compute it (see detailed documentation for how to control that calculation).

```
grav_220 = qnm.modes_cache(s=-2,l=2,m=2,n=0)
omega, A, C = grav_220(a=0.68)
print(omega)
# (0.5239751042900845-0.08151262363119974j)
```

Calling a spin sequence `seq` with `seq(a)` will return the complex quasinormal mode frequency omega, the complex angular separation constant A, and a vector C of coefficients for decomposing the associated spin-weighted spheroidal harmonics as a sum of spin-weighted spherical harmonics (*see below for details*).

Visual inspections of modes are very useful to check if the solver is behaving well. This is easily accomplished with matplotlib. Here are some partial examples (for the full examples, see the file `notebooks/examples.ipynb` in the source repo):

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

s, l, m = (-2, 2, 2)
mode_list = [(s, l, m, n) for n in np.arange(0,7)]
```

(continues on next page)

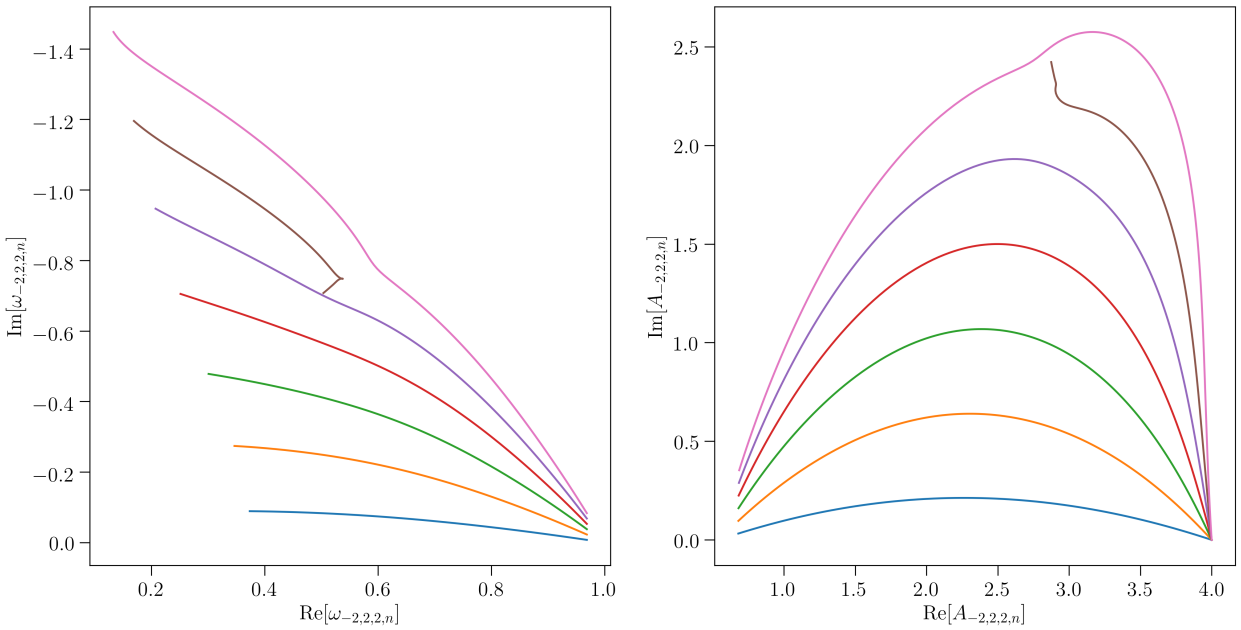
(continued from previous page)

```
modes = { ind : qnm.modes_cache(*ind) for ind in mode_list }

plt.subplot(1, 2, 1)
for mode, seq in modes.items():
    plt.plot(np.real(seq.omega),np.imag(seq.omega))

plt.subplot(1, 2, 2)
for mode, seq in modes.items():
    plt.plot(np.real(seq.A),np.imag(seq.A))
```

Which results in the following figure (modulo formatting):

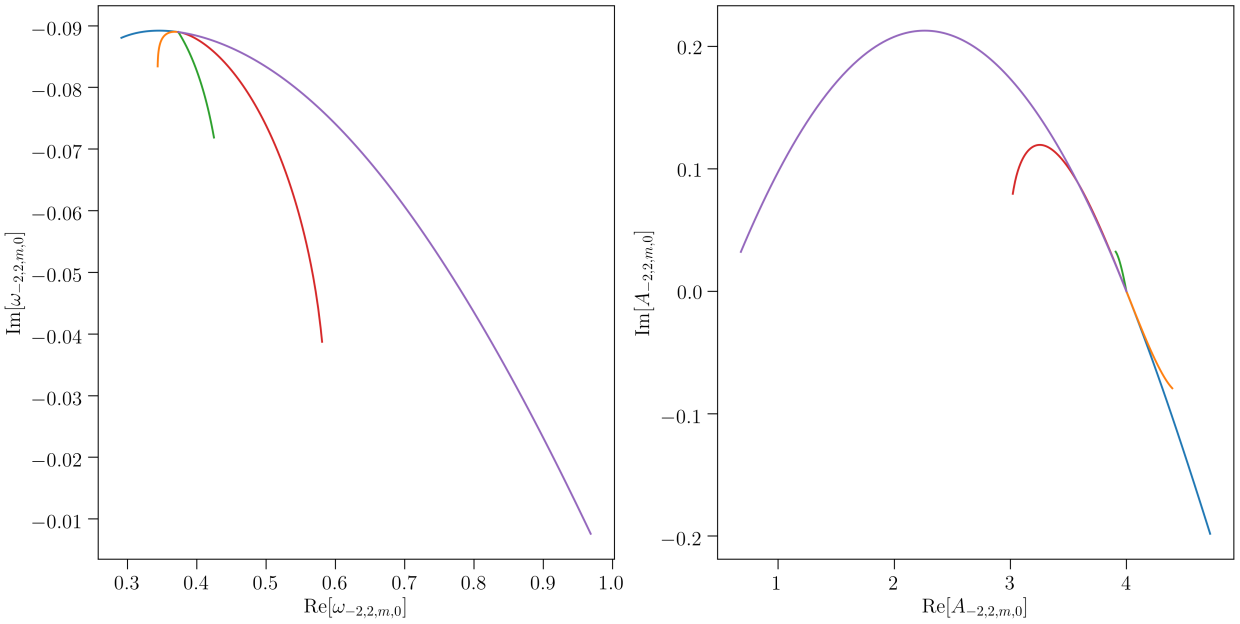



```
s, l, n = (-2, 2, 0)
mode_list = [(s, l, m, n) for m in np.arange(-l,l+1)]
modes = { ind : qnm.modes_cache(*ind) for ind in mode_list }

plt.subplot(1, 2, 1)
for mode, seq in modes.items():
    plt.plot(np.real(seq.omega),np.imag(seq.omega))

plt.subplot(1, 2, 2)
for mode, seq in modes.items():
    plt.plot(np.real(seq.A),np.imag(seq.A))
```

Which results in the following figure (modulo formatting):



1.5 Precision and validation

The default tolerances for continued fractions, `cf_tol`, is 1e-10, and for complex root-polishing, `tol`, is DBL_EPSILON1.5e-8. These can be changed at runtime so you can re-polish the cached values to higher precision.

Greg Cook's precomputed data tables (which were computed with arbitrary-precision arithmetic) can be used for validating the results of this code. See the comparison notebook `notebooks/Comparison-against-Cook-data.ipynb` to see such a comparison, which can be modified to compare any of the available modes.

1.6 Spherical-spheroidal decomposition

The angular dependence of QNMs are naturally spin-weighted *spheroidal* harmonics. The spheroidals are not actually a complete orthogonal basis set. Meanwhile spin-weighted *spherical* harmonics are complete and orthonormal, and are used much more commonly. Therefore you typically want to express a spheroidal (on the left hand side) in terms of sphericals (on the right hand side),

Here min=max(|ml|,|sl|) and max can be chosen at run time. The C coefficients are returned as a complex ndarray, with the zeroth element corresponding to min. To avoid indexing errors, you can get the ndarray of values by calling `qnm.angular.ells`, e.g.

```
ells = qnm.angular.ells(s=-2, m=2, l_max=grav_220.l_max)
```

1.7 Contributing

Contributions are welcome! There are at least two ways to contribute to this codebase:

1. If you find a bug or want to suggest an enhancement, use the issue tracker on GitHub. It's a good idea to look through past issues, too, to see if anybody has run into the same problem or made the same suggestion before.
2. If you will write or edit the python code, we use the fork and pull request model.

You are also allowed to make use of this code for other purposes, as detailed in the MIT license. For any type of contribution, please follow the code of conduct.

1.8 How to cite

If this package contributes to a project that leads to a publication, please acknowledge this by citing the `qnm` article in JOSS. The following BibTeX entry is available in the `qnm.__bibtex__` string:

```
@article{Stein:2019mop,
      author         = "Stein, Leo C.",
      title          = "{qnm: A Python package for calculating Kerr quasinormal
                        modes, separation constants, and spherical-spheroidal
                        mixing coefficients}",
      year           = "2019",
      eprint         = "1908.10377",
      archivePrefix  = "arXiv",
      primaryClass   = "gr-qc",
      SLACcitation   = "%%CITATION = ARXIV:1908.10377;%%"
}
```

1.9 Credits

The code is developed and maintained by Leo C. Stein.

CHAPTER 2

Package API

qnm	Calculate quasinormal modes of Kerr black holes.
qnm.angular	Solve the angular Teukolsky equation via spectral decomposition.
qnm.cached	Caching interface to Kerr QNMs
qnm.contfrac	Infinite continued fractions via Lentz's method.
qnm.nearby	Find a nearby root of the coupled radial/angular Teukolsky equations.
qnm.radial	Solve the radial Teukolsky equation via Leaver's method.
qnm.spinsequence	Follow a QNM labeled by (s,l,m,n) as spin varies from a=0 upwards.
qnm.schwarzschild	Finding QNMs of Schwarzschild (numerically and approximately).
qnm.schwarzschild.approx	Analytic approximations for Schwarzschild QNMs.
qnm.schwarzschild.overtonesequence	Follow a Schwarzschild QNM sequence (s,l) from n=0 upwards.
qnm.schwarzschild.tabulated	Computing, loading, and storing tabulated Schwarzschild QNMs.

2.1 qnm

Calculate quasinormal modes of Kerr black holes.

The highest-level interface is via *qnm.cached.KerrSeqCache*, which will fetch instances of *qnm.spinsequence.KerrSpinSeq*. This is most clearly demonstrated with an example.

2.1.1 Examples

```
>>> import qnm
>>> # qnm.download_data() # Only need to do this once
>>> grav_220 = qnm.modes_cache(s=-2,l=2,m=2,n=0)
>>> omega, A, C = grav_220(a=0.68)
>>> print(omega)
(0.5239751042900845-0.08151262363119986j)
```

Members

qnm.download_data([overwrite])	Fetch and decompress tarball of precomputed spin sequences from the web.
qnm.modes_cache(s, l, m, n[, ...])	Interface to the cache of QNMs.

qnm.**download_data**(*overwrite=False*)

Fetch and decompress tarball of precomputed spin sequences from the web.

Parameters

overwrite: bool, optional [default: False] If there is already a tarball on disk, this flag controls whether or not it is overwritten.

qnm.**modes_cache**(*s*, *l*, *m*, *n*, *compute_if_not_found=None*, *compute_pars=None*) **= <qnm.cached.KerrSeqCache object>**

Interface to the cache of QNMs. This is a singleton instance of *qnm.cached.KerrSeqCache*. It can be called like a function *qnm.modes_cache(s,l,m,n)* to get a specific mode, the result being an instance of *qnm.spinsequence.KerrSpinSeq*.

2.2 qnm.angular

Solve the angular Teukolsky equation via spectral decomposition.

For a given complex QNM frequency ω, the separation constant and spherical-spheroidal decomposition are found as an eigenvalue and eigenvector of an (infinite) matrix problem. The interface to solving this problem is *C_and_sep_const_closest()*, which returns a certain eigenvalue A and eigenvector C. The eigenvector contains the C coefficients in the equation

$$ {}_sY_{\ell m}(\theta,\phi;a\omega) = \sum_{\ell'=\ell_{\min}(s,m)}^{\ell_{\max}} C_{\ell'\ell m}(a\omega)\ {}_sY_{\ell' m}(\theta,\phi)\,. $$

Here min=max(|ml,|sl) (see *l_min()*), and max can be chosen at run time. The C coefficients are returned as a complex ndarray, with the zeroth element corresponding to min. You can get the associated values by calling *ells()*.

Functions

C_and_sep_const_closest(A0, s, c, m, l_max)	Get a single eigenvalue and eigenvector of decomposition matrix, where the eigenvalue is closest to some guess A0.
M_matrix(s, c, m, l_max)	Spherical-spheroidal decomposition matrix truncated at l_max.

Continued on next page

Table 3 – continued from previous page

M_matrix_elem(s, c, m, l, lprime)	The (l, lprime) matrix element from the spherical-spheroidal decomposition matrix from Eq.
M_matrix_old(s, c, m, l_max)	Legacy function.
ells(s, m, l_max)	Vector of values in C vector and M matrix.
give_M_matrix_elem_ufunc(s, c, m)	Legacy function.
l_min(s, m)	Minimum allowed value of l for a given s, m.
sep_const_closest(A0, s, c, m, l_max)	Gives the separation constant that is closest to A0.
sep_consts(s, c, m, l_max)	Finds eigenvalues of decomposition matrix, i.e.
swspericalh_A(s, l, m)	Angular separation constant at a=0.

qnm.angular.**C_and_sep_const_closest**(*A0*, *s*, *c*, *m*, *l_max*)

Get a single eigenvalue and eigenvector of decomposition matrix, where the eigenvalue is closest to some guess A0.

Parameters

A0: complex Value close to the desired separation constant.

s: int Spin-weight of interest

c: complex Oblateness of spheroidal harmonic

m: int Magnetic quantum number

l_max: int Maximum angular quantum number

Returns

complex, complex ndarray The first element of the tuple is the eigenvalue that is closest in value to A0. The second element of the tuple is the corresponding eigenvector. The 0th element of this ndarray corresponds to *l_min()*.

qnm.angular.**M_matrix**(*s*, *c*, *m*, *l_max*)

Spherical-spheroidal decomposition matrix truncated at l_max.

Parameters

s: int Spin-weight of interest

c: complex Oblateness of the spheroidal harmonic

m: int Magnetic quantum number

l_max: int Maximum angular quantum number

Returns

complex ndarray Decomposition matrix

qnm.angular.**M_matrix_elem**(*s*, *c*, *m*, *l*, *lprime*)

The (l, lprime) matrix element from the spherical-spheroidal decomposition matrix from Eq. (55).

Parameters

s: int Spin-weight of interest

c: complex Oblateness of the spheroidal harmonic

m: int Magnetic quantum number

l: int Angular quantum number of interest

lprime: int Primed quantum number of interest

Returns

complex Matrix element M_{l, lprime}

qnm.angular.**M_matrix_old**(*s*, *c*, *m*, *l_max*)
Legacy function. Same as *M_matrix()* except trying to be cute with ufunc's, requiring scope capture with temp func inside *give_M_matrix_elem_ufunc()*, which meant that numba could not speed up this method. Remains here for testing purposes. See documentation for *M_matrix()* parameters and return value.

qnm.angular.**ells**(*s*, *m*, *l_max*)
Vector of values in C vector and M matrix.

The format of the C vector and M matrix is that the 0th element corresponds to l_min(s,m) (see *l_min()*).

Parameters

s: int Spin-weight of interest

m: int Magnetic quantum number

l_max: int Maximum angular quantum number

Returns

int ndarray Vector of values, starting from l_min

qnm.angular.**give_M_matrix_elem_ufunc**(*s*, *c*, *m*)
Legacy function. Gives ufunc that implements matrix elements of the spherical-spheroidal decomposition matrix. This function is used by *M_matrix_old()*.

Parameters

s: int Spin-weight of interest

c: complex Oblateness of the spheroidal harmonic

m: int Magnetic quantum number

Returns

ufunc Implements elements of M matrix

qnm.angular.**l_min**(*s*, *m*)
Minimum allowed value of l for a given s, m.

The formula is l_min = max(|m|,|s|).

Parameters

s: int Spin-weight of interest

m: int Magnetic quantum number

Returns

int l_min

qnm.angular.**sep_const_closest**(*A0*, *s*, *c*, *m*, *l_max*)
Gives the separation constant that is closest to A0.

Parameters

A0: complex Value close to the desired separation constant.

s: int Spin-weight of interest

c: complex Oblateness of spheroidal harmonic

m: int Magnetic quantum number

l_max: int Maximum angular quantum number

Returns

complex Separation constant that is the closest to A0.

qnm.angular.**sep_consts**(*s*, *c*, *m*, *l_max*)
Finds eigenvalues of decomposition matrix, i.e. the separation constants, As.

Parameters

s: int Spin-weight of interest

c: complex Oblateness of spheroidal harmonic

m: int Magnetic quantum number

l_max: int Maximum angular quantum number

Returns

complex ndarray Eigenvalues of spherical-spheroidal decomposition matrix

qnm.angular.**swsphericalh_A**(*s*, *l*, *m*)
Angular separation constant at a=0.

Eq. (50). Has no dependence on m. The formula is A_0 = l(l+1) - s(s+1)

Parameters

s: int Spin-weight of interest

l: int Angular quantum number of interest

m: int Magnetic quantum number, ignored

Returns

int Value of A(a=0) = l(l+1) - s(s+1)

2.3 qnm.cached

Caching interface to Kerr QNMs

This is a high-level interface to the package. The global cache *qnm.modes_cache* (an instance of *KerrSeqCache*) will return instances of *qnm.spinsequence.KerrSpinSeq* from memory or disk. If a spin sequence is neither in memory nor on disk then it will be computed and returned.

Use *download_data()* to fetch a collection of precomputed spin sequences from the web.

Functions

build_package_default_cache(ksc)	Compute the standard list of modes that this package promises to have in its cache.
download_data([overwrite])	Fetch and decompress tarball of precomputed spin sequences from the web.
get_cachedir()	Return the location of the cache directory.
get_home()	Return the user's home directory.
load_cached_mode(s, l, m, n)	Read a KerrSpinSeq from disk.

Continued on next page

Table 4 – continued from previous page

mode_pickle_path(s, l, m, n)	Construct the path to a pickle file for the mode (s, l, m, n)
write_mode(spin_seq[, pickle_path])	Write an instance of KerrSpinSeq to disk.

Classes

KerrSeqCache([init_schw, ...])	High-level caching interface for getting precomputed spin sequences.

class qnm.cached.**KerrSeqCache**(*init_schw=False*, *compute_if_not_found=True*, *compute_pars=None*)

High-level caching interface for getting precomputed spin sequences.

An instance of *KerrSeqCache* will return instances of *qnm.spinsequence.KerrSpinSeq* from memory or disk. If a spin sequence is neither in memory nor on disk then it will be computed and returned.

Use *download_data()* to fetch a collection of precomputed spin sequences from the web.

Parameters

init_schw: bool, optional [default: False] Value of init flag to pass to *qnm.schwarzschild.tabulated.QNMDict*. You should set this to True the first time in a session that you create a QNMDict (most likely via this class).

compute_if_not_found: bool, optional [default: True] If a mode sequence is not found on disk, this flag controls whether to try to compute the sequence from scratch.

compute_pars: dict, optional [default: None] A dict of parameters to pass to *qnm.spinsequence.KerrSpinSeq* if computing a mode sequence from scratch.

Examples

```
>>> import qnm
>>> # qnm.download_data() # Only need to do this once
>>> grav_220 = qnm.modes_cache(s=-2,l=2,m=2,n=0)
>>> omega, A, C = grav_220(a=0.68)
>>> print(omega)
(0.5239751042900845-0.08151262363119986j)
```

Methods

__call__(self, s, l, m, n[, ...])	Load a *qnm.spinsequence.KerrSpinSeq* from the cache or from disk if available.
write_all(self)	Write all of the modes in the cache to disk.

__call__(*self*, *s*, *l*, *m*, *n*, *compute_if_not_found=None*, *compute_pars=None*)

Load a *qnm.spinsequence.KerrSpinSeq* from the cache or from disk if available.

If the mode sequence is not available on disk, optionally compute it from scratch.

Parameters

s: int Spin-weight of field of interest.

l: int Multipole number of mode. l >= angular.l_min(s, m)

m: int Azimuthal number of mode.

n: int Overtone number of mode.

compute_if_not_found: bool, optional [default: self.compute_if_not_found] Whether or not to compute from scratch the spin sequence if it is not available on disk.

compute_pars: dict, optional [default: self.compute_pars] Dict of parameters to pass to KerrSpinSeq if a mode sequence needs to be computed from scratch.

Returns

KerrSpinSeq The mode, if it is in the cache, on disk, or has been computed from scratch. If the mode is not available and compute_if_not_found is false, return None.

write_all(*self*)
Write all of the modes in the cache to disk.

TODO: Take an overwrite argument which will force overwrite or not.

qnm.cached.**build_package_default_cache**(*ksc*)
Compute the standard list of modes that this package promises to have in its cache.

This method is intended to be used for building the modes from scratch in a predictable way. If modes are available on disk then there will be no computation, simply loading all the default modes.

Parameters

ksc: KerrSeqCache The cache that will hold the modes we are about to compute.

Returns

KerrSeqCache The updated cache.

qnm.cached.**download_data**(*overwrite=False*)
Fetch and decompress tarball of precomputed spin sequences from the web.

Parameters

overwrite: bool, optional [default: False] If there is already a tarball on disk, this flag controls whether or not it is overwritten.

qnm.cached.**get_cachedir**()
Return the location of the cache directory. This follows a pattern similar to matplotlib's treatment of config/cache dirs.

The directory is chosen as follows: 1. If the QNMCACHEDIR environment variable is supplied, choose that. 2a. On Linux, follow the XDG specification and look first in

$XDG_CACHE_HOME, if defined, or *$HOME/.cache*.

2b. On other platforms, choose *$HOME/.qnm*. 3. If the chosen directory exists and is writable, use that as the

configuration directory.

4. A writable directory could not be found; return None.

Returns

pathlib.Path object or None

qnm.cached.**get_home**()
Return the user's home directory. If the user's home directory cannot be found, return None.

qnm.cached.**load_cached_mode**(*s*, *l*, *m*, *n*)
Read a KerrSpinSeq from disk.

Path is determined by `mode_pickle_path(s, l, m, n)()`.

Parameters

s: int Spin-weight of field of interest.

l: int Multipole number of mode. l >= angular.l_min(s, m)

m: int Azimuthal number of mode.

n: int Overtone number of mode.

Returns

KerrSpinSeq The mode, if it exists. Otherwise None.

qnm.cached.**mode_pickle_path**(*s*, *l*, *m*, *n*)
Construct the path to a pickle file for the mode (s, l, m, n)

Parameters

s: int Spin-weight of field of interest.

l: int Multipole number of mode. l >= angular.l_min(s, m)

m: int Azimuthal number of mode.

n: int Overtone number of mode.

Returns

pathlib.Path object or None *<cachedir>/data/s<s>_l<l>_m<m>_n<n>.pickle*

qnm.cached.**write_mode**(*spin_seq*, *pickle_path=None*)
Write an instance of KerrSpinSeq to disk.

Parameters

spin_seq: KerrSpinSeq The mode to write to disk.

pickle_path: string or pathlib.Path, optional [default: None] Path to file to write. If None, get the path from *mode_pickle_path()*.

Raises

TODO

2.4 qnm.contfrac

Infinite continued fractions via Lentz's method.

TODO Documentation.

Functions

lentz(a, b[, tol, N_min, N_max, tiny])	Compute a continued fraction via modified Lentz's method.

Continued on next page

Table 7 – continued from previous page

lentz_gen(a, b[, tol, N_min, N_max, tiny])	Compute a continued fraction via modified Lentz's method, using generators rather than functions.

qnm.contfrac.**lentz**(*a*, *b*, *tol=1e-10*, *N_min=0*, *N_max=inf*, *tiny=1e-30*)

Compute a continued fraction via modified Lentz's method.

This implementation is by the book [1].

Parameters

a: callable returning numeric.

b: callable returning numeric.

tol: float [default: 1.e-10] Tolerance for termination of evaluation.

N_min: int [default: 0] Minimum number of iterations to evaluate.

N_max: int or comparable [default: np.Inf] Maximum number of iterations to evaluate.

tiny: float [default: 1.e-30] Very small number to control convergence of Lentz's method when there is cancellation in a denominator.

Returns

(float, float, int) The first element of the tuple is the value of the continued fraction. The second element is the estimated error. The third element is the number of iterations.

References

[1]

Examples

Compute the square root of two using continued fractions:

```
>>> from qnm.contfrac import lentz
>>> def rt2b(n):
...     if (n==0):
...         return 1
...     return 2
...
>>> def rt2a(n): return 1
>>> lentz(rt2a, rt2b)
(1.4142135623638004, 4.488287519421874e-11, 14)
```

Compute phi:

```
>>> phia = rt2a
>>> phib = rt2a
>>> lentz(phia, phib)
(1.6180339887802424, 6.785971784495359e-11, 25)
```

Compute pi:

```
>>> def pia(n):
...     if (n==1):
...         return 4.
...     return (n-1.)*(n-1.)
...
>>> def pib(n):
...     if (n==0):
...         return 0.
...     return 2*n-1.
...
>>> lentz(pia, pib, tol=1.e-15)
(3.1415926535897922, 8.881784197001252e-16, 21)
```

Compute e:

```
>>> def e_a(n):
...     if (n==1):
...         return 1.
...     return (n-1.)
...
>>> def e_b(n):
...     if (n==0):
...         return 2.
...     return n
...
>>> lentz(e_a, e_b, tol=1.e-15)
(2.7182818284590464, 3.330669738754696e-16, 16)
```

cotan(1):

```
>>> def cot1_a(n):
...     return -1.
...
>>> def cot1_b(n):
...     return 2.*n+1.
...
>>> lentz(cot1_a, cot1_b, tol=1.e-15)
(0.6420926159343306, 1.1102230246251565e-16, 9)
```

qnm.contfrac.**lentz_gen**(*a*, *b*, *tol=1e-10*, *N_min=0*, *N_max=inf*, *tiny=1e-30*)

Compute a continued fraction via modified Lentz's method, using generators rather than functions.

This implementation is by the book [1].

Parameters

a: generator yielding numeric.

b: generator yielding numeric.

tol: float [default: 1.e-10] Tolerance for termination of evaluation.

N_min: int [default: 0] Minimum number of iterations to evaluate.

N_max: int or comparable [default: np.Inf] Maximum number of iterations to evaluate.

tiny: float [default: 1.e-30] Very small number to control convergence of Lentz's method when there is cancellation in a denominator.

Returns

(float, float, int) The first element of the tuple is the value of the continued fraction. The second element is the estimated error. The third element is the number of iterations.

References

[1]

Examples

Use generators to compute the square root of 2:

```
>>> from qnm.contfrac import lentz_gen
>>> import itertools
>>> def rt2b_g():
...     yield 1
...     for x in itertools.repeat(2):
...         yield x
...
>>> def rt2a_g():
...     for x in itertools.repeat(1):
...         yield x
...
>>> lentz_gen(rt2a_g(), rt2b_g())
(1.4142135623638004, 4.488287519421874e-11, 14)
```

See the documentation for *lentz()* for more examples.

2.5 qnm.nearby

Find a nearby root of the coupled radial/angular Teukolsky equations.

TODO Documentation.

Classes

NearbyRootFinder(*args, **kwargs)	Object to find and store results from simultaneous roots of radial and angular QNM equations, following the Leaver and Cook-Zalutskiy approach.

class qnm.nearby.**NearbyRootFinder**(**args*, ***kwargs*)

Object to find and store results from simultaneous roots of radial and angular QNM equations, following the Leaver and Cook-Zalutskiy approach.

Parameters

a: float [default: 0.] Dimensionless spin of black hole, 0 <= a < 1.

s: int [default: -2] Spin of field of interest

m: int [default: 2] Azimuthal number of mode of interest

A_closest_to: complex [default: 4.+0.j] Complex value close to desired separation constant. This is intended for tracking the l-number of a sequence starting from the analytically-

known value at a=0

l_max: int [default: 20] Maximum value of l to include in the spherical-spheroidal matrix for finding separation constant and mixing coefficients. Must be sufficiently larger than l of interest that angular spectral method can converge. The number of l's needed for convergence depends on a.

omega_guess: complex [default: .5-.5j] Initial guess of omega for root-finding

tol: float [default: sqrt(double epsilon)] Tolerance for root-finding omega

cf_tol: float [defailt: 1e-10] Tolerance for continued fraction calculation

n_inv: int [default: 0] Inversion number of radial infinite continued fraction, which selects overtone number of interest

Nr: int [default: 300] Truncation number of radial infinite continued fraction. Must be sufficiently large for convergence.

Nr_min: int [default: 300] Floor for Nr (for dynamic control of Nr)

Nr_max: int [default: 4000] Ceiling for Nr (for dynamic control of Nr)

r_N: complex [default: 1.] Seed value taken for truncation of infinite continued fraction. UNUSED, REMOVE

Methods

__call__(self, x)	Internal function for usage with optimize.root, for an instance of this class to act like a function for root-finding.
clear_results(self)	Clears the stored results from last call of *do_solve()*
do_solve(self)	Try to find a root of the continued fraction equation, using the parameters that have been set in *set_params()*.
get_cf_err(self)	Return the continued fraction error and the number of iterations in the last evaluation of the continued fraction.
set_params(self, *args, **kwargs)	Set the parameters for root finding.
set_poles(self[, poles])	Set poles to multiply error function.

__call__(*self*, *x*)
Internal function for usage with optimize.root, for an instance of this class to act like a function for root-finding. optimize.root only works with reals so we pack and unpack complexes into float[2]

clear_results(*self*)
Clears the stored results from last call of *do_solve()*

do_solve(*self*)
Try to find a root of the continued fraction equation, using the parameters that have been set in *set_params()*.

get_cf_err(*self*)
Return the continued fraction error and the number of iterations in the last evaluation of the continued fraction.

Returns

cf_err: float

n_frac: int

set_params (*self*, **args*, ***kwargs*)
Set the parameters for root finding. Parameters are described in the class documentation. Finally calls *clear_results()*.

set_poles (*self*, *poles=[]*)
Set poles to multiply error function.

Parameters

poles: array_like as complex numbers [default: []]

2.6 qnm.radial

Solve the radial Teukolsky equation via Leaver's method.

Functions

D_coeffs(omega, a, s, m, A)	The D_0 through D_4 coefficients that enter into the radial infinite continued fraction, Eqs.
leaver_cf_inv_lentz(omega, a, s, m, A, n_inv)	Compute the n_inv inversion of the infinite continued fraction for solving the radial Teukolsky equation, using modified Lentz's method.
leaver_cf_inv_lentz_old(omega, a, s, m, A, n_inv)	Legacy function.
leaver_cf_trunc_inversion(omega, a, s, m, A, ...)	Legacy function.
sing_pt_char_exps(omega, a, s, m)	Compute the three characteristic exponents of the singular points of the radial Teukolsky equation.

qnm.radial.**D_coeffs** (*omega*, *a*, *s*, *m*, *A*)
The D_0 through D_4 coefficients that enter into the radial infinite continued fraction, Eqs. (31) of [1] .

Parameters

omega: complex The complex frequency in the ansatz for the solution of the radial Teukolsky equation.

a: double Spin parameter of the black hole, 0. <= a < 1 .

s: int Spin weight of the field (i.e. -2 for gravitational).

m: int Azimuthal number for the perturbation.

A: complex Separation constant between angular and radial ODEs.

Returns

array[5] of complex D_0 through D_4 .

References

[1]

qnm.radial.**leaver_cf_inv_lentz**(*omega*, *a*, *s*, *m*, *A*, *n_inv*, *tol=1e-10*, *N_min=0*, *N_max=inf*)

Compute the n_inv inversion of the infinite continued fraction for solving the radial Teukolsky equation, using modified Lentz's method. The value returned is Eq. (44) of [1].

Same as *leaver_cf_inv_lentz_old()*, but with Lentz's method inlined so that numba can speed things up.

Parameters

omega: complex The complex frequency for evaluating the infinite continued fraction.

a: float Spin parameter of the black hole, 0. <= a < 1 .

s: int Spin weight of the field (i.e. -2 for gravitational).

m: int Azimuthal number for the perturbation.

A: complex Separation constant between angular and radial ODEs.

n_inv: int Inversion number for the infinite continued fraction. Finding the nth overtone is typically most stable when n_inv = n .

tol: float, optional [default: 1.e-10] Tolerance for termination of Lentz's method.

N_min: int, optional [default: 0] Minimum number of iterations through Lentz's method.

N_max: int or comparable, optional [default: np.Inf] Maximum number of iterations for Lentz's method.

Returns

(complex, float, int) The first value (complex) is the nth inversion of the infinite continued fraction evaluated with these arguments. The second value (float) is the estimated error from Lentz's method. The third value (int) is the number of iterations of Lentz's method.

References

[1]

Examples

```
>>> from qnm.radial import leaver_cf_inv_lentz
>>> print(leaver_cf_inv_lentz(omega=.4 - 0.2j, a=0.02, s=-2, m=2, A=4.+0.j, n_
↪inv=0))
((-3.566277377049597 2-1.538871079338485j), 9.702532283649582e-11, 76)
```

qnm.radial.**leaver_cf_inv_lentz_old**(*omega*, *a*, *s*, *m*, *A*, *n_inv*, *tol=1e-10*, *N_min=0*, *N_max=inf*)

Legacy function. Same as *leaver_cf_inv_lentz()* except calling *qnm.contfrac.lentz()* with temporary functions that are defined inside this function. Numba does not speed up this type of code. However it remains here for testing purposes. See documentation for *leaver_cf_inv_lentz()* for parameters and return value.

Examples

```
>>> from qnm.radial import leaver_cf_inv_lentz_old, leaver_cf_inv_lentz
>>> print(leaver_cf_inv_lentz_old(omega=.4 - 0.2j, a=0.02, s=-2, m=2, A=4.+0.j, n_
↪inv=0))
((-3.566277377049597 2-1.538710793384 85j), 9.702532283649582e-11, 76)
```

Compare the two versions of the function:

```
>>> old = leaver_cf_inv_lentz_old(omega=.4 - 0.2j, a=0.02, s=-2, m=2, A=4.+0.j, n_
↪inv=0)
>>> new = leaver_cf_inv_lentz(omega=.4 - 0.2j, a=0.02, s=-2, m=2, A=4.+0.j, n_
↪inv=0)
>>> [ old[i]-new[i] for i in range(3)]
[0j, 0.0, 0]
```

qnm.radial.**leaver_cf_trunc_inversion**(*omega*, *a*, *s*, *m*, *A*, *n_inv*, *N=300*, *r_N=1.0*)
Legacy function.

Approximate the n_inv inversion of the infinite continued fraction for solving the radial Teukolsky equation, using N terms total for the approximation. This uses "bottom up" evaluation, and you can pass a seed value r_N to assume for the rest of the infinite fraction which has been truncated. The value returned is Eq. (44) of [1].

Parameters

omega: complex The complex frequency for evaluating the infinite continued fraction.

a: float Spin parameter of the black hole, 0. <= a < 1 .

s: int Spin weight of the field (i.e. -2 for gravitational).

m: int Azimuthal number for the perturbation.

A: complex Separation constant between angular and radial ODEs.

n_inv: int Inversion number for the infinite continued fraction. Finding the nth overtone is typically most stable when n_inv = n .

N: int, optional [default: 300] The depth where the infinite continued fraction is truncated.

r_N: float, optional [default: 1.] Value to assume for the rest of the infinite continued fraction past the point of truncation.

Returns

complex The nth inversion of the infinite continued fraction evaluated with these arguments.

References

[1]

qnm.radial.**sing_pt_char_exps**(*omega*, *a*, *s*, *m*)
Compute the three characteristic exponents of the singular points of the radial Teukolsky equation.

We want ingoing at the outer horizon and outgoing at infinity. The choice of one of two possible characteristic exponents at the inner horizon doesn't affect the minimal solution in Leaver's method, so we just pick one. Thus our choices are, in the nomenclature of [1], (ζ_+, ξ_-, η_+).

Parameters

omega: complex The complex frequency in the ansatz for the solution of the radial Teukolsky equation.

a: double Spin parameter of the black hole, 0. <= a < 1 .

s: int Spin weight of the field (i.e. -2 for gravitational).

m: int Azimuthal number for the perturbation.

Returns

(complex, complex, complex) (ζ_+, ξ_-, η_+)

References

[1]

2.7 qnm.spinsequence

Follow a QNM labeled by (s,l,m,n) as spin varies from a=0 upwards.

Classes

KerrSpinSeq(*args, **kwargs)	Object to follow a QNM up a sequence in a, starting from a=0.

class qnm.spinsequence.**KerrSpinSeq**(**args*, ***kwargs*)

Object to follow a QNM up a sequence in a, starting from a=0. Values for omega and the separation constant from one value of a are used to seed the root finding for the next value of a, to maintain continuity in a when separation constant order can change. Uses NearbyRootFinder to actually perform the root-finding.

Parameters

a_max: float [default: .99] Maximum dimensionless spin of black hole for the sequence, 0 <= a_max < 1.

delta_a: float [default: 0.005] Step size in a for following the sequence from a=0 to a_max

delta_a_min: float [default: 1.e-5] Minimum step size in a.

delta_a_max: float [default: 4.e-3] Maximum step size in a.

s: int [default: 2] Spin of field of interest

m: int [default: 2] Azimuthal number of mode of interest

l: int [default: 2] The l-number of a sequence starting from the analytically-known value at a=0

l_max: int [default: 20] Maximum value of l to include in the spherical-spheroidal matrix for finding separation constant and mixing coefficients. Must be sufficiently larger than l of interest that angular spectral method can converge. The number of l's needed for convergence depends on a.

omega_guess: complex [default: from schwarzschild.QNMDict] Initial guess of omega for root-finding

tol: float [default: sqrt(double epsilon)] Tolerance for root-finding omega

cf_tol: float [defailt: 1e-10] Tolerance for continued fraction calculation

n: int [default: 0] Overtone number of interest (sets the inversion number for infinite continued fraction in Leaver's method)

Nr: int [default: 300] Truncation number of radial infinite continued fraction. Must be sufficiently large for convergence.

Nr_min: int [default: Nr] Minimum number of terms for evaluating continued fraction.

Nr_max: int [default: 4000] Maximum number of terms for evaluating continued fraction.

r_N: complex [default: 0.j] Seed value taken for truncation of infinite continued fraction. UNUSED, REMOVE

Methods

__call__(self, a[, store, interp_only, ...])	Solve for omega, A, and C[] at a given spin a.
build_interps(self)	Build interpolating functions for omega(a) and A(a).
do_find_sequence(self)	Solve for the "spin sequence", i.e.

__call__ (*self*, *a*, *store=False*, *interp_only=False*, *resolve_if_found=False*)
Solve for omega, A, and C[] at a given spin a.

This uses the interpolants, based on the solved sequence, for initial guesses of omega(a) and A(a).

Parameters

a: float Value of spin, 0 <= a < 1.

store: bool, optional [default: False] Whether or not to save newly solved data in sequence. Warning, this can produce a slowdown if a lot of data needs to be moved.

interp_only: bool, optional [default: False] If False, use the Leaver solver to polish the interpolated guess. If True, just use the interpolated guess.

resolve_if_found: bool, optional [default: False] If False, and the value of *a* is found in the sequence, the previously-found solution is returned. If True, the Leaver solver will be used to polish the root with the current parameters for the solver.

Returns

complex, complex, complex ndarray The first element of the tuple is omega. The second element of the tuple is A. The third element of the tuple is the array of complex spherical-spheroidal decomposition coefficients. For documentation on the format of the spherical-spheroidal decomposition coefficient array, see *qnm.angular* or *qnm.angular.C_and_sep_const_closest()*.

build_interps (*self*)
Build interpolating functions for omega(a) and A(a).

This is automatically called at the end of *do_find_sequence()*.

do_find_sequence (*self*)
Solve for the "spin sequence", i.e. solve for the QNM as we go up in spin.

2.8 qnm.schwarzschild

Finding QNMs of Schwarzschild (numerically and approximately).

Schwarzschild QNMs are used as starting points for following a Kerr QNM along a "spin sequence" in *qnm.spinsequence.KerrSpinSeq*. Most end users will not need to directly use the functions in this module.

2.9 qnm.schwarzschild.approx

Analytic approximations for Schwarzschild QNMs.

The approximations implemented in this module can be used as initial guesses when numerically searching for QNM frequencies.

Functions

Schw_QNM_estimate(s, l, n)	Give either *large_overtone_expansion()* or *dolan_ottewill_expansion()*.
dolan_ottewill_expansion(s, l, n)	High l asymptotic expansion of Schwarzschild QNM frequency.
large_overtone_expansion(s, l, n)	The eikonal approximation for QNMs, valid for l >> n >> 1 .

qnm.schwarzschild.approx.**Schw_QNM_estimate**(*s*, *l*, *n*)

Give either *large_overtone_expansion()* or *dolan_ottewill_expansion()*.

The Dolan-Ottewill expansion includes terms with higher powers of the overtone number n, so it breaks down faster at high n.

Parameters

s: int Spin weight of the field of interest.

l: int Multipole number of interest.

[The m parameter is omitted because this is just for Schwarzschild.]

n: int Overtone number of interest.

Returns

complex Analytic approximation of QNM of interest.

qnm.schwarzschild.approx.**dolan_ottewill_expansion**(*s*, *l*, *n*)

High l asymptotic expansion of Schwarzschild QNM frequency.

The result of [1] is an expansion in inverse powers of L = (l+1/2). Their paper stated this series out to L^{-4}, which is how many terms are implemented here. The coefficients in this series are themselves positive powers of N = (n+1/2). This means the expansion breaks down for large N.

Parameters

s: int Spin weight of the field of interest.

l: int Multipole number of interest.

[The m parameter is omitted because this is just for Schwarzschild.]

n: int Overtone number of interest.

Returns

complex Analytic approximation of QNM of interest.

References

[1]

qnm.schwarzschild.approx.**large_overtone_expansion**(*s*, *l*, *n*)

The eikonal approximation for QNMs, valid for l >> n >> 1 .

This is just the first two terms of the series in *dolan_ottewill_expansion()*. The earliest work I know deriving this result is [1] but there may be others. In the eikonal approximation, valid when $l \gg n \gg 1$, the QNM frequency is

$$\sqrt{27}M\omega \approx (l + \frac{1}{2}) - i(n + \frac{1}{2}).$$

Parameters

s: int Spin weight of the field of interest.

l: int Multipole number of interest.

[The m parameter is omitted because this is just for Schwarzschild.]

n: int Overtone number of interest.

Returns

complex Analytic approximation of QNM of interest.

References

[1]

2.10 qnm.schwarzschild.overtonesequence

Follow a Schwarzschild QNM sequence (s,l) from n=0 upwards.

The class *SchwOvertoneSeq* makes it possible to find successive overtones (n's) of a QNM labeled by (s,l), in Schwarzschild (a=0). See its documentation for more details.

Classes

SchwOvertoneSeq(*args, **kwargs)	Object to follow a sequence of Schwarzschild overtones, starting from n=0.

class qnm.schwarzschild.overtonesequence.**SchwOvertoneSeq**(**args*, ***kwargs*)

Object to follow a sequence of Schwarzschild overtones, starting from n=0. First two overtone seeds come from approx.dolan_ottewill_expansion, and afterwards linear extrapolation on the solutions is used to seed the root finding for higher values of n. Uses *qnm.nearby.NearbyRootFinder* to actually perform the root-finding.

Parameters

n_max: int [default: 12] Maximum overtone number to search for (must be positive).

s: int [default: 2] Spin weight of field of interest.

[The m parameter is omitted because this is just for Schwarzschild.]

l: int [default: 2] The multipole number of a sequence starting from the analytically-known value at a=0 .

tol: float [default: 1e-10] Tolerance for root-finding.

Nr: int [default: 300] Truncation number of radial infinite continued fraction. Must be sufficiently large for convergence.

r_N: complex [default: 0.j] Seed value taken for truncation of infinite continued fraction.

Examples

Suppose you want the n=5 overtone for (s=-1, l=3):

```
>>> from qnm.schwarzschild.overtonesequence import SchwOvertoneSeq
>>> seq = SchwOvertoneSeq(s=-1, l=3, n_max=5)
>>> seq.find_sequence()
>>> "{:.10f}".format(seq.omega[5])
'0.5039017454-1.1703558890j'
```

Later, you want to go out to n=8:

```
>>> seq.extend(n_max=8)
>>> "{:.10f}".format(seq.omega[8])
'0.4227909294-1.9136575598j'
```

Attributes

A: float Value of the angular separation constant.

n: array of int Overtone numbers.

omega: np.array of complex The QNM frequencies along the overtone sequence, element i is overtone i.

cf_err: np.array of float Estimate of continued fraction truncation error in solving for QNM frequency.

n_frac: np.array of int Truncation number of continued fraction.

solver: NearbyRootFinder Instance of *qnm.nearby.NearbyRootFinder* that is used to find the QNMs.

Methods

extend(self[, n_max])	Extend the current overtone sequence to a greater n_max.
find_sequence(self)	Alias for *extend()*

extend(*self*, *n_max=None*)

Extend the current overtone sequence to a greater n_max.

Parameters

n_max: int, optional [default: None] If is None, use the value of self.n_max . If given, set self.n_max to this new value and proceed.

Raises

optimize.nonlin.NoConvergence If a root can't be found within a few [TODO make param?] attempted inversion numbers.

find_sequence(*self*)
Alias for *extend()*

2.11 qnm.schwarzschild.tabulated

Computing, loading, and storing tabulated Schwarzschild QNMs.

Functions

build_Schw_dict(*args, **kwargs)	Function to build a dict of Schwarzschild QNMs.
default_pickle_file()	Give the default path of the QNM dict pickle file, *<dirname of this file>/data/Schw_dict.pickle*.

Classes

QNMDict([init, dict_pickle_file])	Object for getting/holding/(pre-)computing Schwarzschild QNMs.

class qnm.schwarzschild.tabulated.**QNMDict**(*init=False*, *dict_pickle_file=PosixPath('/home/docs/checkouts/readthedo*
Object for getting/holding/(pre-)computing Schwarzschild QNMs.

This class uses the "borg" pattern, so the table of precomputed values will be shared amongst all instances of the class. A set of precomputed QNMs can be loaded/stored from a pickle file with *load_dict()* and *write_dict()*. The main interface is via the special *__call__()* method which is invoked via *object(s,l,n)*. If the QNM labeled by (s,l,n) has already been computed, it will be returned. Otherwise we try to compute it and then return it.

Parameters

init: bool, optional [default: False] Whether or not to call *load_dict()* when initializing this instance.

dict_pickle_file: string or Path, optional [default: from default_pickle_file()] Path to pickle file that holds precomputed QNMs. If the value is None, get the default from *default_pickle_file()*.

Attributes

seq_dict: dict Keys are tuples (s,l) which label an overtone sequence of QNMs. Values are instances of *qnm.schwarzschild.overtonesequence.SchwOvertoneSeq*.

loaded_from_disk: bool Whether or not any modes have been loaded from disk

Methods

__call__(self, s, l, n)	Get the Schwarzschild QNM labeled by (s,l,n).
load_dict(self[, dict_pickle_file])	Load a Schw QNM dict from disk.
write_dict(self[, dict_pickle_file])	Write the current state of the QNM dict to disk.

__call__(*self*, *s*, *l*, *n*)
Get the Schwarzschild QNM labeled by (s,l,n).

If the QNM has already been computed, immediately return that value. If the (s,l) sequence is in the dict, but n has not been computed, extend the sequence to n and return the QNM. If the (s,l) sequence is not already in the dict, add it to the dict out to overtone n.

Parameters

s: int Spin weight of the field.

l: int Multipole number of the QNM.

n: int Overtone number of the QNM.

Returns

(complex, double, int) The complex value is the QNM frequency. The double is the estimated truncation error for the continued fraction. The int is the depth of the continued fraction evaluation.

Raises

TODO

load_dict(*self*, *dict_pickle_file=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/qnm/checkouts/v0.4.0/qnm/sc*
Load a Schw QNM dict from disk.

Parameters

dict_pickle_file: string or Path [default: from default_pickle_file()] Filename for reading (or writing) dict of Schwarzschild QNMs

loaded_from_disk = False

write_dict(*self*, *dict_pickle_file=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/qnm/checkouts/v0.4.0/qnm/s*
Write the current state of the QNM dict to disk.

Parameters

dict_pickle_file: string [default: from default_pickle_file()] Filename for reading (or writing) dict of Schwarzschild QNMs

qnm.schwarzschild.tabulated.**build_Schw_dict**(**args*, ***kwargs*)
Function to build a dict of Schwarzschild QNMs.

Loops over values of (s,l), using SchwOvertoneSeq to find sequences in n.

TODO Documentation

Parameters

s_arr: [int] [default: [-2, -1, 0]] Array of s values to run over.

n_max: int [default: 20] Maximum overtone number to run over (inclusive).

l_max: int [default: 20] Maximum angular harmonic number to run over (inclusive).

tol: float [default: 1e-10] Tolerance to pass to SchwOvertoneSeq.

Returns

dict A dict with tuple keys (s,l,n). The value at d[s,l,n] is a tuple (omega, cf_err, n_frac) where omega is the frequency omega_{s,l,n}, cf_err is the estimated truncation error for the continued fraction, and n_frac is the depth of the continued fraction evaluation.

qnm.schwarzschild.tabulated.**default_pickle_file**()
Give the default path of the QNM dict pickle file, *<dirname of this file>/data/Schw_dict.pickle*.

Returns

Path object <*dirname of this file*>*/data/Schw_dict.pickle*

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] WH Press, SA Teukolsky, WT Vetterling, BP Flannery, "Numerical Recipes," 3rd Ed., Cambridge University Press 2007, ISBN 0521880688, 9780521880688 .

[1] WH Press, SA Teukolsky, WT Vetterling, BP Flannery, "Numerical Recipes," 3rd Ed., Cambridge University Press 2007, ISBN 0521880688, 9780521880688 .

[1] GB Cook, M Zalutskiy, "Gravitational perturbations of the Kerr geometry: High-accuracy study," Phys. Rev. D 90, 124021 (2014), https://arxiv.org/abs/1410.7698 .

[1] GB Cook, M Zalutskiy, "Gravitational perturbations of the Kerr geometry: High-accuracy study," Phys. Rev. D 90, 124021 (2014), https://arxiv.org/abs/1410.7698 .

[1] GB Cook, M Zalutskiy, "Gravitational perturbations of the Kerr geometry: High-accuracy study," Phys. Rev. D 90, 124021 (2014), https://arxiv.org/abs/1410.7698 .

[1] GB Cook, M Zalutskiy, "Gravitational perturbations of the Kerr geometry: High-accuracy study," Phys. Rev. D 90, 124021 (2014), https://arxiv.org/abs/1410.7698 .

[1] SR Dolan, AC Ottewill, "On an expansion method for black hole quasinormal modes and Regge poles," CQG 26 225003 (2009), https://arxiv.org/abs/0908.0329 .

[1] V Ferrari, B Mashhoon, "New approach to the quasinormal modes of a black hole," Phys. Rev. D 30, 295 (1984)

Python Module Index

q

qnm, 9
qnm.angular, 10
qnm.cached, 13
qnm.contfrac, 16
qnm.nearby, 19
qnm.radial, 21
qnm.schwarzschild, 25
qnm.schwarzschild.approx, 26
qnm.schwarzschild.overtonesequence, 27
qnm.schwarzschild.tabulated, 29
qnm.spinsequence, 24

Index

Symbols

__call__() (*qnm.cached.KerrSeqCache method*), 14
__call__() (*qnm.nearby.NearbyRootFinder method*), 20
__call__() (*qnm.schwarzschild.tabulated.QNMDict method*), 29
__call__() (*qnm.spinsequence.KerrSpinSeq method*), 25

B

build_interps() (*qnm.spinsequence.KerrSpinSeq method*), 25
build_package_default_cache() (*in module qnm.cached*), 15
build_Schw_dict() (*in module qnm.schwarzschild.tabulated*), 30

C

C_and_sep_const_closest() (*in module qnm.angular*), 11
clear_results() (*qnm.nearby.NearbyRootFinder method*), 20

D

D_coeffs (*in module qnm.radial*), 21
default_pickle_file() (*in module qnm.schwarzschild.tabulated*), 30
do_find_sequence() (*qnm.spinsequence.KerrSpinSeq method*), 25
do_solve() (*qnm.nearby.NearbyRootFinder method*), 20
dolan_ottewill_expansion() (*in module qnm.schwarzschild.approx*), 26
download_data() (*in module qnm*), 10
download_data() (*in module qnm.cached*), 15

E

ells (*in module qnm.angular*), 12
extend() (*qnm.schwarzschild.overtonesequence.SchwOvertoneSeq method*), 28

F

find_sequence() (*qnm.schwarzschild.overtonesequence.SchwOvertone method*), 28

G

get_cachedir() (*in module qnm.cached*), 15
get_cf_err() (*qnm.nearby.NearbyRootFinder method*), 20
get_home() (*in module qnm.cached*), 15
give_M_matrix_elem_ufunc() (*in module qnm.angular*), 12

K

KerrSeqCache (*class in qnm.cached*), 14
KerrSpinSeq (*class in qnm.spinsequence*), 24

L

l_min (*in module qnm.angular*), 12
large_overtone_expansion() (*in module qnm.schwarzschild.approx*), 26
leaver_cf_inv_lentz (*in module qnm.radial*), 21
leaver_cf_inv_lentz_old() (*in module qnm.radial*), 22
leaver_cf_trunc_inversion() (*in module qnm.radial*), 23
lentz() (*in module qnm.contfrac*), 17
lentz_gen() (*in module qnm.contfrac*), 18
load_cached_mode() (*in module qnm.cached*), 15
load_dict() (*qnm.schwarzschild.tabulated.QNMDict method*), 30
loaded_from_disk (*qnm.schwarzschild.tabulated.QNMDict attribute*), 30

M

M_matrix (*in module qnm.angular*), 11
M_matrix_elem (*in module qnm.angular*), 11

M_matrix_old() (*in module qnm.angular*), 12
mode_pickle_path() (*in module qnm.cached*), 16
modes_cache (*in module qnm*), 10

N

NearbyRootFinder (*class in qnm.nearby*), 19

Q

qnm (*module*), 9
qnm.angular (*module*), 10
qnm.cached (*module*), 13
qnm.contfrac (*module*), 16
qnm.nearby (*module*), 19
qnm.radial (*module*), 21
qnm.schwarzschild (*module*), 25
qnm.schwarzschild.approx (*module*), 26
qnm.schwarzschild.overtonesequence (*module*), 27
qnm.schwarzschild.tabulated (*module*), 29
qnm.spinsequence (*module*), 24
QNMDict (*class in qnm.schwarzschild.tabulated*), 29

S

Schw_QNM_estimate() (*in module qnm.schwarzschild.approx*), 26
SchwOvertoneSeq (*class in qnm.schwarzschild.overtonesequence*), 27
sep_const_closest() (*in module qnm.angular*), 12
sep_consts() (*in module qnm.angular*), 13
set_params() (*qnm.nearby.NearbyRootFinder method*), 21
set_poles() (*qnm.nearby.NearbyRootFinder method*), 21
sing_pt_char_exps (*in module qnm.radial*), 23
swsphericalh_A (*in module qnm.angular*), 13

W

write_all() (*qnm.cached.KerrSeqCache method*), 15
write_dict() (*qnm.schwarzschild.tabulated.QNMDict method*), 30
write_mode() (*in module qnm.cached*), 16