
qnm Documentation

Release 0.2.7

Leo C. Stein

May 17, 2019

Contents:

1	Welcome to qnm	3
1.1	Installation	3
1.2	Dependencies	3
1.3	Documentation	4
1.4	Usage	4
1.5	Spherical-spheroidal decomposition	6
1.6	Credits	7
2	Package API	9
2.1	qnm	9
2.2	qnm.angular	10
2.3	qnm.cached	13
2.4	qnm.contfrac	17
2.5	qnm.nearby	19
2.6	qnm.radial	21
2.7	qnm.spinsequence	25
2.8	qnm.schwarzschild	27
2.9	qnm.schwarzschild.approx	27
2.10	qnm.schwarzschild.overtonesequence	29
2.11	qnm.schwarzschild.tabulated	30
3	Indices and tables	33
	Bibliography	35
	Python Module Index	37

For a quick start, take a look at the [usage section](#) of the Welcome node.

[github](#) [PyPI](#) [version](#) [DOI](#) [license](#) [Documentation](#) [Status](#)

CHAPTER 1

Welcome to qnm

Python implementation of the [Cook-Zalutskiy spectral approach](#) to computing Kerr quasinormal frequencies (QNMs).

With this python package, you can compute the QNMs labeled by different (s, l, m, n) , at a desired dimensionless spin parameter $0 \leq a < 1$. 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 number of low-lying modes ($s=-2$ and $s=-1$, all $l \leq 8$, all $n \leq 7$). 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`

1.2 Dependencies

All of these can be installed through pip or conda.

- `numpy`
- `scipy`
- `numba`
- `tqdm` (just for `qnm.download_data()` progress)

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.cached.KerrSeqCache` 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).

```
ksc = qnm.cached.KerrSeqCache(init_schw=True) # Only need init_schw once
mode_seq = ksc(s=-2, l=2, m=2, n=0)
omega, A, C = mode_seq(a=0.68)
print(omega)
# (0.5239751042900845-0.08151262363119974j)
```

Calling a spin sequence with `mode_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 simple examples:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('text', usetex = True)

s, l, m = (-2, 2, 2)
mode_list = [(s, l, m, n) for n in np.arange(0,7)]
modes = {}
for ind in mode_list:
    modes[ind] = ksc(*ind)

plt.figure(figsize=(16,8))
```

(continues on next page)

(continued from previous page)

```

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

modestr = "{},{}, {}, n".format(s, l, m)
plt.xlabel(r'$\text{Re}[\omega_{\text{' + modestr + r'}}]$', fontsize=16)
plt.ylabel(r'$\text{Im}[\omega_{\text{' + modestr + r'}}]$', fontsize=16)
plt.gca().tick_params(labelsize=16)
plt.gca().invert_yaxis()

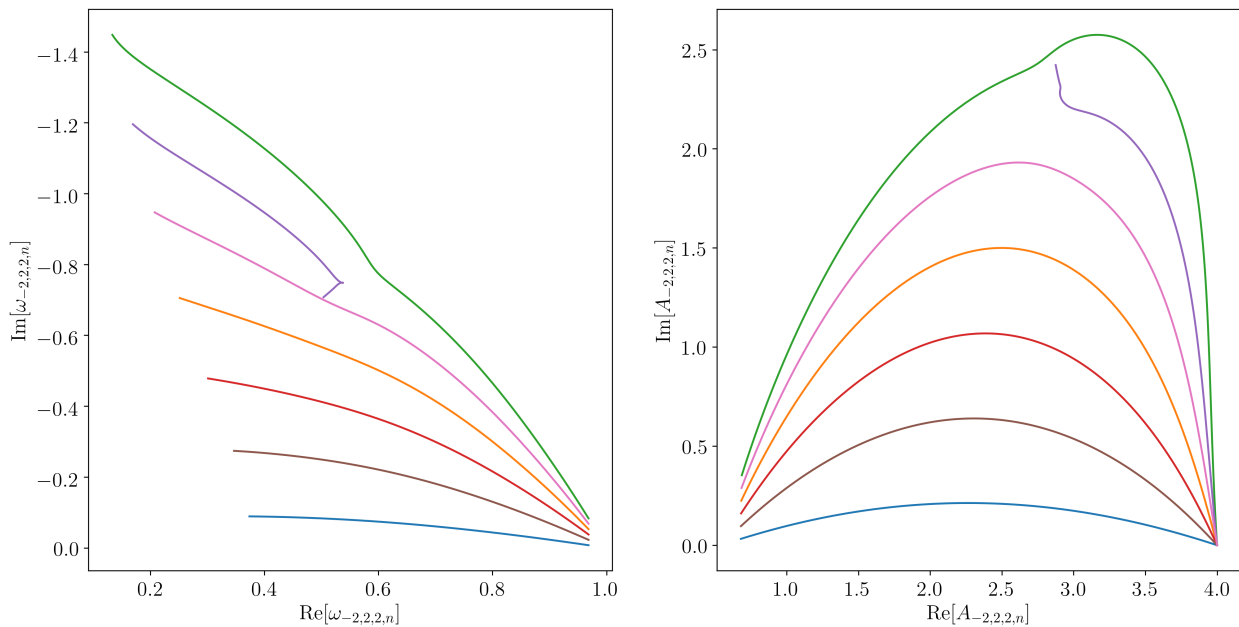
plt.subplot(1, 2, 2)
for mode, seq in modes.items():
    plt.plot(np.real(seq.A), np.imag(seq.A))

plt.xlabel(r'$\text{Re}[A_{\text{' + modestr + r'}}]$', fontsize=16)
plt.ylabel(r'$\text{Im}[A_{\text{' + modestr + r'}}]$', fontsize=16)
plt.gca().tick_params(labelsize=16)

plt.show()

```

Which results in the following figure:



example_22n

plot

```

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

plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)

```

(continues on next page)

(continued from previous page)

```

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

modestr = "{},{},m,0".format(s,l)
plt.xlabel(r'$\text{Re}[\omega_{\text{' + modestr + r'}}]$', fontsize=16)
plt.ylabel(r'$\text{Im}[\omega_{\text{' + modestr + r'}}]$', fontsize=16)
plt.gca().tick_params(labelsize=16)
plt.gca().invert_yaxis()

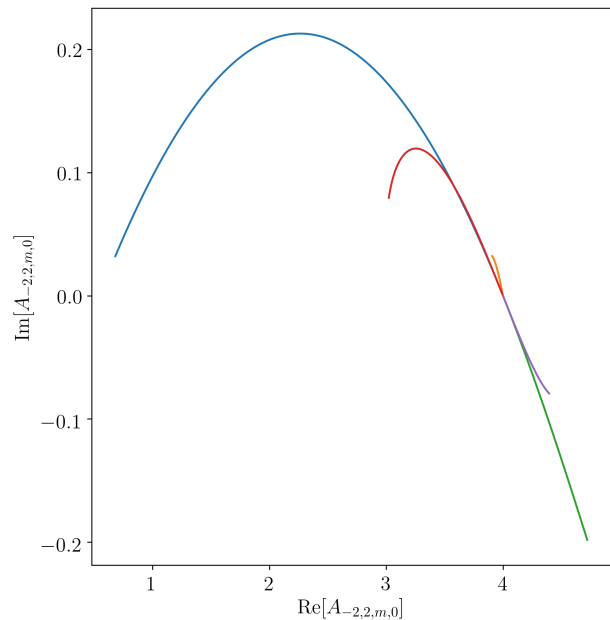
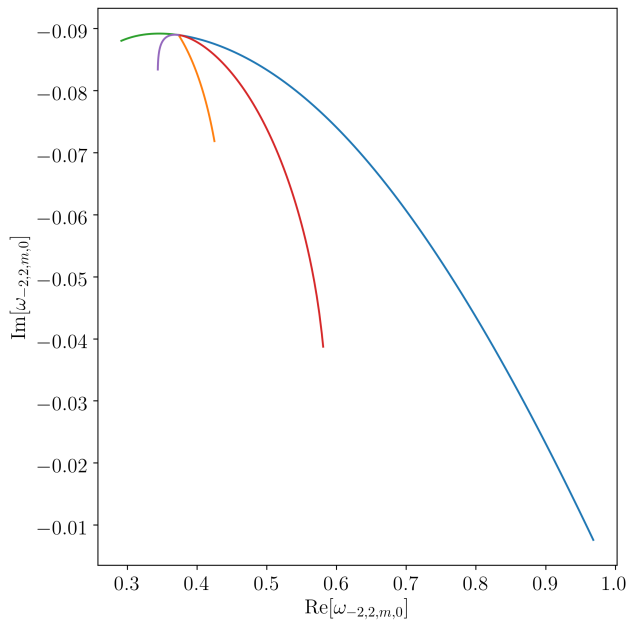
plt.subplot(1, 2, 2)
for mode, seq in modes.items():
    plt.plot(np.real(seq.A), np.imag(seq.A))

plt.xlabel(r'$\text{Re}[A_{\text{' + modestr + r'}}]$', fontsize=16)
plt.ylabel(r'$\text{Im}[A_{\text{' + modestr + r'}}]$', fontsize=16)
plt.gca().tick_params(labelsize=16)

plt.show()

```

Which results in the following figure:



example_2m0

plot

1.5 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),

equation

Here $\min=\max(|m|,|l|)$ and \max can be chosen at run time. The C coefficients are returned as a complex ndarray, with

the zeroth element corresponding to min.

1.6 Credits

The code is developed and maintained by [Leo C. Stein](#).

CHAPTER 2

Package API

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

TODO More documentation

2.1.1 Examples

```
>>> import qnm
>>> # qnm.download_data() # Only need to do this once
>>> ksc = qnm.cached.KerrSeqCache(init_schw=True) # Only need init_schw once per_
↪ session
>>> mode_seq = ksc(s=-2,l=2,m=2,n=0)
>>> omega, A, C = mode_seq(a=0.68)
>>> print(omega)
(0.5239751042900845-0.08151262363119974j)
```

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 $\min=\max(|m|,|s|)$ (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()`.

TODO More documentation.

Functions

<code>C_and_sep_const_closest(A0, s, c, m, l_max)</code>	Get a single eigenvalue and eigenvector of decomposition matrix, where the eigenvalue is closest to some guess A0.
<code>M_matrix_old(s, c, m, l_max)</code>	Legacy function.
<code>give_M_matrix_elem_ufunc(s, c, m)</code>	Gives ufunc that implements matrix elements of the spherical-spheroidal decomposition matrix.
<code>sep_const_closest(A0, s, c, m, l_max)</code>	Gives the separation constant that is closest to A0.
<code>sep_consts(s, c, m, l_max)</code>	Finds eigenvalues of decomposition matrix, i.e.

`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 A_0 . 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)`

Gives ufunc that implements matrix elements of the spherical-spheroidal decomposition matrix. This function is used by `M_matrix_old()` and can be considered legacy.

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, A_s .

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. 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.

TODO More documentation.

Functions

<code>build_package_default_cache(ksc)</code>	Compute the standard list of modes that this package promises to have in its cache.
<code>download_data([overwrite])</code>	Fetch and decompress tarball of precomputed spin sequence from the web.
<code>load_cached_mode(s, l, m, n)</code>	Read a KerrSpinSeq from disk.
<code>mode_pickle_path(s, l, m, n)</code>	Construct the path to a pickle file for the mode (s, l, m, n)
<code>write_mode(spin_seq[, pickle_path])</code>	Write an instance of KerrSpinSeq to disk.

Classes

<code>KerrSeqCache([init_schw, ...])</code>	High-level caching interface for getting precomputed spin sequences.
<code>TqdmUpTo([iterable, desc, total, leave, ...])</code>	Provides <code>update_to(n)</code> which uses <code>tqdm.update(delta_n)</code> .

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
>>> ksc = qnm.cached.KerrSeqCache(init_schw=True) # Only need init_schw once per_
↪ session
>>> mode_seq = ksc(s=-2, l=2, m=2, n=0)
>>> omega, A, C = mode_seq(a=0.68)
>>> print(omega)
(0.5239751042900845-0.08151262363119974j)
```

Methods

<code>__call__(self, s, l, m, n[, ...])</code>	Load a <code>qnm.spinsequence.KerrSpinSeq</code> from the cache or from disk if available.
<code>write_all(self)</code>	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. TODO maybe take param: dict of params to pass to `KerrSpinSeq`?

Parameters

s: int Spin-weight of field of interest.

l: int Multipole number of mode. $l \geq \text{angular.l_min}(s, m)$

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.

```
class qnm.cached.TqdmUpTo(iterable=None, desc=None, total=None, leave=True, file=None,
ncols=None, mininterval=0.1, maxinterval=10.0, miniters=None,
ascii=None, disable=False, unit='it', unit_scale=False, dy-
namic_ncols=False, smoothing=0.3, bar_format=None, initial=0,
position=None, postfix=None, unit_divisor=1000, write_bytes=None,
gui=False, **kwargs)
```

Provides `update_to(n)` which uses `tqdm.update(delta_n)`.

Attributes

format_dict Public API for read-only member access.

monitor

Methods

<code>clear(self[, nolock])</code>	Clear current bar display.
<code>close(self)</code>	Cleanup and (if <code>leave=False</code>) close the progressbar.
<code>display(self[, msg, pos])</code>	Use <i>self.sp</i> to display <i>msg</i> in the specified <i>pos</i> .
<code>ema(x[, mu, alpha])</code>	Exponential moving average: smoothing to give progressively lower weights to older values.
<code>external_write_mode([file, nolock])</code>	Disable tqdm within context and refresh tqdm when exits.
<code>format_interval(t)</code>	Formats a number of seconds as a clock time, [H:]MM:SS
<code>format_meter(n, total, elapsed[, ncols, ...])</code>	Return a string-based progress bar given some parameters
<code>format_num(n)</code>	Intelligent scientific notation (.3g).
<code>format_sizeof(num[, suffix, divisor])</code>	Formats a number (greater than unity) with SI Order of Magnitude prefixes.
<code>get_lock()</code>	Get the global lock.
<code>pandas(*targs, **kwargs)</code>	Registers the given <i>tqdm</i> class with
<code>refresh(self[, nolock])</code>	Force refresh the display of this bar.
<code>reset(self[, total])</code>	Resets to 0 iterations for repeated use.
<code>set_description(self[, desc, refresh])</code>	Set/modify description of the progress bar.
<code>set_description_str(self[, desc, refresh])</code>	Set/modify description without ': ' appended.
<code>set_lock(lock)</code>	Set the global lock.
<code>set_postfix(self[, ordered_dict, refresh])</code>	Set/modify postfix (additional stats) with automatic formatting based on datatype.
<code>set_postfix_str(self[, s, refresh])</code>	Postfix without dictionary expansion, similar to prefix handling.
<code>status_printer(file)</code>	Manage the printing and in-place updating of a line of characters.
<code>unpause(self)</code>	Restart tqdm timer from last print time.
<code>update(self[, n])</code>	Manually update the progress bar, useful for streams such as reading files.
<code>update_to(self[, b, bsize, tsize])</code>	<i>b</i> : int, optional
<code>write(s[, file, end, nolock])</code>	Print a message via tqdm (without overlap with bars).

moveto

update_to (*self, b=1, bsize=1, tsize=None*)

b [int, optional] Number of blocks transferred so far [default: 1].

bsize [int, optional] Size of each block (in tqdm units) [default: 1].

tsize [int, optional] Total size (in tqdm units). If [default: None] remains unchanged.

`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 sequence 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.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. $l \geq \text{angular.l_min}(s, m)$

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. $l \geq \text{angular.l_min}(s, m)$

m: int Azimuthal number of mode.

n: int Overtone number of mode.

Returns

string *<dirname of this file>/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, 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

<code>lentz(a, b[, tol, N_min, N_max, tiny])</code>	Compute a continued fraction via modified Lentz's method.
<code>lentz_gen(a, b[, tol, N_min, N_max, tiny])</code>	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, 3.3306690738754696e-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

<code>NearbyRootFinder(*args, **kwargs)</code>	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, $0 \leq a < 1$.
- 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: 1e-10]** Tolerance for root-finding
- 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.

Methods

<code>__call__(self, x, tol)</code>	Internal function for usage with optimize.root, for an instance of this class to act like a function for root-finding.
<code>clear_results(self)</code>	TODO Documentation
<code>do_solve(self)</code>	TODO Documentation
<code>get_cf_err(self)</code>	TODO Documentation
<code>set_params(self, *args, **kwargs)</code>	Set the parameters for root finding.
<code>set_poles(self[, poles])</code>	Set poles to multiply error function.

`__call__(self, x, tol)`

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*)
 TODO Documentation

do_solve (*self*)
 TODO Documentation

get_cf_err (*self*)
 TODO Documentation

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.

TODO Documentation.

Functions

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

`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, $0 \leq a < 1$.

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]

`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, $0. \leq a < 1$.

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, $0. \leq a < 1$.

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 *n*th 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))
((-3.5662773770495546-1.5388710793384461j), 9.702542314939062e-11, 76)
```

`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, $0 \leq a < 1$.

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, $0 \leq a < 1$.

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))
((-3.5662773770495546-1.5388710793384461j), 9.702542314939062e-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))
((-3.5662773770495546-1.5388710793384461j), 9.702542314939062e-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, $0 \leq a < 1$.

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 n th 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 n th inversion of the infinite continued fraction evaluated with these arguments.

References

[1]

`qnm.radial.sing_pt_char_exps` (ω, 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, $0 \leq a < 1$.

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.

TODO Documentation.

Classes

<code>KerrSpinSeq(*args, **kwargs)</code>	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 ω 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, $0 \leq a_{\text{max}} < 1$.
- 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 ω for root-finding
- tol: float [default: 1e-10]** Tolerance for root-finding
- 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. UN-USED, REMOVE

Methods

<code>__call__(self, a[, store])</code>	Solve for ω , A , and $C[]$ at a given spin a .
<code>build_interps(self)</code>	Build interpolating functions for $\omega(a)$ and $A(a)$.
<code>do_find_sequence(self)</code>	TODO Document

`__call__(self, a, store=False)`
Solve for ω , 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, $0 \leq a < 1$.

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.

Returns

complex, complex, complex ndarray The first element of the tuple is ω . 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*)

TODO Document

2.8 qnm.schwarzschild

Finding QNMs of Schwarzschild (numerically and approximately).

TODO Documentation.

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 (<i>s</i> , <i>l</i> , <i>n</i>)	Give either large_overtone_expansion() or dolan_ottewill_expansion() .
dolan_ottewill_expansion (<i>s</i> , <i>l</i> , <i>n</i>)	High <i>l</i> asymptotic expansion of Schwarzschild QNM frequency.
large_overtone_expansion (<i>s</i> , <i>l</i> , <i>n</i>)	The eikonal approximation for QNMs, valid for $l \gg n \gg 1$.

`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 $l \gg n \gg 1$.

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

<i>SchwOvertoneSeq</i> (*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()
>>> print(seq.omega[5])
(0.5039017454081958-1.1703558890487786j)
```

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

```
>>> seq.extend(n_max=8)
>>> print(seq.omega[8])
(0.4227909293895908-1.9136575597714864j)
```

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

<code>extend(self, n_max)</code>	Extend the current overtone sequence to a greater <code>n_max</code> .
<code>find_sequence(self)</code>	Alias for <code>extend()</code>

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.

TODO Documentation.

Functions

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

Classes

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

class `qnm.schwarzschild.tabulated.QNMDict` (*init=False*, *dict_pickle_file='/home/docs/checkouts/readthedocs.org/user'*)
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, 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

<code>__call__(self, s, l, n)</code>	Get the Schwarzschild QNM labeled by (s,l,n).
<code>load_dict(self[, dict_pickle_file])</code>	Load a Schw QNM dict from disk.
<code>write_dict(self[, dict_pickle_file])</code>	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='/home/docs/checkouts/readthedocs.org/user_builds/qnm/checkouts/v0.2.7/qnm/schwarzschild')`
Load a Schw QNM dict from disk.

Parameters

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

`loaded_from_disk = False`

write_dict (*self*, *dict_pickle_file*='/home/docs/checkouts/readthedocs.org/user_builds/qnm/checkouts/v0.2.7/qnm/schwarzschild')
 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

string *<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] 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)

q

- `qnm`, [9](#)
- `qnm.angular`, [10](#)
- `qnm.cached`, [13](#)
- `qnm.contfrac`, [17](#)
- `qnm.nearby`, [19](#)
- `qnm.radial`, [21](#)
- `qnm.schwarzschild`, [27](#)
- `qnm.schwarzschild.approx`, [27](#)
- `qnm.schwarzschild.overtonesequence`, [29](#)
- `qnm.schwarzschild.tabulated`, [30](#)
- `qnm.spinsequence`, [25](#)

Symbols

[__call__\(\) \(qnm.cached.KerrSeqCache method\), 14](#)
[__call__\(\) \(qnm.nearby.NearbyRootFinder method\), 20](#)
[__call__\(\) \(qnm.schwarzschild.tabulated.QNMDict method\), 31](#)
[__call__\(\) \(qnm.spinsequence.KerrSpinSeq method\), 26](#)

B

[build_interps\(\) \(qnm.spinsequence.KerrSpinSeq method\), 27](#)
[build_package_default_cache\(\) \(in module qnm.cached\), 16](#)
[build_Schw_dict\(\) \(in module qnm.schwarzschild.tabulated\), 32](#)

C

[C_and_sep_const_closest\(\) \(in module qnm.angular\), 10](#)
[clear_results\(\) \(qnm.nearby.NearbyRootFinder method\), 20](#)

D

[D_coeffs \(in module qnm.radial\), 23](#)
[D_coeffs\(\) \(in module qnm.radial\), 22](#)
[default_pickle_file\(\) \(in module qnm.schwarzschild.tabulated\), 32](#)
[do_find_sequence\(\) \(qnm.spinsequence.KerrSpinSeq method\), 27](#)
[do_solve\(\) \(qnm.nearby.NearbyRootFinder method\), 21](#)
[dolan_ottewill_expansion\(\) \(in module qnm.schwarzschild.approx\), 28](#)
[download_data\(\) \(in module qnm.cached\), 16](#)

E

[ells \(in module qnm.angular\), 11](#)

[extend\(\) \(qnm.schwarzschild.overtonesequence.SchwOvertoneSeq method\), 30](#)

F

[find_sequence\(\) \(qnm.schwarzschild.overtonesequence.SchwOvertoneSeq method\), 30](#)

G

[get_cf_err\(\) \(qnm.nearby.NearbyRootFinder method\), 21](#)
[give_M_matrix_elem_ufunc\(\) \(in module qnm.angular\), 11](#)

K

[KerrSeqCache \(class in qnm.cached\), 13](#)
[KerrSpinSeq \(class in qnm.spinsequence\), 26](#)

L

[l_min \(in module qnm.angular\), 12](#)
[large_overtone_expansion\(\) \(in module qnm.schwarzschild.approx\), 28](#)
[leaver_cf_inv_lentz \(in module qnm.radial\), 23](#)
[leaver_cf_inv_lentz\(\) \(in module qnm.radial\), 22](#)
[leaver_cf_inv_lentz_old\(\) \(in module qnm.radial\), 24](#)
[leaver_cf_trunc_inversion\(\) \(in module qnm.radial\), 24](#)
[lentz\(\) \(in module qnm.conffrac\), 17](#)
[lentz_gen\(\) \(in module qnm.conffrac\), 18](#)
[load_cached_mode\(\) \(in module qnm.cached\), 16](#)
[load_dict\(\) \(qnm.schwarzschild.tabulated.QNMDict method\), 31](#)
[loaded_from_disk \(qnm.schwarzschild.tabulated.QNMDict attribute\), 31](#)

M

[M_matrix \(in module qnm.angular\), 11](#)
[M_matrix_elem \(in module qnm.angular\), 11](#)

`M_matrix_old()` (in module *qnm.angular*), 11
`mode_pickle_path()` (in module *qnm.cached*), 16

N

`NearbyRootFinder` (class in *qnm.nearby*), 20

Q

`qnm` (module), 9
`qnm.angular` (module), 10
`qnm.cached` (module), 13
`qnm.contfrac` (module), 17
`qnm.nearby` (module), 19
`qnm.radial` (module), 21
`qnm.schwarzschild` (module), 27
`qnm.schwarzschild.approx` (module), 27
`qnm.schwarzschild.overtonesequence` (module), 29
`qnm.schwarzschild.tabulated` (module), 30
`qnm.spinsequence` (module), 25
`QNMDict` (class in *qnm.schwarzschild.tabulated*), 30

S

`Schw_QNM_estimate()` (in module *qnm.schwarzschild.approx*), 27
`SchwOvertoneSeq` (class in *qnm.schwarzschild.overtonesequence*), 29
`sep_const_closest()` (in module *qnm.angular*), 12
`sep_consts()` (in module *qnm.angular*), 12
`set_params()` (*qnm.nearby.NearbyRootFinder* method), 21
`set_poles()` (*qnm.nearby.NearbyRootFinder* method), 21
`sing_pt_char_exps` (in module *qnm.radial*), 25
`sing_pt_char_exps()` (in module *qnm.radial*), 21
`swsphericalh_A` (in module *qnm.angular*), 12

T

`TqdmUpTo` (class in *qnm.cached*), 14

U

`update_to()` (*qnm.cached.TqdmUpTo* method), 15

W

`write_all()` (*qnm.cached.KerrSeqCache* method), 14
`write_dict()` (*qnm.schwarzschild.tabulated.QNMDict* method), 31
`write_mode()` (in module *qnm.cached*), 16