GfTool

Release 0.11.0+59.g354f390

Weh Andreas

2024-04-27

CONTENTS

Release
0.11.0+59.g354f390

Date
2024-04-27

DOI
10.5281/zenodo.4744545

This reference manual details functions, modules, and objects included in *GfTool*, describing what they are and what they do.

GfTool is a collection of commonly used Green's functions and utilities. The main purpose of this module is to have a tested and thus reliable basis to do numerics. It happened to me too often, that I just made a mistake copying the Green's function and was then wondering what was wrong with my algorithm.

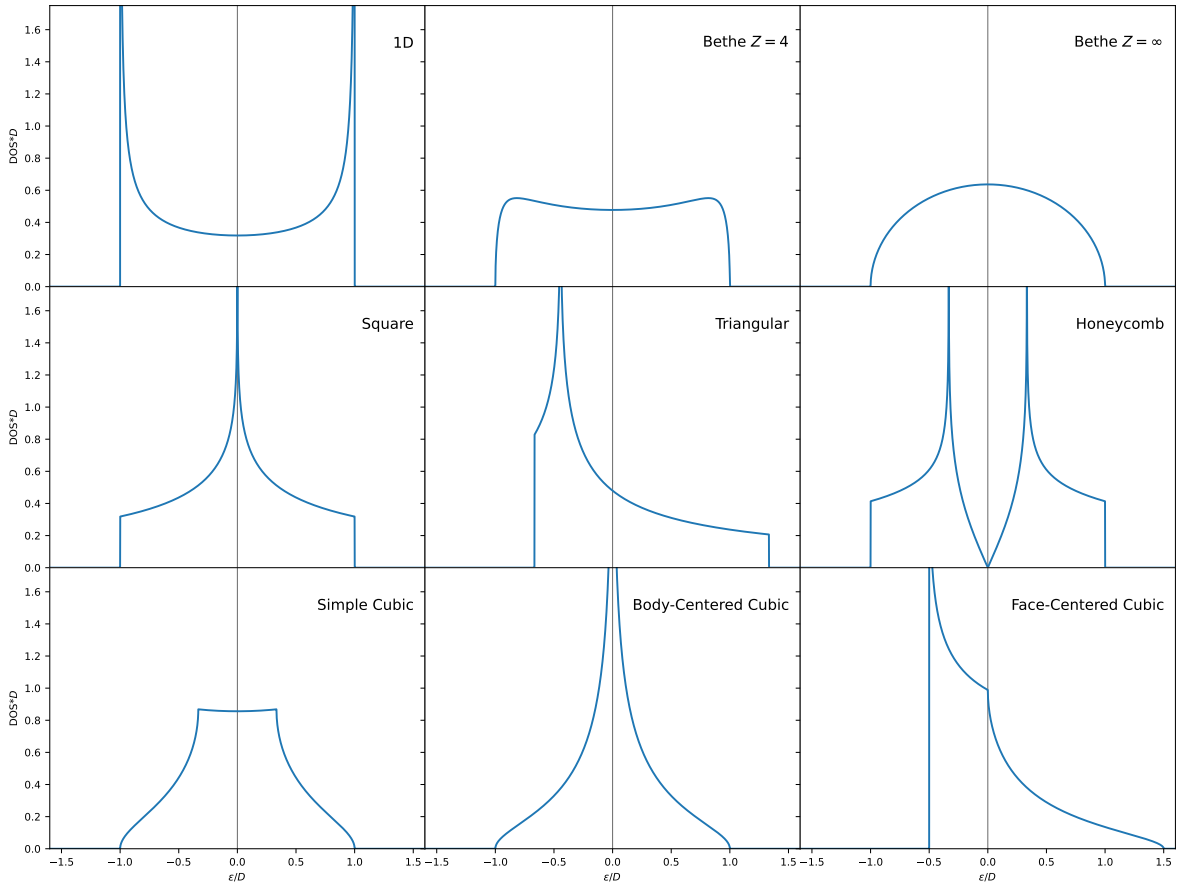


Fig. 1: *Selection* of lattice Green's functions or rather the corresponding DOSs available in *GfTool*.

The main use case of *GfTool* was DMFT and its real space generalization, in particular using CT-QMC algorithms.

CHAPTER

ONE

GETTING STARTED

1.1 GfTool's main content

gftool

- Collection of non-interacting Green's functions and spectral functions, see also the `lattice` submodule.
- Utility functions like Matsubara frequencies and Fermi functions.
- Reliable calculation of particle numbers via Matsubara sums.

cpa*/*beb

- Single site approximation to disorder.
- Diagonal disorder only (CPA) and diagonal and off-diagonal (BEB).
- Average local Green's function and component Green's functions.

fourier

- Fourier transforms from Matsubara frequencies to imaginary time and back, including the handling of high-frequencies moments (especially important for transforms from Matsubara to imaginary time).
- Laplace transform from real times to complex frequencies.

matrix

- Helper for Green's functions in matrix form.

pade

- Analytic continuation via the Padé algorithm.
  - Calculates a rational polynomial as interpolation of the data points.
  - Note: the module is out-dated, so it's not well documented at a bit awkward to use. Consider using `polepade` instead.

polepade

- Analytic continuation via a pole-based variant of the Padé algorithm.
  - Based on explicit calculation of the pole in a least-squares sense.
  - Allows including uncertainties as weights.

siam

- Basic Green's functions for the non-interacting Single Impurity Anderson model.

1.2 Installation

The package is available on PyPI:

```
$ pip install gftool
```

For conda users, *GfTool* is also available on conda-forge

```
$ conda install -c conda-forge gftool
```

Alternatively, it can be installed via GitHub. You can install it using

```
$ pip install https://github.com/DerWeh/gftools/archive/VERSION.zip
```

where *VERSION* can be a release (e.g. *0.5.1*) or a branch (e.g. *develop*). (As always, it is not advised to install it into your system Python, consider using pipenv, venv, conda, pyenv, or similar tools.) Of course, you can also clone or fork the project.

If you clone the project, you can locally build the documentation:

```
$ pip install -r requirements-doc.txt
$ python setup.py build_sphinx
```

1.3 Note on documentation

We try to follow `numpy` broadcasting rules. Many functions acting on an axis act like generalized ufuncs. In this case, a function can be called for stacked arguments instead of looping over the specific arguments.

We indicate this by argument shapes containing an ellipse e.g. *(…)* or *(…, N)*. It must be possible for all ellipses to be broadcasted against each other. A good example is the *`fourier`* module.

We calculate the Fourier transforms *`iw2tau`* for Green's functions with different on-site energies without looping:

```
>>> e_onsite = np.array([-0.5, 0, 0.5])
>>> beta = 10
>>> iws = gt.matsubara_frequencies(range(1024), beta=beta)
>>> gf_iw = gt.bethe_gf_z(iws - e_onsite[..., np.newaxis], half_bandwidth=1.0)
>>> gf_iw.shape
(3, 1024)
```

```
>>> from gftool import fourier
>>> gf_tau = fourier.iw2tau(gf_iw, beta=beta, moments=np.ones([1]))
>>> gf_tau.shape
(3, 2049)
```

The moments are automatically broadcasted. We can also explicitly give the second moments:

```
>>> moments = np.stack([np.ones([3]), e_onsite], axis=-1)
>>> gf_tau = fourier.iw2tau(gf_iw, beta=beta, moments=moments)
>>> gf_tau.shape
(3, 2049)
```

CHAPTER

TWO

TUTORIAL

This tutorial explains some of the basic functionality. Throughout the tutorial we assume you have imported *GfTool* as

```
>>> import gftool as gt
```

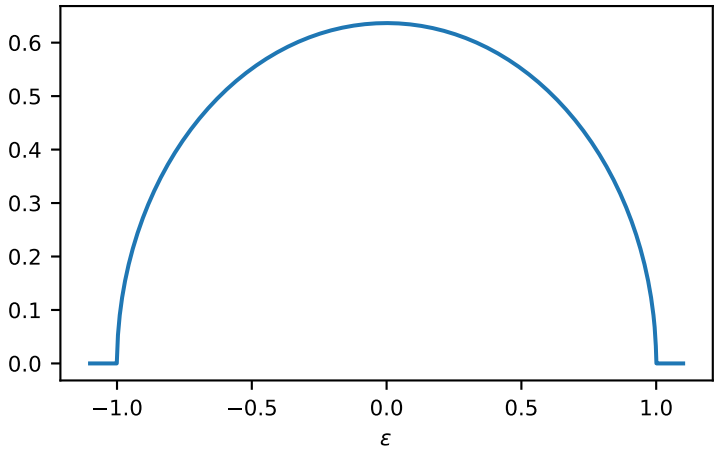
and the packages `numpy` and `matplotlib` are imported as usual

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

2.1 Lattice Green's functions

The package contains non-interacting Green's functions for some tight-binding lattices. They can be found in *gftool.lattice*. E.g. the *dos* of the Bethe lattice, *bethe*:

```
>>> ww = np.linspace(-1.1, 1.1, num=1000)
>>> dos_ww = gt.lattice.bethe.dos(ww, half_bandwidth=1.)
>>> __ = plt.plot(ww, dos_ww)
>>> __ = plt.xlabel(r"$\epsilon$")
>>> plt.show()
```

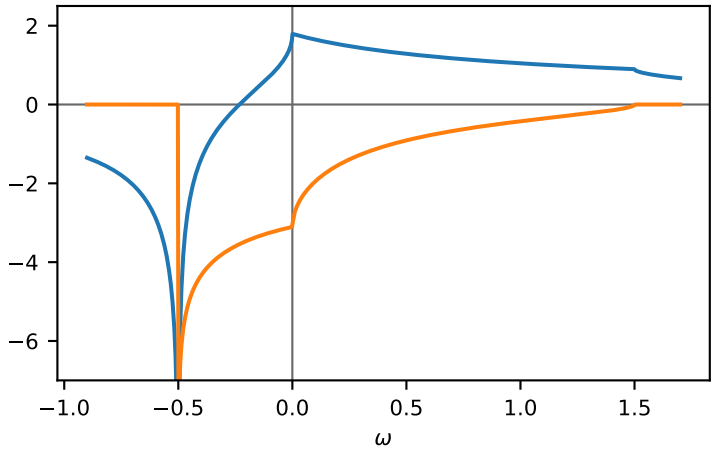


Typically, a shorthand for these functions exist in the top-level module e.g. *gftool.bethe_dos*

```
>>> gt.bethe_dos is gt.lattice.bethe.dos
True
```

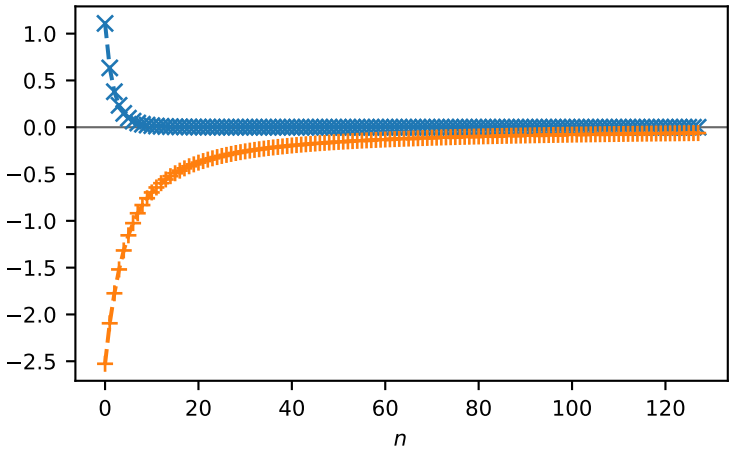
The corresponding Green's functions are also available. The Green's functions can be evaluated for any complex frequency, excluding the real axis, where it might become singular. E.g. the *gf_z* of the face-centered cubic (fcc) lattice, *fcc*, on a contour parallel to the real axis:

```
>>> ww = np.linspace(-0.9, 1.7, num=1000) + 1e-6j
>>> gf_ww = gt.lattice.fcc.gf_z(ww, half_bandwidth=1.)
>>> __ = plt.axhline(0, color="dimgray", linewidth=0.8)
>>> __ = plt.axvline(0, color="dimgray", linewidth=0.8)
>>> __ = plt.plot(ww.real, gf_ww.real)
>>> __ = plt.plot(ww.real, gf_ww.imag)
>>> __ = plt.xlabel(r"$\omega$")
>>> __ = plt.ylim(-7.0, 2.5)
>>> plt.show()
```



or on the Matsubara axis:

```
>>> beta = 50
>>> iws = gt.matsubara_frequencies(range(128), beta=beta)
>>> gf_iw = gt.lattice.fcc.gf_z(iws, half_bandwidth=1.)
>>> __ = plt.axhline(0, color="dimgray", linewidth=0.8)
>>> __ = plt.plot(gf_iw.real, "x--")
>>> __ = plt.plot(gf_iw.imag, "+--")
>>> __ = plt.xlabel("$n$")
>>> plt.show()
```



2.2 Density

We can also calculate the density (occupation number) from the imaginary axis for local Green's function. We have the relation

$$\langle n \rangle = T \sum_{\{i_n\}} \Re G(i_n)$$

To calculate the density for a given temperature from *1024* (fermionic) Matsubara frequencies we use *density_iw*:

```
>>> temperature = 0.02
>>> iws = gt.matsubara_frequencies(range(1024), beta=1./temperature)
>>> gf_iw = gt.bethe_gf_z(iws, half_bandwidth=1.)
>>> occ = gt.density_iw(iws, gf_iw, beta=1./temperature)
>>> occ
0.5
```

We can also search the chemical potential for a given occupation using *chemical_potential*. To get, e.g., the Bethe lattice at quarter filling, we write:

```
>>> occ_quarter = 0.25
>>> def bethe_occ_diff(mu):
...     """Calculate the difference to the desired occupation, note the sign."""
...     gf_iw = gt.bethe_gf_z(iws + mu, half_bandwidth=1.)
...     return gt.density_iw(iws, gf_iw, beta=1./temperature) - occ_quarter
...
>>> mu = gt.chemical_potential(bethe_occ_diff)
>>> mu
-0.406018...
```

Validate the result:

```
>>> gf_quarter_iw = gt.bethe_gf_z(iws + mu, half_bandwidth=1.)
>>> gt.density_iw(iws, gf_quarter_iw, beta=1./temperature)
0.249999...
```

2.3 Fourier transform

GfTool offers also accurate Fourier transformations between Matsubara frequencies and imaginary time for local Green's functions, see `gftool.fourier`. As a major part of the package, these functions are gu-functions. This is indicated in the docstrings via the shapes *(..., N)*. The ellipsis stands for arbitrary leading dimensions. Let's consider a simple example with magnetic splitting.

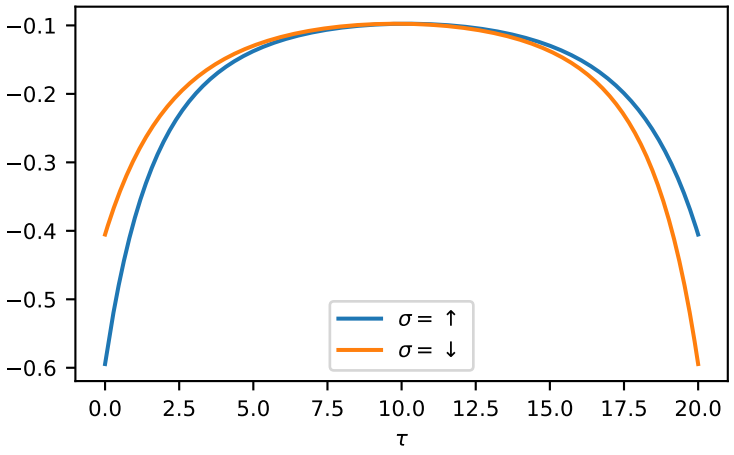
```
>>> beta = 20  # inverse temperature
>>> h = 0.3  # magnetic splitting
>>> eps = np.array([-0.5*h, +0.5*h])  # on-site energy
>>> iws = gt.matsubara_frequencies(range(1024), beta=beta)
```

We can calculate the Fourier transform using broadcasting, no need for any loops.

```
>>> gf_iw = gt.bethe_gf_z(iws + eps[:, np.newaxis], half_bandwidth=1)
>>> gf_iw.shape
(2, 1024)
>>> gf_tau = gt.fourier.iw2tau(gf_iw, beta=beta)
```

The Fourier transform generates the imaginary time Green's function on the interval $\in[0^+, ^-]$

```
>>> tau = np.linspace(0, beta, num=gf_tau.shape[-1])
>>> __ = plt.plot(tau, gf_tau[0], label=r'$\sigma=\uparrow$')
>>> __ = plt.plot(tau, gf_tau[1], label=r'$\sigma=\downarrow$')
>>> __ = plt.xlabel(r'$\tau$')
>>> __ = plt.legend()
>>> plt.show()
```



We see the asymmetry due to the magnetic field. Let's check the back transformation.

```
>>> gf_ft = gt.fourier.tau2iw(gf_tau, beta=beta)
>>> np.allclose(gf_ft, gf_iw, atol=2e-6)
True
```

Up to a certain threshold the transforms agree, they are not exact inverse transformations here. Accuracy can be improved e.g. by providing (or fitting) high-frequency moments.

2.4 Single site approximation of disorder

We also offer the single site approximation for disordered Hamiltonians, namely *cpa* and its extension to off-diagonal disorder *beb*. These methods treat substitutional disorder. A multi-component system is considered, where each lattice site is randomly occupied by one of the components. The concentration of the components is known.

2.4.1 Coherent potential approximation (CPA)

We first consider the *cpa*, where only the on-site energies depend on the component. As example, we consider a system of three components. We choose the on-site energies and concentrations (which should add to *1*), as lattice we consider a Bethe lattice with half-bandwidth *1*:

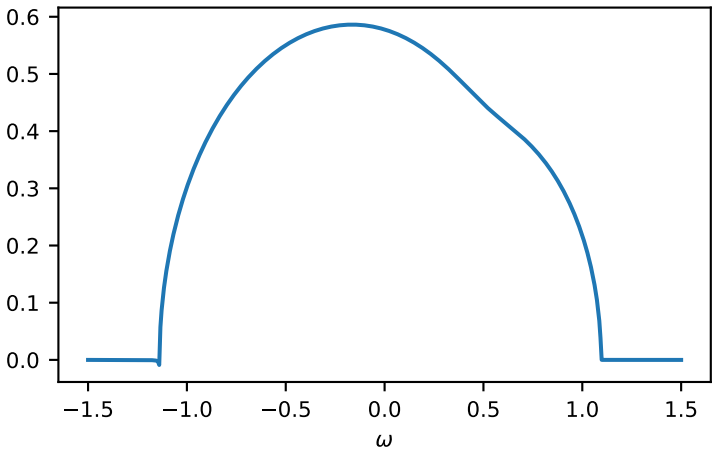
```
>>> from functools import partial
>>> e_onsite = np.array([-0.3, -0.1, 0.4])
>>> concentration = np.array([0.3, 0.5, 0.2])
>>> g0 = partial(gt.bethe_gf_z, half_bandwidth=1)
```

The average local Green's function and the component Green's functions (conditional average for local site fixed to a specific component) are calculated in CPA using an effective medium. The self-consistent effective medium is obtained via a root search using *solve_root*:

```
>>> ww = np.linspace(-1.5, 1.5, num=501) + 1e-6j
>>> self_cpa_ww = gt.cpa.solve_root(ww, e_onsite, concentration, hilbert_trafo=g0)
```

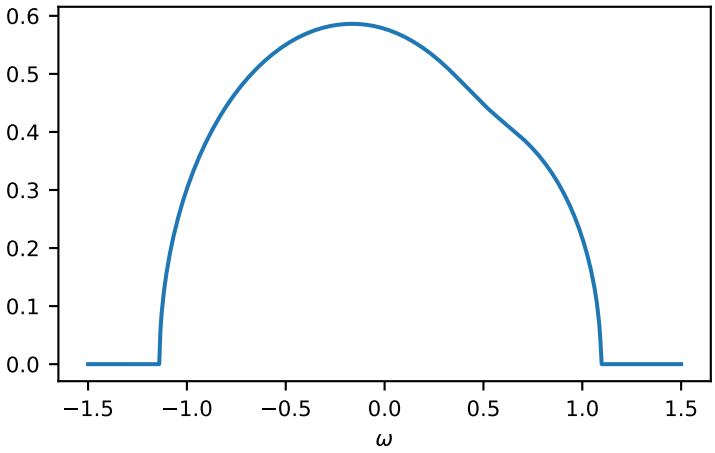
The average Green's function is

```
>>> gf_coher_ww = g0(ww - self_cpa_ww)
>>> __ = plt.plot(ww.real, -1/np.pi*gf_coher_ww.imag)
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```



For frequencies close to the real axis, issues might arise, that the conjugate solution (advanced instead of retarded) is obtained. The default *restricted=True* uses some heuristic to avoid this. In this example we see at the left band-edge, that for small imaginary part this can still fail. In this case, it is enough to just increase the accuracy of the root search. Additional keyword arguments are passed to `scipy.optimize.root`:

```
>>> self_cpa_ww = gt.cpa.solve_root(ww, e_onsite, concentration, hilbert_trafo=g0,
...                                 options=dict(fatol=1e-10))
>>> gf_coher_ww = g0(ww - self_cpa_ww)
>>> __ = plt.plot(ww.real, -1/np.pi*gf_coher_ww.imag)
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```



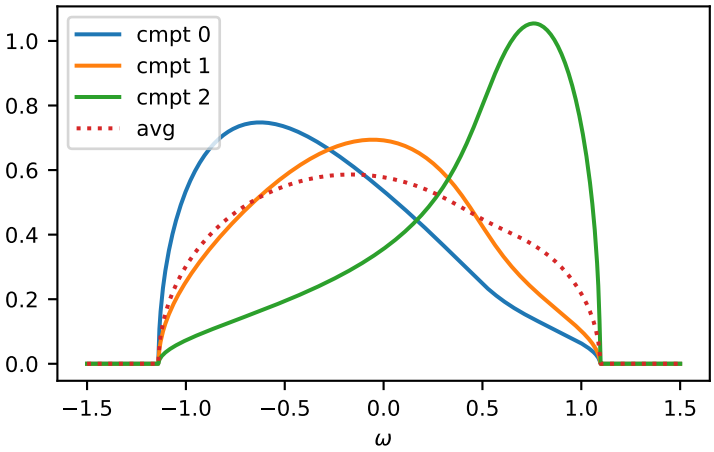
Now, everything looks fine. The component Green's functions are calculated by *gftool.cpa.gf_cmpt_z*. The law of total expectation relates the component Green's functions to the average Green's function: *np.sum(concentration*gf_cmpt_ww, axis=-1) == gf_coher_ww*:

```
>>> gf_cmpt_ww = gt.cpa.gf_cmpt_z(ww, self_cpa_ww, e_onsite, hilbert_trafo=g0)
>>> np.allclose(np.sum(concentration*gf_cmpt_ww, axis=-1), gf_coher_ww)
True
>>> for cmpt in range(3):
...     __ = plt.plot(ww.real, -1/np.pi*gf_cmpt_ww[..., cmpt].imag, label=f"cmpt
↪{cmpt}")
>>> __ = plt.plot(ww.real, -1/np.pi*gf_coher_ww.imag, linestyle=':', label="avg")
>>> __ = plt.legend()
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```

Of course, it can be calculated for any lattice Hilbert transform. Furthermore, the function is vectorized. Let's consider a *fcc* lattice, where one component has different on-site energies for up and down spin. The on-site energies can simply be stacked as 2-dimensional array. We can also take the previous self-energy as a starting guess *self_cpa_z0*:

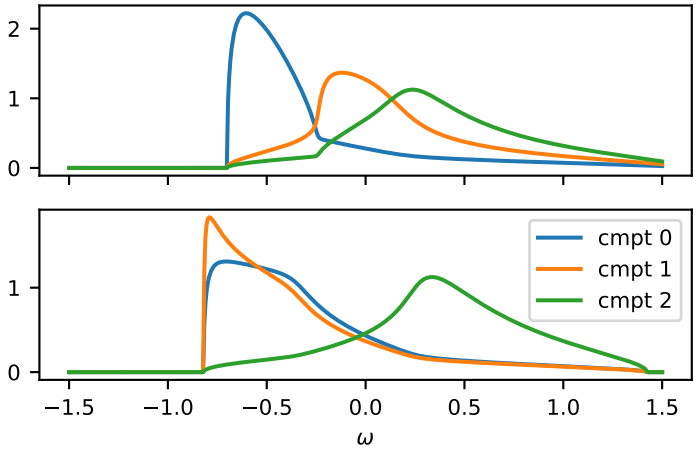
```
>>> e_onsite = np.array([[-0.3, +0.15, +0.4],
...                      [-0.3, -0.35, +0.4]])
>>> concentration = np.array([0.3, 0.5, 0.2])
>>> g0 = partial(gt.fcc_gf_z, half_bandwidth=1)
>>> self_cpa_ww = gt.cpa.solve_root(ww[:, np.newaxis], e_onsite, concentration,
...                                 hilbert_trafo=g0, options=dict(fatol=1e-8),
...                                 self_cpa_z0=self_cpa_ww[:, np.newaxis])
>>> gf_cmpt_ww = gt.cpa.gf_cmpt_z(ww[:, np.newaxis], self_cpa_ww, e_onsite, hilbert_
↪trafo=g0)
>>> __, axes = plt.subplots(nrows=2, sharex=True)
```

(continues on next page)



(continued from previous page)

```
>>> for spin, ax in enumerate(axes):
...     for cmpt in range(3):
...         __ = ax.plot(ww.real, -1/np.pi*gf_cmpt_ww[:, spin, cmpt].imag, label=f
↪"cmpt {cmpt}")
>>> __ = plt.legend()
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```



2.4.2 Blackman, Esterling, Berk (BEB)

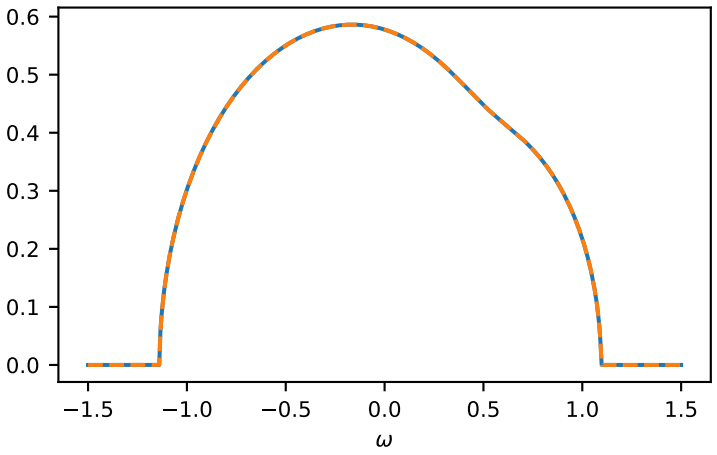
The *beb* formalism is an extension of *cpa* to off-diagonal disorder. It allows us to provide different hopping amplitudes. We have the additional parameter *hopping* which gives the relative hopping amplitudes. The *cpa* corresponds to *hopping=np.ones([N, N])*, where *N* is the number of components. The *beb* module works very similar to *cpa*: We use *solve_root* to get the effective medium, in BEB, however, the effective medium is a matrix. Next the component Green's function are calculated using *gf_loc_z*. These are, however, already multiplied by the concentration. So the average Green's function is *gf_loc_z.sum(axis=-1)*. Let's compare *cpa* and *beb*:

```
>>> from functools import partial
>>> e_onsite = np.array([-0.3, -0.1, 0.4])
>>> concentration = np.array([0.3, 0.5, 0.2])
>>> hopping = np.ones([3, 3])
>>> g0 = partial(gt.bethe_gf_z, half_bandwidth=1)
>>> ww = np.linspace(-1.5, 1.5, num=501) + 1e-5j
```

```
>>> self_cpa_ww = gt.cpa.solve_root(ww, e_onsite, concentration, hilbert_trafo=g0)
>>> gf_coher_ww = g0(ww - self_cpa_ww)
```

```
>>> self_beb_ww = gt.beb.solve_root(ww, e_onsite, concentration=concentration,
...                                 hopping=hopping, hilbert_trafo=g0)
>>> gf_loc_ww = gt.beb.gf_loc_z(ww, self_beb_ww, hopping=hopping, hilbert_trafo=g0)
```

```
>>> __ = plt.plot(ww.real, -1/np.pi*gf_coher_ww.imag, label="CPA avg")
>>> __ = plt.plot(ww.real, -1/np.pi*gf_loc_ww.sum(axis=-1).imag,
...               linestyle='--', label="BEB avg")
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```



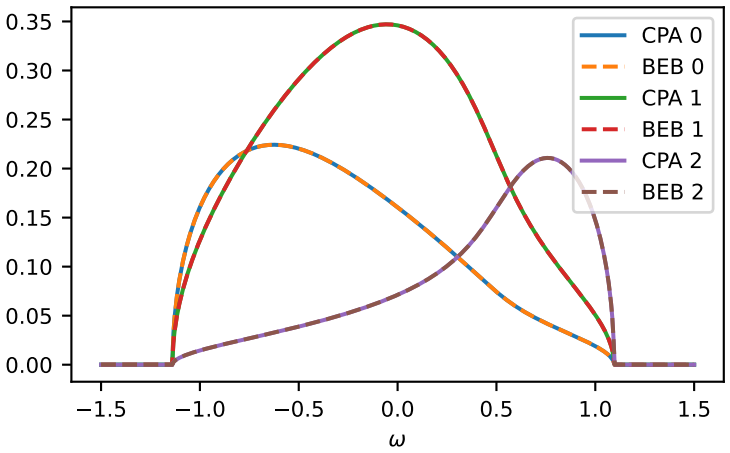
Of course, also the components match:

```
>>> gf_cmpt_ww = gt.cpa.gf_cmpt_z(ww, self_cpa_ww, e_onsite, hilbert_trafo=g0)
>>> c_gf_cmpt_ww = gf_cmpt_ww * concentration  # to compare with BEB
>>> for cmpt in range(3):
...     __ = plt.plot(ww.real, -1/np.pi*c_gf_cmpt_ww[..., cmpt].imag, label=f"CPA
```

(continues on next page)

(continued from previous page)

```
↪{cmpt}")
...     __ = plt.plot(ww.real, -1/np.pi*gf_loc_ww[..., cmpt].imag, '--', label=f"BEB
↪{cmpt}")
>>> __ = plt.legend()
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```



The relevant case is when *hopping* differs from the CPA case. Then the components can have different band-widths and also the hopping between different components can be different. Let's say we have two components 'A' and 'B'. The values *hopping=np.array([[1.0, 0.5], [0.5, 1.2]])* mean that the hopping amplitude between 'B' sites is *1.2* times the hopping amplitude between 'A' sites; the hopping amplitude from 'A' to 'B' is *0.5* times the hopping amplitude between 'A' sites.

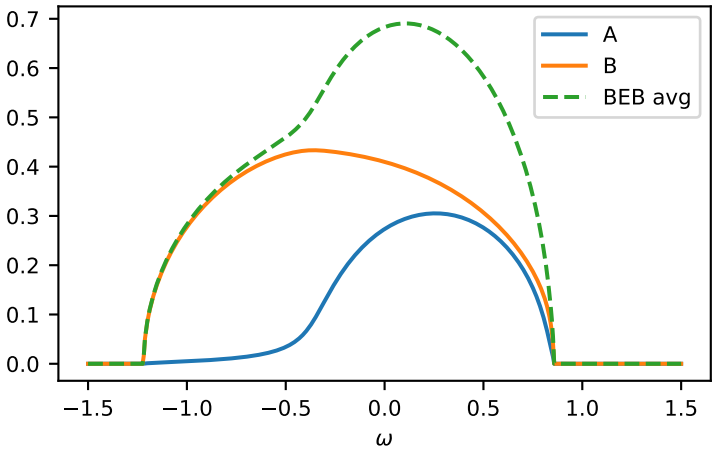
```
>>> from functools import partial
>>> e_onsite = np.array([0.2, -0.2])
>>> concentration = np.array([0.3, 0.7])
>>> hopping = np.array([[1.0, 0.5],
...                     [0.5, 1.2]])
>>> g0 = partial(gt.bethe_gf_z, half_bandwidth=1)
>>> ww = np.linspace(-1.5, 1.5, num=501) + 1e-5j
```

```
>>> self_beb_ww = gt.beb.solve_root(ww, e_onsite, concentration=concentration,
...                                 hopping=hopping, hilbert_trafo=g0)
>>> gf_loc_ww = gt.beb.gf_loc_z(ww, self_beb_ww, hopping=hopping, hilbert_trafo=g0)
>>> __ = plt.plot(ww.real, -1/np.pi*gf_loc_ww[..., 0].imag, label="A")
>>> __ = plt.plot(ww.real, -1/np.pi*gf_loc_ww[..., 1].imag, label="B")
>>> __ = plt.plot(ww.real, -1/np.pi*gf_loc_ww.sum(axis=-1).imag,
...               linestyle='--', label="BEB avg")
>>> __ = plt.legend()
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```

Additional diagnostic output is logged, you can get information on the convergence by setting:

```
>>> import logging
>>> logging.basicConfig()
```

(continues on next page)



(continued from previous page)

```
>>> logging.getLogger('gftool.beb').setLevel(logging.DEBUG)
```

2.5 Matrix Green's functions via diagonalization

The module *gftool.matrix* contains some helper functions for matrix diagonalization. A main use case is to calculate the one-particle Green's function from the resolvent. Instead of inverting the matrix for every frequency point, we can diagonalize the Hamiltonian once:

$$G(z) = [1z - H]^{-1} = [1z - UU^\dagger]^{-1} = U[z -]^{-1}U^\dagger$$

Let's consider the simple example of a 2D square lattice with nearest-neighbor hopping. The Hamiltonian can be easily constructed:

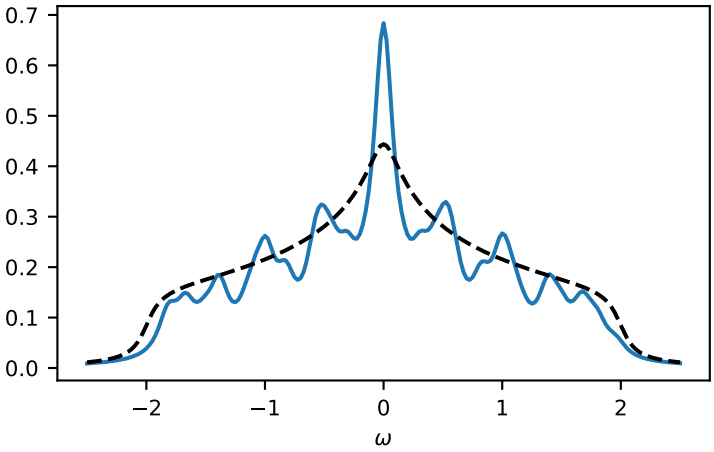
```
>>> N = 21   # system size in one dimension
>>> t = tx = ty = 0.5  # hopping amplitude
>>> hamilton = np.zeros([N]*4)
>>> diag = np.arange(N)
>>> hamilton[diag[1:], :, diag[:-1], :] = hamilton[diag[:-1], :, diag[1:], :] = -tx
>>> hamilton[:, diag[1:], :, diag[:-1]] = hamilton[:, diag[:-1], :, diag[1:]] = -ty
>>> ham_mat = hamilton.reshape(N**2, N**2)  # turn in into a matrix
```

Let's diagonalize it using the helper in *gftool.matrix* and calculated the Green's function

```
>>> dec = gt.matrix.decompose_her(ham_mat)
>>> ww = np.linspace(-2.5, 2.5, num=201) + 1e-1j  # frequency match
>>> gf_ww = dec.reconstruct(1.0/(ww[:, np.newaxis] - dec.eig))
>>> gf_ww = gf_ww.reshape(ww.size, *[N]*4)  # reshape for easy access
```

Where we used *reconstruct* to calculate the matrix product for the given diagonal matrix. Let's check the local spectral function of the central lattice site:

```
>>> __ = plt.plot(ww.real, -1.0/np.pi*gf_ww.imag[:, N//2, N//2, N//2, N//2])
>>> __ = plt.plot(ww.real, -1.0/np.pi*gt.square_gf_z(ww, half_bandwidth=4*t).imag,
...               color='black', linestyle='--')
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```



Oftentimes we are only interested in the local Green's functions and can avoid a large part of the computation, only calculating the diagonal elements. This can be done using the *kind* argument:

```
>>> gf_diag = dec.reconstruct(1.0/(ww[:, np.newaxis] - dec.eig), kind='diag')
>>> gf_diag = gf_diag.reshape(ww.size, N, N)
```

CHAPTER

THREE

GFTOOL

Collection of commonly used Green's functions and utilities.

Main purpose is to have a tested base.

3.1 Submodules

gftool.basis	Different function bases.
gftool.beb	Blackman, Esterling, and Berk (BEB) approach to off-diagonal disorder.
gftool.cpa	Coherent cluster approximation (CPA) to substitutional disorder.
gftool.fourier	Fourier transformations of Green's functions.
gftool.hermpade	Hermite-Padé approximants from Taylor expansion.
gftool.lattice	Collection of different lattices and their Green's functions.
gftool.linalg	Collection of linear algebra algorithms not contained in `numpy` or `scipy`.
gftool.linearprediction	Linear prediction to extrapolated retarded Green's function.
gftool.matrix	Functions to work with Green's functions in matrix from.
gftool.pade	Padé analytic continuation for Green's functions and self-energies.
gftool.polepade	Padé based on robust pole finding.
gftool.siam	Basic functions for the (non-interacting) single impurity Anderson model (SIAM).

3.1.1 gftool.basis

Different function bases.

The basis classes are based on `NamedTuple`, hence they have hardly any overhead. On the other hand, no additional checks are performed in the constructor.

Submodules

pole	Representation using poles and the corresponding residues.

gftool.basis.pole

Representation using poles and the corresponding residues.

Assuming we have only simple poles Green's functions, we can represent Green's functions using these poles and their corresponding residues:

$$g(z) = \sum_j r_j/(z - {}_j)$$

where ${}_j$ are the poles and r_j the corresponding residues. Self-energies can also be represented by the poles after subtracting the static part.

The pole representation is closely related to the Padé approximation, as rational polynomials with numerator degree *N* bigger then dominator degree *M*, can also be represented using *M* poles.

API

Functions

gf_d1_z(z, poles, weights)	First derivative of Green's function given by a finite number of *poles*.
gf_from_moments(moments[, width])	Find pole Green's function matching given *moments*.
gf_from_tau(gf_tau, n_pole, beta[, moments, ...])	Find pole Green's function fitting *gf_tau*.
gf_from_z(z, gf_z, n_pole[, moments, width, ...])	Find pole causal Green's function fitting *gf_z*.
gf_ret_t(tt, poles, weights)	Retarded time Green's function given by a finite number of *poles*.
gf_tau(tau, poles, weights, beta)	Imaginary time Green's function given by a finite number of *poles*.
gf_z(z, poles, weights)	Green's function given by a finite number of *poles*.
moments(poles, weights, order)	High-frequency moments of the pole Green's function.

gftool.basis.pole.gf_d1_z

gftool.basis.pole.**gf_d1_z**(*z*, *poles*, *weights*)

First derivative of Green's function given by a finite number of *poles*.

To be a Green's function, *np.sum(weights)* has to be 1 for the 1/z tail.

Parameters

z
[(…) complex array_like] Green's function is evaluated at complex frequency z.

poles, weights
[(…, N) float array_like or float] The position and weight of the poles.

Returns

(…) complex np.ndarray
Derivative of the Green's function.

See also:

gf_z

gftool.basis.pole.gf_from_moments

gftool.basis.pole.**gf_from_moments**(*moments*, *width=1.0*) → *PoleFct*

Find pole Green's function matching given `moments`.

Finds poles and weights for a pole Green's function matching the given high frequency `moments` for large *z*: *g(z) = np.sum(weights / (z - poles)) = moments / z**np.arange(N)*

Note that for an odd number of moments, the central pole is at *z = 0*, so *g(0)* diverges.

Parameters

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *G(z) = moments / z**np.arange(1, N+1)* for large *z*.

width
[float or (…) float array_like or None, optional] Spread of the poles; they are in the interval [-width, width]. *width=1* are the normal Chebyshev nodes in the interval [-1, 1]. If *width=None* and the second moment *moments[…, 1]* is given, the largest pole will match the second moment, unless it is small (*abs(moments[…, 1]) < 0.1*), then we choose *width=1*.

Returns

gf.resids
[(…, N) float np.ndarray] Residues (or weight) of the poles.

gf.poles
[(N) or (…, N) float np.ndarray] Position of the poles, these are the Chebyshev nodes for degree *N*.

Notes

We employ the similarity of the relation betweens the `moments` and the poles and residues with polynomials and the Vandermond matrix. The poles are chooses as Chebyshev nodes, the residues are calculated accordingly.

gftool.basis.pole.gf_from_tau

gftool.basis.pole.**gf_from_tau**(*gf_tau*, *n_pole*, *beta*, *moments=()*, *occ=False*, *width=1.0*, *weight=None*) → *PoleGf*

Find pole Green's function fitting `gf_tau`.

Finds poles and weights for a pole Green's function matching the given Green's function `gf_tau`.

Note that for an odd number of moments, the central pole is at *z = 0*, so the causal Green's function *g(0)* diverges.

Parameters

gf_tau
[(…, N_tau) float np.ndarray] Imaginary times Green's function which is fitted.

n_pole
[int] Number of poles to fit.

beta
[float] The inverse temperature $beta = 1/k_B T$.

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *G(z) = moments / z**np.arange(N)* for large z.

occ
[float, optional] If given, fix occupation of pole Green's function to *occ* (default: False).

width
[float, optional] Distance of the largest pole to the origin (default: 1.).

weight
[(…, N_tau) float np.ndarray, optional] Weight the values of `gf_tau`, can be provided to include uncertainty.

Returns

gf.resids
[(…, N) float np.ndarray] Residues (or weight) of the poles.

gf.poles
[(N) float np.ndarray] Position of the poles, these are the Chebyshev nodes for degree *N*.

Raises

ValueError
If more moments are given than poles are fitted (*len(moments) > n_pole*).

Notes

We employ the similarity of the relation betweens the `moments` and the poles and residues with polynomials and the Vandermond matrix. The poles are chooses as Chebyshev nodes, the residues are calculated accordingly.

gftool.basis.pole.gf_from_z

gftool.basis.pole.**gf_from_z**(*z*, *gf_z*, *n_pole*, *moments=()*, *width=1.0*, *weight=None*) → *PoleFct*

Find pole causal Green's function fitting `gf_z`.

This function is only meaningful away from the real axis. Finds poles and weights for a pole Green's function matching the given Green's function `gf_z`.

Note that for an odd number of moments, the central pole is at *z = 0*, so the causal Green's function *g(0)* diverges.

Parameters

z
[(…, N_z) complex np.ndarray] Frequencies at which `gf_z` is given. Mind that the fit is only meaningful away from the real axis.

gf_z
[(…, N_z) complex np.ndarray] Causal Green's function which is fitted.

n_pole
[int] Number of poles to fit.

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *G(z) = moments / z**np.arange(N)* for large *z*.

width
[float or (…) float array_like or None, optional] Spread of the poles; they are in the interval [-width, width]. (default: 1.) *width=1* are the normal Chebyshev nodes in the interval [-1, 1]. If *width=None* and the second moment *moments[…, 1]* is given, the largest pole will match the second moment, unless it is small (*abs(moments[…, 1]) < 0.1*), then we choose *width=1*.

weight
[(…, N_z) float np.ndarray, optional] Weighting of the fit. If an error σ of the input `gf_z` is known, this should be *weight=1/σ*. If high-frequency moments should be fitted correctly, *weight=abs(z)**(N+1)* is a good fit.

Returns

gf.resids
[(…, N) float np.ndarray] Residues (or weight) of the poles.

gf.poles
[(N) or (…, N) float np.ndarray] Position of the poles, these are the Chebyshev nodes for degree *N*.

Raises

ValueError
If more moments are given than poles are fitted (*len(moments) > n_pole*).

Notes

We employ the similarity of the relation betweens the `moments` and the poles and residues with polynomials and the Vandermond matrix. The poles are chooses as Chebyshev nodes, the residues are calculated accordingly.

gftool.basis.pole.gf_ret_t

gftool.basis.pole.**gf_ret_t**(*tt*, *poles*, *weights*)

Retarded time Green's function given by a finite number of *poles*.

Parameters

tt
[(…) float array_like] Green's function is evaluated at times *tt*, for *tt<0* it is 0.

poles, weights
[(…, N) float array_like or float] Position and weight of the poles.

Returns

(…) float np.ndarray
Retarded time Green's function.

See also:

pole_gf_z
Corresponding commutator Green's function.

gftool.basis.pole.gf_tau

gftool.basis.pole.**gf_tau**(*tau*, *poles*, *weights*, *beta*)

Imaginary time Green's function given by a finite number of *poles*.

Parameters

tau
[(…) float array_like] Green's function is evaluated at imaginary times *tau*. Only implemented for $\in[0,]$.

poles, weights
[(…, N) float array_like or float] Position and weight of the poles.

beta
[float] Inverse temperature.

Returns

(…) float np.ndarray
Imaginary time Green's function.

See also:

pole_gf_z
Corresponding commutator Green's function.

gftool.basis.pole.gf_z

gftool.basis.pole.**gf_z**(*z*, *poles*, *weights*)

Green's function given by a finite number of *poles*.

To be a Green's function, *np.sum(weights)* has to be 1 for the *1/z* tail or respectively the normalization.

Parameters

z
[(…) complex array_like] Green's function is evaluated at complex frequency z.

poles, weights
[(…, N) float array_like or float] The position and weight of the poles.

Returns

(…) complex np.ndarray
Green's function.

See also:

gf_d1_z
First derivative of the Green's function.

gf_tau
Corresponding fermionic imaginary time Green's function.

gt.pole_gf_tau_b
Corresponding bosonic imaginary time Green's function.

gftool.basis.pole.moments

gftool.basis.pole.**moments**(*poles*, *weights*, *order*)

High-frequency moments of the pole Green's function.

Return the moments *mom* of the expansion $g(z) = \sum_m mom_m / z^m$ For the pole Green's function we have the simple relation $1/(z -) = \sum_{m=1}{}^{m-1}/z^m$.

Parameters

poles, weights
[(…, N) float np.ndarray] Position and weight of the poles.

order
[(…, M) int array_like] Order (degree) of the moments. *order* needs to be a positive integer.

Returns

(…, M) float np.ndarray
High-frequency moments.

Classes

PoleFct(poles, residues)	Function given by finite number of simple *poles* and *residues*.
PoleGf(poles, residues)	Fermionic Green's function given by finite number of *poles* and *residues*.

gftool.basis.pole.PoleFct

class gftool.basis.pole.**PoleFct**(*poles: ndarray*, *residues: ndarray*)

Function given by finite number of simple *poles* and *residues*.

Parameters

poles, residues
[(…, N) complex np.ndarray] Poles and residues of the function.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)

count(value, /)	Return number of occurrences of value.
eval_z(z)	Evaluate the function at *z*.
from_moments(moments[, width])	Generate instance matching high-frequency *moments*.
from_z(z, gf_z, n_pole[, moments, width, weight])	Generate instance fitting *gf_z*.
index(value[, start, stop])	Return first index of value.
moments(order)	Calculate high-frequency moments of *order*.

gftool.basis.pole.PoleFct.__init__

PoleFct.**__init__**(**args*, ***kwargs*)

gftool.basis.pole.PoleFct.count

PoleFct.**count**(*value*, */*)

Return number of occurrences of value.

gftool.basis.pole.PoleFct.eval_z

PoleFct.**eval_z**(*z*)

Evaluate the function at *z*.

gftool.basis.pole.PoleFct.from_moments

classmethod PoleFct.**from_moments**(*moments*, *width=1.0*)

Generate instance matching high-frequency *moments*.

Parameters

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *g(z) = moments / z**np.arange(1, N+1)* for large *z*.

width
[float or (…) float array_like or None, optional] Spread of the poles; they are in the interval [-width, width]. *width=1* are the normal Chebyshev nodes in the interval [-1, 1]. If *width=None* and the second moment *moments[…, 1]* is given, the largest pole will match the second moment, unless it is small (*abs(moments[…, 1]) < 0.1*), then we choose *width=1*.

Returns

PoleFct
Pole function with high-frequency *moments*.

See also:

gf_from_moments
Contains the details how *PoleFct* is constructed.

gftool.basis.pole.PoleFct.from_z

classmethod PoleFct.**from_z**(*z*, *gf_z*, *n_pole*, *moments=()*, *width=1.0*, *weight=None*)

Generate instance fitting *gf_z*.

This function is only meaningful away from the real axis. Finds poles and weights for a pole Green's function matching the given Green's function *gf_z*.

Note that for an odd number of moments, the central pole is at *z = 0*, so the causal Green's function *g(0)* diverges.

Parameters

z
[(…, N_z) complex np.ndarray] Frequencies at which `gf_z` is given. Mind that the fit is only meaningful away from the real axis.

gf_z
[(…, N_z) complex np.ndarray] Causal Green's function which is fitted.

n_pole
[int] Number of poles to fit.

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *G(z) = moments /z**np.arange(N)* for large *z*.

width
[float, optional] Distance of the largest pole to the origin (default: 1.0).

weight
[(…, N_z) float np.ndarray, optional] Weighting of the fit. If an error σ of the input `gf_z` is known, this should be *weight=1/σ*. If high-frequency moments should be fitted correctly, *width=abs(z)**(N+1)* is a good fit.

Returns

PoleFct
Instance with (N) poles at the Chebyshev nodes for degree *N* and (…, N) residues such that the pole function fits `gf_z`.

Raises

ValueError
If more moments are given than poles are fitted (*len(moments) > n_pole*).

See also:

gf_from_z

Notes

We employ the similarity of the relation betweens the `moments` and the poles and residues with polynomials and the Vandermond matrix. The poles are chooses as Chebyshev nodes, the residues are calculated accordingly.

gftool.basis.pole.PoleFct.index

`PoleFct.`**`index`**(*value*, *start=0*, *stop=sys.maxsize*, */*)

Return first index of value.

Raises ValueError if the value is not present.

gftool.basis.pole.PoleFct.moments

PoleFct.**moments**(*order*)

Calculate high-frequency moments of *order*.

Parameters

order
[(…, M) int array_like] Order (degree) of the moments. *order* needs to be a positive integer. Leading all but the last dimension must be broadcastable with *self.poles* and *self.residues*.

Returns

(…, M) float np.ndarray
High-frequency moments.

See also:

moments

Attributes

poles	Poles of the function.
residues	Residues corresponding to the poles.

gftool.basis.pole.PoleFct.poles

property PoleFct.**poles**

Poles of the function.

gftool.basis.pole.PoleFct.residues

property PoleFct.**residues**

Residues corresponding to the poles.

gftool.basis.pole.PoleGf

class gftool.basis.pole.**PoleGf**(*poles: ndarray*, *residues: ndarray*)

Fermionic Green's function given by finite number of *poles* and *residues*.

Parameters

poles, residues
[(…, N) complex np.ndarray] Poles and residues of the function.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	

count(value, /)	Return number of occurrences of value.
eval_ret_t(tt)	Evaluate the retarded time Green's function.
eval_tau(tau, beta)	Evaluate the imaginary time Green's function.
eval_z(z)	Evaluate the function at z.
from_moments(moments[, width])	Generate instance matching high-frequency *moments*.
from_tau(gf_tau, n_pole, beta[, moments, ...])	Generate instance fitting *gf_tau*.
from_z(z, gf_z, n_pole[, moments, width, weight])	Generate instance fitting *gf_z*.
index(value[, start, stop])	Return first index of value.
moments(order)	Calculate high-frequency moments of *order*.
occ(beta)	Calculate the occupation number.

gftool.basis.pole.PoleGf.__init__

PoleGf.**__init__**(**args*, ***kwargs*)

gftool.basis.pole.PoleGf.count

PoleGf.**count**(*value*, */*)

Return number of occurrences of value.

gftool.basis.pole.PoleGf.eval_ret_t

PoleGf.**eval_ret_t**(*tt*)

Evaluate the retarded time Green's function.

Parameters

tt
[(…) float array_like] Green's function is evaluated at times *tt*, for *tt<0* it is 0.

Returns

(…) float np.ndarray
Retarded time Green's function.

See also:

gf_ret_t

gftool.basis.pole.PoleGf.eval_tau

PoleGf.**eval_tau**(*tau*, *beta*)

Evaluate the imaginary time Green's function.

Parameters

tau
[(…) float array_like] Green's function is evaluated at imaginary times *tau*. Only implemented for $\in[0,]$.

beta
[float] Inverse temperature.

Returns

(…) float np.ndarray
Imaginary time Green's function.

See also:

gf_tau

gftool.basis.pole.PoleGf.eval_z

PoleGf.**eval_z**(*z*)

Evaluate the function at *z*.

gftool.basis.pole.PoleGf.from_moments

classmethod PoleGf.**from_moments**(*moments*, *width=1.0*)

Generate instance matching high-frequency *moments*.

Parameters

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *g(z) = moments / z**np.arange(1, N+1)* for large *z*.

width
[float or (…) float array_like or None, optional] Spread of the poles; they are in the interval [-width, width]. *width=1* are the normal Chebyshev nodes in the interval [-1, 1]. If *width=None* and the second moment *moments[…, 1]* is given, the largest pole will match the second moment, unless it is small (*abs(moments[…, 1]) < 0.1*), then we choose *width=1*.

Returns

PoleFct
Pole function with high-frequency *moments*.

See also:

gf_from_moments
Contains the details how *PoleFct* is constructed.

gftool.basis.pole.PoleGf.from_tau

classmethod PoleGf.**from_tau**(*gf_tau*, *n_pole*, *beta*, *moments=()*, *occ=False*, *width=1.0*, *weight=None*)

Generate instance fitting `gf_tau`.

Finds poles and weights for a pole Green's function matching the given Green's function `gf_tau`.

Note that for an odd number of moments, the central pole is at $z = 0$, so the causal Green's function *g(0)* diverges.

Parameters

gf_tau
[(…, N_tau) float np.ndarray] Imaginary times Green's function which is fitted.

n_pole
[int] Number of poles to fit.

beta
[float] The inverse temperature $beta = 1/k_B T$.

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *G(z) = moments / z**np.arange(N)* for large z.

occ
[float, optional] If given, fix occupation of pole Green's function to `occ` (default: False).

width
[float, optional] Distance of the largest pole to the origin (default: 1.0).

weight
[(…, N_tau) float np.ndarray, optional] Weight the values of `gf_tau`, can be provided to include uncertainty.

Returns

PoleFct
Instance with (N) poles at the Chebyshev nodes for degree *N* and (…, N) residues such that the pole function fits `gf_z`.

Raises

ValueError
If more moments are given than poles are fitted (*len(moments) > n_pole*).

See also:

gf_from_tau

Notes

We employ the similarity of the relation betweens the *moments* and the poles and residues with polynomials and the Vandermond matrix. The poles are chooses as Chebyshev nodes, the residues are calculated accordingly.

gftool.basis.pole.PoleGf.from_z

classmethod PoleGf.**from_z**(*z*, *gf_z*, *n_pole*, *moments=()*, *width=1.0*, *weight=None*)

Generate instance fitting *gf_z*.

This function is only meaningful away from the real axis. Finds poles and weights for a pole Green's function matching the given Green's function *gf_z*.

Note that for an odd number of moments, the central pole is at $z = 0$, so the causal Green's function *g(0)* diverges.

Parameters

z
[(…, N_z) complex np.ndarray] Frequencies at which *gf_z* is given. Mind that the fit is only meaningful away from the real axis.

gf_z
[(…, N_z) complex np.ndarray] Causal Green's function which is fitted.

n_pole
[int] Number of poles to fit.

moments
[(…, N) float array_like] Moments of the high-frequency expansion, where *G(z) = moments /z**np.arange(N)* for large *z*.

width
[float, optional] Distance of the largest pole to the origin (default: 1.0).

weight
[(…, N_z) float np.ndarray, optional] Weighting of the fit. If an error σ of the input *gf_z* is known, this should be *weight=1/σ*. If high-frequency moments should be fitted correctly, *width=abs(z)**(N+1)* is a good fit.

Returns

PoleFct
Instance with (N) poles at the Chebyshev nodes for degree *N* and (…, N) residues such that the pole function fits *gf_z*.

Raises

ValueError
If more moments are given than poles are fitted (*len(moments) > n_pole*).

See also:

gf_from_z

Notes

We employ the similarity of the relation betweens the *moments* and the poles and residues with polynomials and the Vandermond matrix. The poles are chooses as Chebyshev nodes, the residues are calculated accordingly.

gftool.basis.pole.PoleGf.index

PoleGf.**index**(*value*, *start=0*, *stop=sys.maxsize*, /)

Return first index of value.

Raises ValueError if the value is not present.

gftool.basis.pole.PoleGf.moments

PoleGf.**moments**(*order*)

Calculate high-frequency moments of *order*.

Parameters

order
[(…, M) int array_like] Order (degree) of the moments. *order* needs to be a positive integer. Leading all but the last dimension must be broadcastable with *self.poles* and *self.residues*.

Returns

(…, M) float np.ndarray
High-frequency moments.

See also:

moments

gftool.basis.pole.PoleGf.occ

PoleGf.**occ**(*beta*)

Calculate the occupation number.

Parameters

beta
[float or (…, 1) float array_like] The inverse temperature $beta = 1/k_B T$.

Returns

(…) float np.ndarray
Occupation number.

Attributes

poles	Alias for field number 0
residues	Alias for field number 1

gftool.basis.pole.PoleGf.poles

property PoleGf.**poles**

Alias for field number 0

gftool.basis.pole.PoleGf.residues

property PoleGf.**residues**

Alias for field number 1

API

Classes

RatPol(numer, denom)	Rational polynomial given as numerator and denominator.
ZeroPole(zeros, poles[, amplitude])	Rational polynomial characterized by zeros and poles.

gftool.basis.RatPol

class gftool.basis.**RatPol**(*numer: Polynomial*, *denom: Polynomial*)

Rational polynomial given as numerator and denominator.

Parameters

numer, denom
[np.polynomial.Polynomial] Numerator and denominator, given as numpy polynomials.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)

count(value, /)	Return number of occurrences of value.
eval(z)	Evaluate the rational polynomial at *z*.
index(value[, start, stop])	Return first index of value.

gftool.basis.RatPol.__init__

RatPol.**__init__**(**args*, ***kwargs*)

gftool.basis.RatPol.count

RatPol.**count**(*value*, /)

Return number of occurrences of value.

gftool.basis.RatPol.eval

RatPol.**eval**(*z*)

Evaluate the rational polynomial at *z*.

gftool.basis.RatPol.index

RatPol.**index**(*value*, *start=0*, *stop=sys.maxsize*, /)

Return first index of value.

Raises ValueError if the value is not present.

Attributes

denom	Denominator of the rational polynomial.
numer	Numerator of the rational polynomial.

gftool.basis.RatPol.denom

property RatPol.**denom**

Denominator of the rational polynomial.

gftool.basis.RatPol.numer

property RatPol.**numer**

Numerator of the rational polynomial.

gftool.basis.ZeroPole

class gftool.basis.**ZeroPole**(*zeros: ndarray*, *poles: ndarray*, *amplitude: complex = 1.0*)

Rational polynomial characterized by zeros and poles.

The function is given by `ZeroPole.eval(z) = amplitude * np.prod(z - zeros) / np.prod(z - poles)`

Parameters

zeros, poles
[(…, Nz), (…, Np) complex np.ndarray] Zeros and poles of the represented function.

amplitude
[(…) complex np.ndarray or complex] The amplitude of the function. This is also the large *abs(z)* limit of the function *ZeroPole.eval(z) = amplitude * z**(Nz-Np)*.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	

count(value, /)	Return number of occurrences of value.
eval(z)	Evaluate the function at *z*.
index(value[, start, stop])	Return first index of value.
reciprocal(z)	Evaluate the reciprocal *1./ZeroPole.eval(z)* at *z*.
to_ratpol()	Represent the rational polynomial as fraction of two polynomial.

gftool.basis.ZeroPole.__init__

ZeroPole.**__init__**(**args*, ***kwargs*)

gftool.basis.ZeroPole.count

ZeroPole.**count**(*value*, */*)

Return number of occurrences of value.

gftool.basis.ZeroPole.eval

ZeroPole.**eval**(*z*)

Evaluate the function at *z*.

Parameters

z
[(…) complex np.ndarray] Point at which the function is evaluated.

Returns

(…) complex np.ndarray
The function evaluated at z.

gftool.basis.ZeroPole.index

`ZeroPole.`**`index`**(*value*, *start=0*, *stop=sys.maxsize*, */*)

Return first index of value.

Raises ValueError if the value is not present.

gftool.basis.ZeroPole.reciprocal

`ZeroPole.`**`reciprocal`**(*z*)

Evaluate the reciprocal *1./ZeroPole.eval(z)* at z.

Parameters

z
[(…) complex np.ndarray] Point at which the reciprocal of the function is evaluated.

Returns

(…) complex np.ndarray
The reciprocal of the function evaluated at z.

gftool.basis.ZeroPole.to_ratpol

`ZeroPole.`**`to_ratpol`**() → *RatPol*

Represent the rational polynomial as fraction of two polynomial.

Attributes

amplitude	Amplitude of the function, i.e., the prefactor.
poles	Poles of the rational polynomial.
zeros	Zeros of the rational polynomial.

gftool.basis.ZeroPole.amplitude

property `ZeroPole.`**`amplitude`**

Amplitude of the function, i.e., the prefactor.

gftool.basis.ZeroPole.poles

property ZeroPole.**poles**
Poles of the rational polynomial.

gftool.basis.ZeroPole.zeros

property ZeroPole.**zeros**
Zeros of the rational polynomial.

3.1.2 gftool.beb

Blackman, Esterling, and Berk (BEB) approach to off-diagonal disorder.

It extends CPA allowing for random hopping amplitudes. [blackman1971]

The implementation is based on a SVD of the *hopping* matrix, which is the dimensionless scaling of the hopping of the components. [weh2021] However, we use the unitary eigendecomposition instead of the SVD.

Physical quantities

The main quantity of interest is the average local Green's function *gf*.

First the effective medium *self_beb_z* has to be calculated using `solve_root`. With this result the Green's function can be calculated by the function `gf_loc_z`.

In the BEB formalism, the local Green's function *gf* is a matrix in the components. The self-consistent Green's function *gf* is diagonal, its trace is the average physical Green's function. If only the non-vanishing diagonal elements have been calculated *gf=gf_loc_z(..., diag=True)*, the average Green's function is *np.sum(gf, axis=-1)*. The diagonal elements of *gf* are the average for a specific component (conditional average) multiplied by the concentration of that component.

References

Examples

We consider a Bethe lattice with two components 'A' and 'B'. The have the on-site energies *-0.5* and *0.5* respectively, the concentrations *0.3* and *0.7*. Furthermore, we assume that the hopping amplitude between 'A' and 'B' is only *0.3* times the hopping between two 'A' sites, while the hopping between two 'B' sites is *1.2* times the hopping between two 'A' sites.

Then the following code calculates the local Green's function for component 'A' and 'B' (conditionally averaged) as well as the average Green's function of the system.

```
from functools import partial

import gftool as gt
import numpy as np
import matplotlib.pyplot as plt

eps = np.array([-0.5, 0.5])
c = np.array([0.3, 0.7])
t = np.array([[1.0, 0.3],
              [0.3, 1.2]])
```

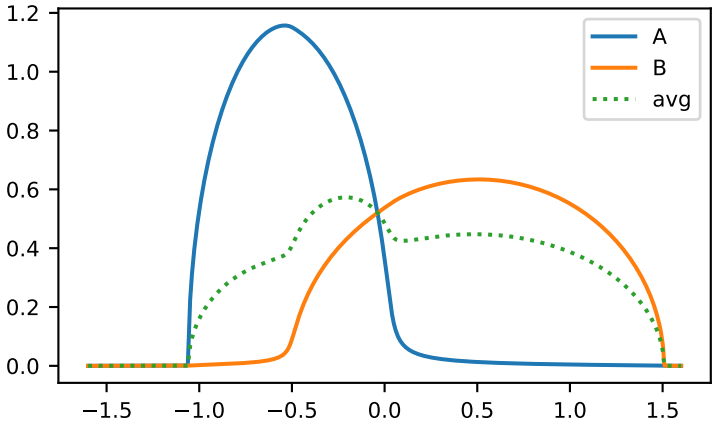
(continues on next page)

(continued from previous page)

```
hilbert = partial(gt.bethe_hilbert_transform, half_bandwidth=1)

ww = np.linspace(-1.6, 1.6, num=1000) + 1e-4j
self_beb_ww = gt.beb.solve_root(ww, e_onsite=eps, concentration=c, hopping=t,
                                hilbert_trafo=hilbert)
gf_loc_ww = gt.beb.gf_loc_z(ww, self_beb_ww, hopping=t, hilbert_trafo=hilbert)

__ = plt.plot(ww.real, -1./np.pi/c[0]*gf_loc_ww[:, 0].imag, label='A')
__ = plt.plot(ww.real, -1./np.pi/c[1]*gf_loc_ww[:, 1].imag, label='B')
__ = plt.plot(ww.real, -1./np.pi*np.sum(gf_loc_ww.imag, axis=-1), ':', label='avg')
__ = plt.legend()
plt.show()
```



API

Functions

gf_loc_z(z, self_beb_z, hopping, hilbert_trafo)	Calculate average local Green's function matrix in components.
restrict_self_root_eq(self_beb_z, *args, **kwds)	Wrap *self_root_eq* to restrict the solutions to *diagonal(self_beb_z).imag > 0*.
self_root_eq(self_beb_z, z, e_onsite, ...)	Root equation r(Σ)=0 for BEB.
solve_root(z, e_onsite, concentration, ...)	Determine the BEB self-energy by solving the root problem.

gftool.beb.gf_loc_z

gftool.beb.**gf_loc_z**(*z*, *self_beb_z*, *hopping*, *hilbert_trafo: Callable[[complex], complex]*, *diag=True*, *rcond=None*)

Calculate average local Green's function matrix in components.

For the self-consistent self-energy *self_beb_z* it is diagonal in the components. Note, that *gf_loc_z* contain the *concentration*.

Parameters

z
[(…) complex np.ndarray] Frequency points.

self_beb_z
[(…, N_cmpt, N_cmpt) complex np.ndarray] BEB self-energy.

hopping
[(N_cmpt, N_cmpt) float array_like] Hopping matrix in the components.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the local Green's function.

diag
[bool, optional] If *diag*, only the diagonal elements are calculated, else the full matrix (default: True).

rcond
[float, optional] Cut-off ratio for small singular values of *hopping*. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of *hopping*.

Returns

(…, N_cmpt) or (…, N_cmpt, N_cmpt) complex np.ndarray
The average local Green's function matrix.

See also:

solve_root

gftool.beb.restrict_self_root_eq

gftool.beb.**restrict_self_root_eq**(*self_beb_z*, **args*, ***kwds*)

Wrap *self_root_eq* to restrict the solutions to *diagonal(self_beb_z).imag > 0*.

gftool.beb.self_root_eq

gftool.beb.**self_root_eq**(*self_beb_z*, *z*, *e_onsite*, *concentration*, *hopping_dec: SpecDec*, *hilbert_trafo: Callable[[complex], complex]*)

Root equation r(Σ)=0 for BEB.

Parameters

self_beb_z
[(…, N_cmpt, N_cmpt) complex np.ndarray] BEB self-energy.

z
[(…) complex np.ndarray] Frequency points.

e_onsite
[(…, N_cmpt) float or complex array_like] On-site energy of the components.

concentration
[(…, N_cmpt) float array_like] Concentration of the different components.

hopping_dec
[SVD] Compact SVD decomposition of the (N_cmpt, N_cmpt) hopping matrix in the components.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the local Green's function.

Returns

(…, N_cmpt, N_cmpt)
Difference of the inverses of the local and the average Green's function. If *diff = 0*, *self_beb_z* is the correct self-energy.

See also:

solve_root

gftool.beb.solve_root

gftool.beb.**solve_root**(*z*, *e_onsite*, *concentration*, *hopping*, *hilbert_trafo: Callable[[complex], complex]*, *self_beb_z0=None*, *restricted=True*, *rcond=None*, ***root_kwds*)

Determine the BEB self-energy by solving the root problem.

Note, that the result should be checked, whether the obtained solution is physical.

Parameters

z
[(…) complex np.ndarray] Frequency points.

e_onsite
[(…, N_cmpt) float or complex np.ndarray] On-site energy of the components.

concentration
[(…, N_cmpt) float np.ndarray] Concentration of the different components.

hopping
[(N_cmpt, N_cmpt) float array_like] Hopping matrix in the components.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the local Green's function.

self_beb_z0
[(…, N_cmpt, N_cmpt) complex np.ndarray, optional] Starting guess for the BEB self-energy.

restricted
[bool, optional] Whether the diagonal of *self_beb_z* is restricted to *self_beb_z.imag <= 0* (default: True). Note, that even if *restricted=True*, the imaginary part can get negative within tolerance. This should be removed by hand if necessary.

rcond
[float, optional] Cut-off ratio for small singular values of *hopping*. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of *hopping*.

****root_kwds**
Additional arguments passed to `scipy.optimize.root`. *method* can be used to choose a solver. *options=dict(fatol=tol)* can be specified to set the desired tolerance *tol*.

Returns

(…, N_cmpt, N_cmpt) complex np.ndarray
The BEB self-energy as the root of *self_root_eq*.

Raises

RuntimeError
If the root problem cannot be solved.

See also:

gf_loc_z

Notes

The root problem is solved for the complete input simultaneously. This provides a speed up as the code is vectorized, however, it comes with the trade-off of complicating the root search. So in some cases, it makes sense to split the input arrays, and calculate the root separately.

The default method is 'krylov', which typically does a good job. In some cases 'excitingmixing' was found to do a better job, especially close to the CPA limit, where some singular values become small.

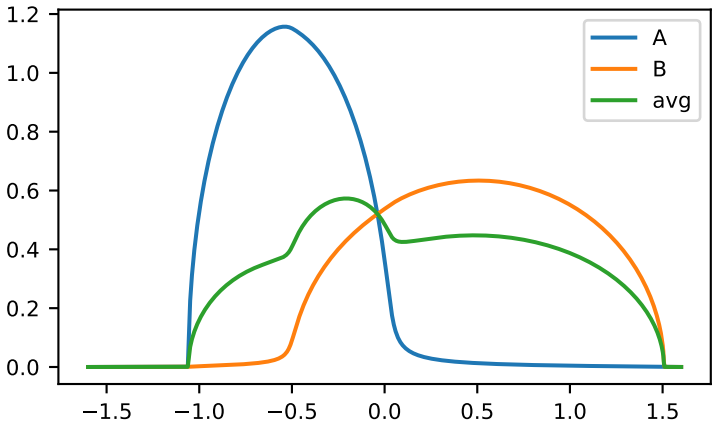
The progress of the root search is logged for the `logging.DEBUG` level.

Examples

```
>>> from functools import partial
>>> eps = np.array([-0.5, 0.5])
>>> c = np.array([0.3, 0.7])
>>> t = np.array([[1.0, 0.3],
...               [0.3, 1.2]])
>>> hilbert = partial(gt.bethe_hilbert_transform, half_bandwidth=1)
```

```
>>> ww = np.linspace(-1.6, 1.6, num=1000) + 1e-4j
>>> self_beb_ww = gt.beb.solve_root(ww, e_onsite=eps, concentration=c, hopping=t,
...                                 hilbert_trafo=hilbert)
>>> gf_loc_ww = gt.beb.gf_loc_z(ww, self_beb_ww, hopping=t, hilbert_trafo=hilbert)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(ww.real, -1./np.pi/c[0]*gf_loc_ww[:, 0].imag, label='A')
>>> __ = plt.plot(ww.real, -1./np.pi/c[1]*gf_loc_ww[:, 1].imag, label='B')
>>> __ = plt.plot(ww.real, -1./np.pi*np.sum(gf_loc_ww.imag, axis=-1), label='avg')
>>> __ = plt.legend()
>>> plt.show()
```



Classes

SpecDec(rv, eig, rv_inv)	SVD like spectral decomposition.

gftool.beb.SpecDec

class gftool.beb.**SpecDec**(*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*)

SVD like spectral decomposition.

Works only for N×N matrices unlike the `UDecomposition` base class.

Parameters

rv
[(…, N, N) complex np.ndarray] The matrix of right eigenvectors.

eig
[(…, N) float np.ndarray] The vector of real eigenvalues.

rv_inv
[(…, N, N) complex np.ndarray] The inverse of *rv*.

__init__ (*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*) → None

Methods

__init__(rv, eig, rv_inv)	
count(value)	
from_gf(gf)	Decompose the inverse Green's function matrix.
from_hamiltonian(hamilton)	Decompose the Hamiltonian matrix.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
partition([return_sqrts])	Symmetrically partition the spectral decomposition as *u * eig**0.5, eig**0.5 * uh*.
reconstruct([eig, kind])	Get matrix back from *Decomposition*.
truncate([rcond])	Return the truncated spectral decomposition.

gftool.beb.SpecDec.__init__

SpecDec.**__init__** (*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*) → None

gftool.beb.SpecDec.count

SpecDec.**count** (*value*) → integer -- return number of occurrences of value

gftool.beb.SpecDec.from_gf

classmethod SpecDec.**from_gf** (*gf*) → *Decomposition*

Decompose the inverse Green's function matrix.

The similarity transformation:

$$G^{-1} = PgP^{-1}, \quad g = diag(\iota)$$

Parameters

gf
[(…, N, N) complex np.ndarray] Matrix to be decomposed.

Returns

Decomposition

gftool.beb.SpecDec.from_hamiltonian

classmethod SpecDec.**from_hamiltonian**(*hamilton*)

Decompose the Hamiltonian matrix.

The similarity transformation:

$$H = UhU^\dagger, \quad h = diag(_l)$$

Parameters

hamilton
[(…, N, N) complex np.ndarray] Hermitian matrix to be decomposed.

Returns

Decomposition

gftool.beb.SpecDec.index

SpecDec.**index**(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

gftool.beb.SpecDec.partition

SpecDec.**partition**(*return_sqrts=False*)

Symmetrically partition the spectral decomposition as *u * eig**0.5, eig**0.5 * uh*.

If *return_sqrts* then *us, np.sqrt(s), suh* is returned, else only *us, suh* is returned (default: False).

gftool.beb.SpecDec.reconstruct

SpecDec.**reconstruct**(*eig=None*, *kind='full'*)

Get matrix back from *Decomposition*.

If the reciprocal of *self.eig* was taken, this corresponds to the inverse of the original matrix.

Parameters

eig
[(…, N) np.ndarray, optional] Alternative value used for *self.eig*. This argument can be used instead of modifying *self.eig*.

kind
[{'diag', 'full'} or str] Defines how to reconstruct the matrix. If *kind* is 'diag', only the diagonal elements are computed, if it is 'full' the complete matrix is returned. Alternatively a *str* used for subscript of `numpy.einsum` can be given.

Returns

(…, N, N) or (…, N) np.ndarray
The reconstructed matrix. If a subscript string is given as *kind*, the shape of the output might differ.

gftool.beb.SpecDec.truncate

SpecDec.**truncate**(*rcond=None*) → *SpecDec*

Return the truncated spectral decomposition.

Singular values smaller than *rcond* times the largest singular values are discarded.

Parameters

rcond
[float, rcond] Cut-off ratio for small singular values.

Returns

SpecDec
The truncates the spectral decomposition discarding small singular values.

Attributes

eig	The vector of eigenvalues.
is_trunacted	Check if SVD of square matrix is truncated/compact or full.
rv	The matrix of right eigenvectors.
rv_inv	The inverse of *rv*.
s	Singular values in descending order, different from order of *eig*.
u	Unitary matrix of right eigenvectors, same as *rv*.
uh	Hermitian conjugate of unitary matrix *rv*, same as *rv_inv*.

gftool.beb.SpecDec.eig

SpecDec.**eig**: ndarray

The vector of eigenvalues.

gftool.beb.SpecDec.is_trunacted

property SpecDec.**is_trunacted**: bool

Check if SVD of square matrix is truncated/compact or full.

gftool.beb.SpecDec.rv

SpecDec.**rv**: ndarray

The matrix of right eigenvectors.

gftool.beb.SpecDec.rv_inv

SpecDec.**rv_inv**: **ndarray**
The inverse of *rv*.

gftool.beb.SpecDec.s

property SpecDec.**s**
Singular values in descending order, different from order of *eig*.

gftool.beb.SpecDec.u

property SpecDec.**u**
Unitary matrix of right eigenvectors, same as *rv*.

gftool.beb.SpecDec.uh

property SpecDec.**uh**
Hermitian conjugate of unitary matrix *rv*, same as *rv_inv*.

3.1.3 gftool.cpa

Coherent cluster approximation (CPA) to substitutional disorder.

For a high-level interface use *solve_root* to solve the CPA problem for arbitrary frequencies *z*. Solutions for fixed occupation *occ* can be obtained on the imaginary axis only using *solve_fxdocc_root*.

Fixed occupation on the real axis is currently not support. We recommend obtaining the chemical potential *mu* for the given *occ* using *solve_fxdocc_root* on the imaginary axis, and then run *solve_root* with the given *mu* on the real axis. In fact, we expect this to be more stable than fixing the charge on the real axis directly.

API

Functions

gf_cmpt_z(z, self_cpa_z, e_onsite, hilbert_trafo)	Green's function for the components embedded in *self_cpa_z*.
restrict_self_root_eq(self_cpa_z, *args, **kwds)	Wrap *self_root_eq* to restrict the solutions to *self_cpa_z.imag > 0*.
self_fxdpnt_eq(self_cpa_z, z, e_onsite, ...)	Fixed-point equation f(Σ)=Σ for CPA.
self_root_eq(self_cpa_z, z, e_onsite, ...)	Root equation r(Σ)=0 for CPA.
solve_fxdocc_root(iws, e_onsite, ...[, occ, ...])	Determine the CPA self-energy by solving the root problem for fixed *occ*.
solve_root(z, e_onsite, concentration, ...)	Determine the CPA self-energy by solving the root problem.

gftool.cpa.gf_cmpt_z

gftool.cpa.**gf_cmpt_z**(*z*, *self_cpa_z*, *e_onsite*, *hilbert_trafo: Callable[[complex], complex]*)

Green's function for the components embedded in *self_cpa_z*.

Parameters

z, self_cpa_z
[(…) complex np.ndarray] Frequency points and corresponding CPA self-energy.

e_onsite
[(…, N_cmpt) float of complex np.ndarray] On-site energy of the components. This can also include a local frequency dependent self-energy of the component sites.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the coherent Green's function.

Returns

(…, N_cmpt) complex np.ndarray
The Green's function of the components embedded in *self_cpa_z*.

gftool.cpa.restrict_self_root_eq

gftool.cpa.**restrict_self_root_eq**(*self_cpa_z*, **args*, ***kwds*)

Wrap `self_root_eq` to restrict the solutions to *self_cpa_z.imag > 0*.

gftool.cpa.self_fxdpnt_eq

gftool.cpa.**self_fxdpnt_eq**(*self_cpa_z*, *z*, *e_onsite*, *concentration*, *hilbert_trafo: Callable[[complex], complex]*)

Fixed-point equation f(Σ)=Σ for CPA.

The fixed-point equation writes *f(Σ, z) = Σ + T(z) / (1 + T(z)*hilbert_trafo(z-Σ))*.

Parameters

self_cpa_z
[(…) complex np.ndarray] CPA self-energy.

z
[(…) complex array_like] Frequency points.

e_onsite
[(…, N_cmpt) float complex np.ndarray] On-site energy of the components. This can also include a local frequency dependent self-energy of the component sites.

concentration
[(…, N_cmpt) float array_like] Concentration of the different components used for the average.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the coherent Green's function.

Returns

(…, N_z) complex np.ndarray
The new self-energy f(Σ), if it is Σ again and hence a fixed-point, *self_cpa_z_new* is the correct CPA self-energy.

gftool.cpa.self_root_eq

gftool.cpa.**self_root_eq**(*self_cpa_z*, *z*, *e_onsite*, *concentration*, *hilbert_trafo: Callable[[complex], complex]*)

Root equation r(Σ)=0 for CPA.

The root equation writes *r(Σ, z) = T(z) / (1 + T(z)*hilbert_trafo(z-Σ))*.

Parameters

self_cpa_z
[(…) complex np.ndarray] CPA self-energy.

z
[(…) complex array_like] Frequency points.

e_onsite
[(…, N_cmpt) float or complex np.ndarray] On-site energy of the components. This can also include a local frequency dependent self-energy of the component sites.

concentration
[(…, N_cmpt) float array_like] Concentration of the different components used for the average.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the coherent Green's function.

Returns

(…) complex np.ndarray
The result of r(Σ), if it is *0* and hence a root, *self_cpa_z* is the correct CPA self-energy.

gftool.cpa.solve_fxdocc_root

gftool.cpa.**solve_fxdocc_root**(*iws*, *e_onsite*, *concentration*, *hilbert_trafo: Callable[[complex], complex]*, *beta: float*, *occ: Optional[float] = None*, *self_cpa_iw0=None*, *mu0: float = 0*, *weights=1*, *n_fit=0*, *restricted=True*, ***root_kwds*) → *RootFxdocc*

Determine the CPA self-energy by solving the root problem for fixed *occ*.

Parameters

iws
[(N_iw) complex array_like] Positive fermionic Matsubara frequencies.

e_onsite
[(N_cmpt) float or (…, N_iw, N_cmpt) complex np.ndarray] On-site energy of the components. This can also include a local frequency dependent self-energy of the component sites. If multiple non-frequency dependent on-site energies should be considered simultaneously, pass an on-site energy with *N_z=1*: *e_onsite[…, np.newaxis, :]*.

concentration
[(…, N_cmpt) float array_like] Concentration of the different components used for the average.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the coherent Green's function.

beta
[float] Inverse temperature.

occ
[float] Total occupation.

self_cpa_iw0, mu0
[(…, N_iw) complex np.ndarray and float, optional] Starting guess for CPA self-energy and chemical potential. *self_cpa_iw0* implicitly contains the chemical potential *mu0*, thus they should match.

Returns

root.self_cpa
[(…, N_iw) complex np.ndarray] The CPA self-energy as the root of `self_root_eq`.

root.mu
[float] Chemical potential for the given occupation *occ*.

Other Parameters

weights
[(N_iw) float np.ndarray, optional] Passed to `gftool.density_iw`. Residues of the frequencies with respect to the residues of the Matsubara frequencies *1/beta*. (default: 1.) For Padé frequencies this needs to be provided.

n_fit
[int, optional] Passed to `gftool.density_iw`. Number of additionally fitted moments. If Padé frequencies are used, this is typically not necessary (default: 0).

restricted
[bool, optional] Whether *self_cpa_z* is restricted to *self_cpa_z.imag <= 0*. (default: True) Note, that even if *restricted=True*, the imaginary part can get negative within tolerance. This should be removed by hand if necessary.

****root_kwds**
Additional arguments passed to `scipy.optimize.root`. *method* can be used to choose a solver. *options=dict(fatol=tol)* can be specified to set the desired tolerance *tol*.

Raises

RuntimeError
If unable to find a solution.

See also:

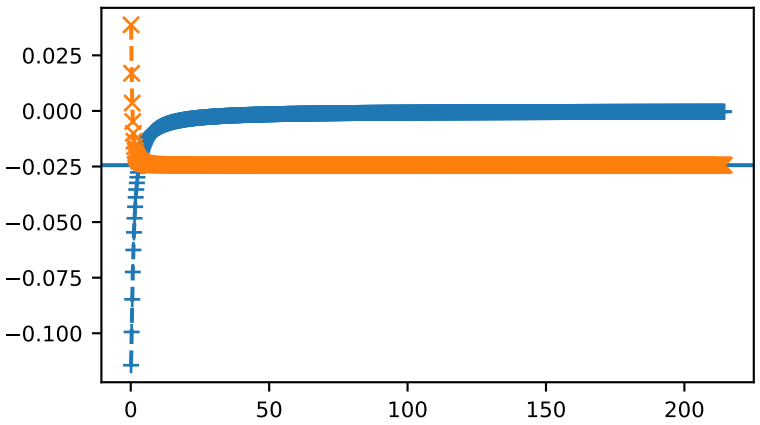
`solve_root`

Examples

```
>>> from functools import partial
>>> beta = 30
>>> e_onsite = [-0.3, 0.3]
>>> conc = [0.3, 0.7]
>>> hilbert = partial(gt.bethe_gf_z, half_bandwidth=1)
>>> occ = 0.5,
```

```
>>> iws = gt.matsubara_frequencies(range(1024), beta=30)
>>> self_cpa_iw, mu = gt.cpa.solve_fxdocc_root(iws, e_onsite, conc,
...                                            hilbert, occ=occ, beta=beta)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(iws.imag, self_cpa_iw.imag, '+--')
>>> __ = plt.axhline(np.average(e_onsite, weights=conc) - mu)
>>> __ = plt.plot(iws.imag, self_cpa_iw.real, 'x--')
>>> plt.show()
```



check occupation

```
>>> gf_coher_iw = hilbert(iws - self_cpa_iw)
>>> gt.density_iw(iws, gf_coher_iw, beta=beta, moments=[1, self_cpa_iw[-1].real])
0.499999...
```

check CPA

```
>>> self_compare = gt.cpa.solve_root(iws, np.array(e_onsite)-mu, conc,
...                                  hilbert_trafo=hilbert)
>>> np.allclose(self_cpa_iw, self_compare, atol=1e-5)
True
```

gftool.cpa.solve_root

gftool.cpa.**solve_root**(*z*, *e_onsite*, *concentration*, *hilbert_trafo: Callable[[complex], complex]*, *self_cpa_z0=None*, *restricted=True*, ***root_kwds*)

Determine the CPA self-energy by solving the root problem.

Note that the result should be checked, whether the obtained solution is physical.

Parameters

z
[(…) complex array_like] Frequency points.

e_onsite
[(…, N_cmpt) float or complex np.ndarray] On-site energy of the components. This can also include a local frequency dependent self-energy of the component sites.

concentration
[(…, N_cmpt) float array_like] Concentration of the different components used for the average.

hilbert_trafo
[Callable[[complex], complex]] Hilbert transformation of the lattice to calculate the coherent Green's function.

self_cpa_z0
[(…) complex np.ndarray, optional] Starting guess for CPA self-energy.

restricted
[bool, optional] Whether *self_cpa_z* is restricted to *self_cpa_z.imag <= 0*. (default: True) Note, that even if *restricted=True*, the imaginary part can get negative within tolerance. This should be removed by hand if necessary.

****root_kwds**
Additional arguments passed to `scipy.optimize.root`. *method* can be used to choose a solver. *options=dict(fatol=tol)* can be specified to set the desired tolerance *tol*.

Returns

(…) complex np.ndarray
The CPA self-energy as the root of `self_root_eq`.

Raises

RuntimeError
If unable to find a solution.

Notes

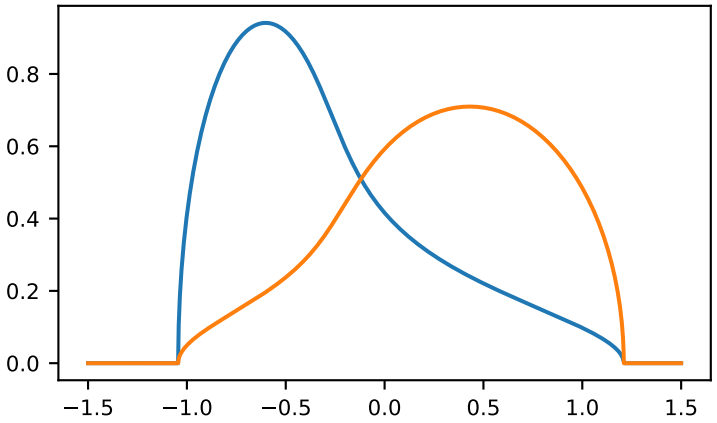
For *restricted=True* root-serach, we made good experince with the methods *'anderson'*, *'krylov'* and *'df-sane'*. For *restricted=False*, we made made good experince with the method *'broyden2'*.

Examples

```
>>> from functools import partial
>>> parameter = dict(
...     e_onsite=[-0.3, 0.3],
...     concentration=[0.3, 0.7],
...     hilbert_trafo=partial(gt.bethe_gf_z, half_bandwidth=1),
... )
```

```
>>> ww = np.linspace(-1.5, 1.5, num=5000) + 1e-10j
>>> self_cpa_ww = gt.cpa.solve_root(ww, **parameter)
>>> del parameter['concentration']
>>> gf_cmpt_ww = gt.cpa.gf_cmpt_z(ww, self_cpa_ww, **parameter)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(ww.real, -1./np.pi*gf_cmpt_ww[..., 0].imag)
>>> __ = plt.plot(ww.real, -1./np.pi*gf_cmpt_ww[..., 1].imag)
>>> plt.show()
```



Classes

RootFxdocc(self_cpa, mu)	CPA solution for the self-energy root-equation for fixed occupation.

gftool.cpa.RootFxdocc

class gftool.cpa.**RootFxdocc**(*self_cpa: ndarray*, *mu: float*)

CPA solution for the self-energy root-equation for fixed occupation.

Parameters

self_cpa
[np.ndarray or complex] The CPA self-energy.

mu
[float] Chemical potential.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

gftool.cpa.RootFxdocc.__init__

RootFxdocc.**__init__**(**args*, ***kwargs*)

gftool.cpa.RootFxdocc.count

RootFxdocc.**count**(*value*, */*)

Return number of occurrences of value.

gftool.cpa.RootFxdocc.index

RootFxdocc.**index**(*value*, *start=0*, *stop=sys.maxsize*, */*)

Return first index of value.

Raises ValueError if the value is not present.

Attributes

mu	Chemical potential.
self_cpa	The CPA self-energy.

gftool.cpa.RootFxdocc.mu

property RootFxdocc.**mu**
Chemical potential.

gftool.cpa.RootFxdocc.self_cpa

property RootFxdocc.**self_cpa**
The CPA self-energy.

3.1.4 gftool.fourier

Fourier transformations of Green's functions.

Fourier transformation between imaginary time and Matsubara frequencies. The function in this module should be used after explicitly treating the high-frequency behavior, as this is not yet implemented. Typically, transformation from τ-space to Matsubara frequency are unproblematic.

The Fourier transforms are defined in the following way:

Definitions

real time → complex frequencies

The Laplace integral for the Green's function is defined as

$$G(z) = \int_{-\infty}^{\infty} dt G(t) \exp(izt)$$

This integral is only well defined

- in the upper complex half-plane *z.imag>=0* for retarded Green's function $\propto(t)$
- in the lower complex half-plane *z.imag<=0* for advanced Green's function $\propto(-t)$

The recommended high-level function to perform this Laplace transform is:

- `tt2z` for both retarded and advanced Green's function

Two different kind of algorithms are available

- `tt2z_trapz` and `tt2z_lin` which approximate the integral,
- `tt2z_pade` and `tt2z_herm2` which are Padé-Fourier type transformations.

Currently, sub-functions can be used equivalently, the abstraction `tt2z` is mostly for consistency with the imaginary time ↔ Matsubara frequencies Fourier transformations.

imaginary time → Matsubara frequencies

The Fourier integral for the Matsubara Green's function is defined as:

$$G(i_n) = 0.5 \int_{-} dG() \exp(i_n)$$

with $iw_n = in/$. For fermionic Green's functions only odd frequencies are non-vanishing, for bosonic Green's functions only even.

The recommended high-level function to perform this Fourier transform is:

- `tau2iw` for *fermionic* Green's functions
- `tau2iv` for *bosonic* Green's functions

Matsubara frequencies → imaginary time

The Fourier sum for the imaginary time Green's function is defined as:

$$G() = 1/ \sum_{n=-\infty}^{\infty} G(i_n) \exp(-i_n).$$

The recommended high-level function to perform this Fourier transform is:

- `iw2tau` for *fermionic* Green's functions

Glossary

dft
: <discrete Foruier transform>

ft
: <Fourier transformation> In contrast to *dft*, this is used for Fourier integration of continous variables without discretization.

Previously defined:

- *iv*
- *iw*
- *tau*

API

Functions

iw2tau(gf_iw, beta[, moments, fourier, n_fit])	Discrete Fourier transform of the Hermitian Green's function *gf_iw*.
iw2tau_dft(gf_iw, beta)	Discrete Fourier transform of the Hermitian Green's function *gf_iw*.
iw2tau_dft_soft(gf_iw, beta)	Discrete Fourier transform of the Hermitian Green's function *gf_iw*.
izp2tau(izp, gf_izp, tau, beta[, moments])	Fourier transform of the Hermitian Green's function *gf_izp* to *tau*.
tau2iv(gf_tau, beta[, fourier])	Fourier transformation of the real Green's function *gf_tau*.
tau2iv_dft(gf_tau, beta)	Discrete Fourier transform of the real Green's function *gf_tau*.
tau2iv_ft_lin(gf_tau, beta)	Fourier integration of the real Green's function *gf_tau*.
tau2iw(gf_tau, beta[, n_pole, moments, fourier])	Fourier transform of the real Green's function *gf_tau*.
tau2iw_dft(gf_tau, beta)	Discrete Fourier transform of the real Green's function *gf_tau*.
tau2iw_ft_lin(gf_tau, beta)	Fourier integration of the real Green's function *gf_tau*.
tau2izp(gf_tau, beta, izp[, moments, occ, ...])	Fourier transform of the real Green's function *gf_tau* to *izp*.
tt2z(tt, gf_t, z[, laplace])	Laplace transform of the real-time Green's function *gf_t*.
tt2z_herm2(tt, gf_t, z[, herm2, quad])	Square Fourier-Padé transform of the real-time Green's function *gf_t*.
tt2z_lin(tt, gf_t, z)	Laplace transform of the real-time Green's function *gf_t*.
tt2z_lpz(tt, gf_t, z[, order, quad])	Linear prediction Z-transform of the real-time Green's function *gf_t*.
tt2z_pade(tt, gf_t, z[, degree, pade, quad])	Fourier-Padé transform of the real-time Green's function *gf_t*.
tt2z_simps(tt, gf_t, z)	Laplace transform of the real-time Green's function *gf_t*.
tt2z_trapz(tt, gf_t, z)	Laplace transform of the real-time Green's function *gf_t*.

gftool.fourier.iw2tau

gftool.fourier.**iw2tau**(*gf_iw*, *beta*, *moments=(1.0,)*, *fourier=<function iw2tau_dft>*, *n_fit=0*)

Discrete Fourier transform of the Hermitian Green's function *gf_iw*.

Fourier transformation of a fermionic Matsubara Green's function to imaginary-time domain. We assume a Hermitian Green's function *gf_iw*, i.e. $G(-i_n) = G^*(i_n)$, which is the case for commutator Green's functions $G_{AB}() = \langle A()B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_tau* is then real.

Parameters

gf_iw
[(…, N_iw) complex np.ndarray] The Green's function at positive **fermionic** Matsubara frequencies i_n.

beta
[float] The inverse temperature $beta = 1/k_B T$.

moments
[(…, m) float array_like] High-frequency moments of *gf_iw*.

fourier
[{*iw2tau_dft*, *iw2tau_dft_soft*}, optional] Back-end to perform the actual Fourier transformation.

n_fit
[int, optional] Number of additionally fitted moments (in fact, *gf_iw* is fitted, not not directly moments).

Returns

(…, 2*N_iw + 1) float np.ndarray
The Fourier transform of *gf_iw* for imaginary times $\in [0,]$.

See also:

iw2tau_dft
Back-end: plain implementation of Fourier transform.

iw2tau_dft_soft
Back-end: Fourier transform with artificial softening of oszillations.

pole_gf_from_moments
Function handling the given *moments*.

Notes

For accurate an accurate Fourier transform, it is necessary, that *gf_iw* has already reached it's high-frequency behaviour, which need to be included explicitly. Therefore, the accuracy of the FT depends implicitely on the bandwidth!

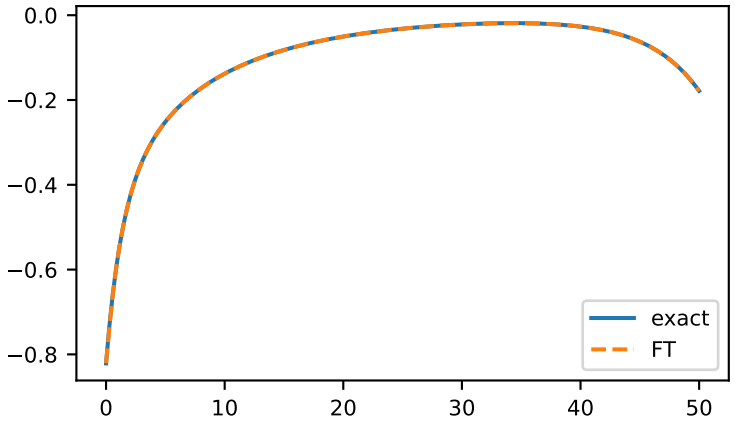
Examples

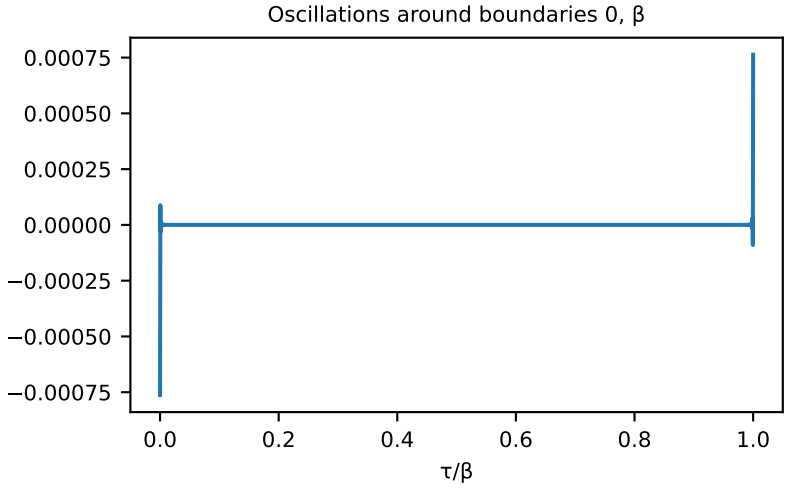
```
>>> BETA = 50
>>> iws = gt.matsubara_frequencies(range(1024), beta=BETA)
>>> tau = np.linspace(0, BETA, num=2*iws.size + 1, endpoint=True)
```

```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_iw = gt.pole_gf_z(iws, poles=poles, weights=weights)
>>> gf_dft = gt.fourier.iw2tau(gf_iw, beta=BETA)
>>> gf_iw.size, gf_dft.size
(1024, 2049)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(tau, gf_tau, label='exact')
>>> __ = plt.plot(tau, gf_dft, '--', label='FT')
>>> __ = plt.legend()
>>> plt.show()
```

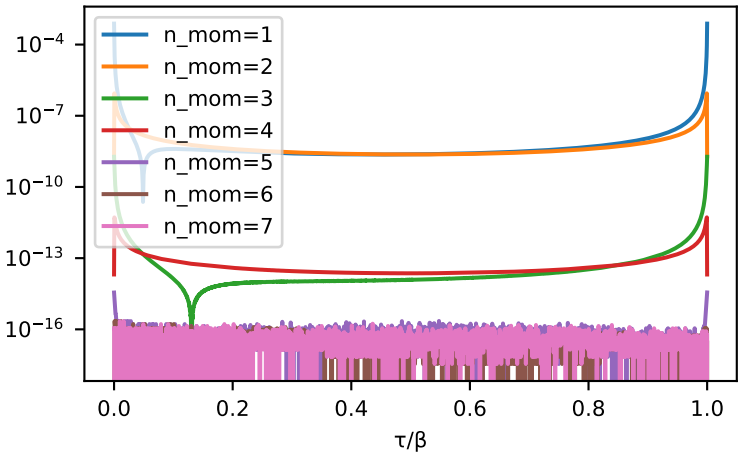
```
>>> __ = plt.title('Oscillations around boundaries 0, β')
>>> __ = plt.plot(tau/BETA, gf_tau - gf_dft)
>>> __ = plt.xlabel('τ/β')
>>> plt.show()
```


0.0
−0.2
−0.4
−0.6
−0.8
0
10
20
30
40
50
exact
FT


Oscillations around boundaries 0, β
0.00075
0.00050
0.00025
0.00000
−0.00025
−0.00050
−0.00075
0.0
0.2
0.4
0.6
0.8
1.0
τ/β

Results can be drastically improved giving high-frequency moments, this reduces the truncation error.

```
>>> mom = np.sum(weights[:, np.newaxis] * poles[:, np.newaxis]**range(8), axis=0)
>>> for n in range(1, 8):
...     gf = gt.fourier.iw2tau(gf_iw, moments=mom[:n], beta=BETA)
...     __ = plt.plot(tau/BETA, abs(gf_tau - gf), label=f'n_mom={n}')
>>> __ = plt.legend()
>>> __ = plt.xlabel('τ/β')
>>> plt.yscale('log')
>>> plt.show()
```



The method is resistant against noise:

```
>>> magnitude = 2e-7
>>> noise = np.random.normal(scale=magnitude, size=gf_iw.size)
>>> for n in range(1, 7, 2):
...     gf = gt.fourier.iw2tau(gf_iw+noise, moments=mom[:n], beta=BETA)
...     __ = plt.plot(tau/BETA, abs(gf_tau - gf), '--', label=f'n_mom={n}')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(tau/BETA, abs(gf_tau - gf_dft), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

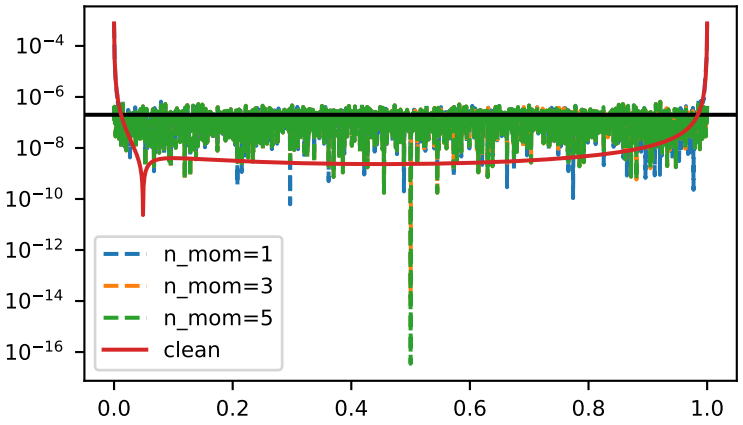
gftool.fourier.iw2tau_dft

gftool.fourier.**iw2tau_dft**(*gf_iw*, *beta*)

Discrete Fourier transform of the Hermitian Green's function *gf_iw*.

Fourier transformation of a fermionic Matsubara Green's function to imaginary-time domain. The infinite Fourier sum is truncated. We assume a Hermitian Green's function *gf_iw*, i.e. $G(-i_n) = G^*(i_n)$, which is the case for commutator Green's functions $G_{AB}() = \langle A()B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_tau* is then real.

Parameters



gf_iw
: [(…, N_iw) complex np.ndarray] The Green's function at positive **fermionic** Matsubara frequencies i_n.

beta
: [float] The inverse temperature $beta = 1/k_B T$.

Returns

(…, 2*N_iw + 1) float np.ndarray
: The Fourier transform of *gf_iw* for imaginary times $\in [0,]$.

See also:

iw2tau_dft_soft
: Fourier transform with artificial softening of oszillations.

Notes

For accurate an accurate Fourier transform, it is necessary, that *gf_iw* has already reached it's high-frequency behaviour, which need to be included explicitly. Therefore, the accuracy of the FT depends implicitely on the bandwidth!

Examples

```
>>> BETA = 50
>>> iws = gt.matsubara_frequencies(range(1024), beta=BETA)
>>> tau = np.linspace(0, BETA, num=2*iws.size + 1, endpoint=True)
```

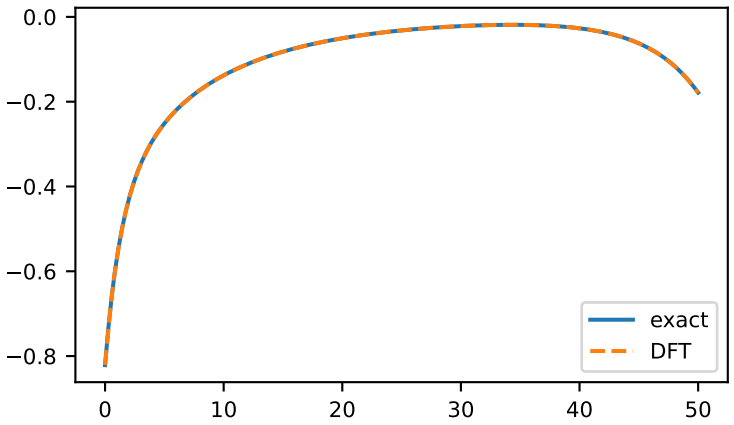
```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_iw = gt.pole_gf_z(iws, poles=poles, weights=weights)
>>> # 1/z tail has to be handled manually
```

(continues on next page)

(continued from previous page)

```
>>> gf_dft = gt.fourier.iw2tau_dft(gf_iw - 1/iws, beta=BETA) - .5
>>> gf_iw.size, gf_dft.size
(1024, 2049)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
```

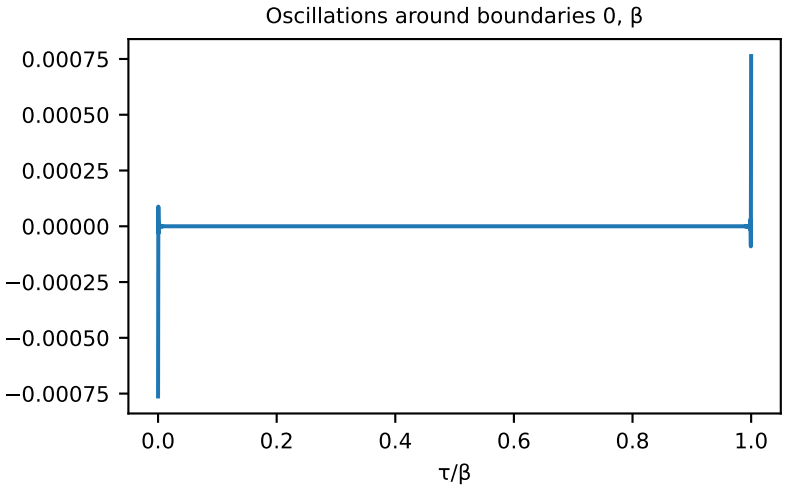
```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(tau, gf_tau, label='exact')
>>> __ = plt.plot(tau, gf_dft, '--', label='DFT')
>>> __ = plt.legend()
>>> plt.show()
```

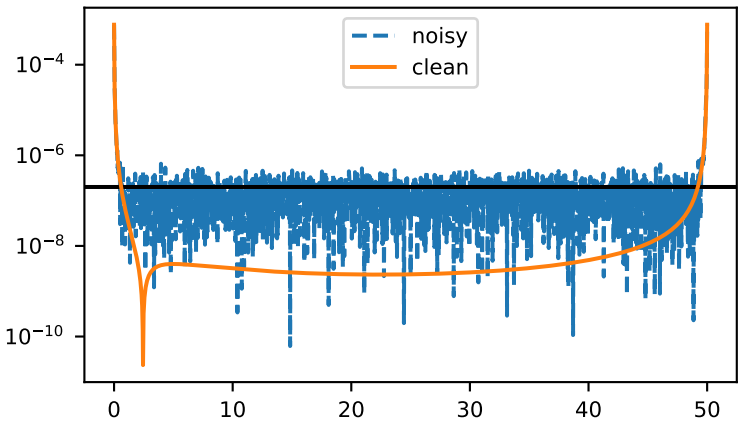



```
>>> __ = plt.title('Oscillations around boundaries 0, β')
>>> __ = plt.plot(tau/BETA, gf_tau - gf_dft)
>>> __ = plt.xlabel('τ/β')
>>> plt.show()
```

The method is resistant against noise:

```
>>> magnitude = 2e-7
>>> noise = np.random.normal(scale=magnitude, size=gf_iw.size)
>>> gf_dft_noisy = gt.fourier.iw2tau_dft(gf_iw + noise - 1/iws, beta=BETA) - .5
>>> __ = plt.plot(tau, abs(gf_tau - gf_dft_noisy), '--', label='noisy')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(tau, abs(gf_tau - gf_dft), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```


Oscillations around boundaries 0, β
0.00075
0.00050
0.00025
0.00000
−0.00025
−0.00050
−0.00075
0.0
0.2
0.4
0.6
0.8
1.0
τ/β


noisy
clean
10^{-4}
10^{-6}
10^{-8}
10^{-10}
0
10
20
30
40
50

gftool.fourier.iw2tau_dft_soft

gftool.fourier.**iw2tau_dft_soft**(*gf_iw*, *beta*)

Discrete Fourier transform of the Hermitian Green's function *gf_iw*.

Fourier transformation of a fermionic Matsubara Green's function to imaginary-time domain. Add a tail letting *gf_iw* go to 0. The tail is just a cosine function to exactly hit the 0. This is unphysical but suppresses oscillations. This methods should be used with care, as it might hide errors. We assume a Hermitian Green's function *gf_iw*, i.e. $G(-i_n) = G^*(i_n)$, which is the case for commutator Green's functions $G_{AB}() = \langle A()B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_tau* is then real.

Parameters

gf_iw
[(…, N_iw) complex np.ndarray] The Green's function at positive **fermionic** Matsubara frequencies i_n.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

(…, 2*N_iw + 1) float np.ndarray
The Fourier transform of *gf_iw* for imaginary times $\in [0,]$.

See also:

iw2tau_dft
Plain implementation of Fourier transform.

Notes

For accurate an accurate Fourier transform, it is necessary, that *gf_iw* has already reached it's high-frequency behaviour, which need to be included explicitly. Therefore, the accuracy of the FT depends implicitely on the bandwidth!

Examples

```
>>> BETA = 50
>>> iws = gt.matsubara_frequencies(range(1024), beta=BETA)
>>> tau = np.linspace(0, BETA, num=2*iws.size + 1, endpoint=True)
```

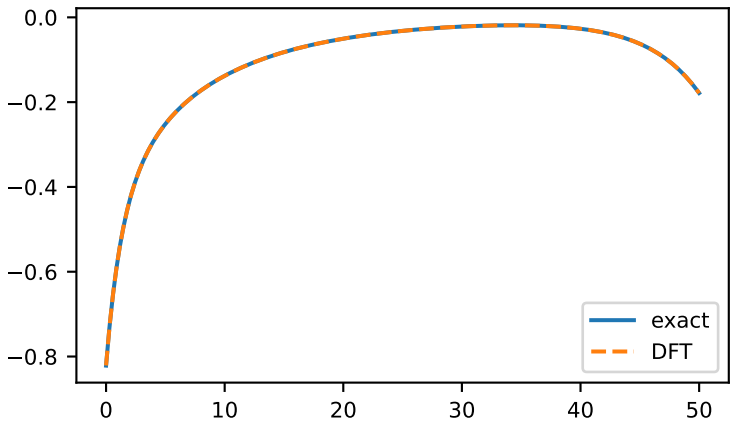
```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_iw = gt.pole_gf_z(iws, poles=poles, weights=weights)
>>> # 1/z tail has to be handled manually
>>> gf_dft = gt.fourier.iw2tau_dft_soft(gf_iw - 1/iws, beta=BETA) - .5
>>> gf_iw.size, gf_dft.size
(1024, 2049)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(tau, gf_tau, label='exact')
>>> __ = plt.plot(tau, gf_dft, '--', label='DFT')
```

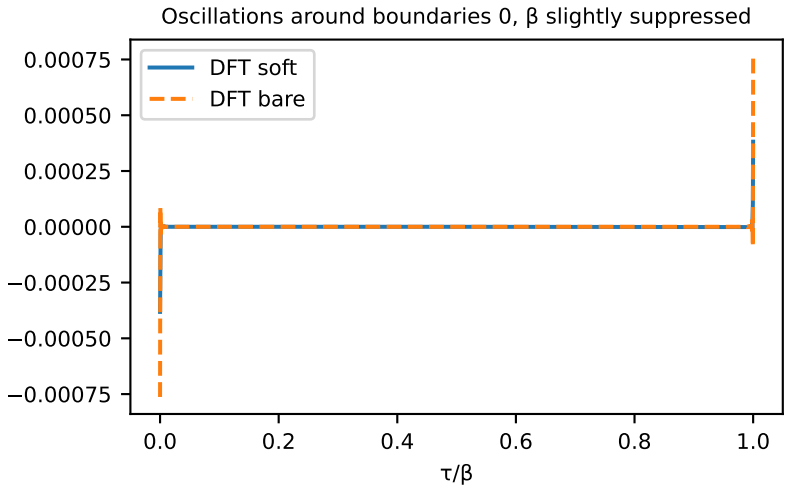
(continues on next page)

(continued from previous page)

```
>>> __ = plt.legend()
>>> plt.show()
```

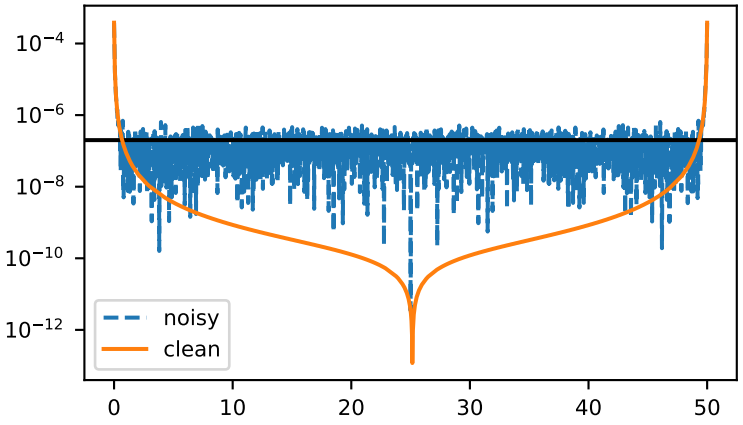



```
>>> __ = plt.title('Oscillations around boundaries 0, β slightly suppressed')
>>> __ = plt.plot(tau/BETA, gf_tau - gf_dft, label='DFT soft')
>>> gf_dft_bare = gt.fourier.iw2tau_dft(gf_iw - 1/iws, beta=BETA) - .5
>>> __ = plt.plot(tau/BETA, gf_tau - gf_dft_bare, '--',  label='DFT bare')
>>> __ = plt.legend()
>>> __ = plt.xlabel('τ/β')
>>> plt.show()
```



The method is resistant against noise:

```
>>> magnitude = 2e-7
>>> noise = np.random.normal(scale=magnitude, size=gf_iw.size)
>>> gf_dft_noisy = gt.fourier.iw2tau_dft_soft(gf_iw + noise - 1/iws, beta=BETA) -␣
↪.5
>>> __ = plt.plot(tau, abs(gf_tau - gf_dft_noisy), '--', label='noisy')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(tau, abs(gf_tau - gf_dft), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```



gftool.fourier.izp2tau

`gftool.fourier.`**`izp2tau`**(*izp*, *gf_izp*, *tau*, *beta*, *moments=(1.0,)*)

Fourier transform of the Hermitian Green's function *gf_izp* to *tau*.

Fourier transformation of a fermionic Padé Green's function to imaginary-time domain. We assume a Hermitian Green's function *gf_izp*, i.e. $G(-i_n) = G^*(i_n)$, which is the case for commutator Green's functions $G_{AB}() = \langle A()B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_tau* is then real.

TODO: this function is not vectorized yet.

Parameters

izp, gf_izp
: [(N_izp) float np.ndarray] Positive **fermionic** Padé frequencies iz_p and the Green's function at specified frequencies.

tau
: [(N_tau) float np.ndarray] Imaginary times *0* <= *tau* <= *beta* at which the Fourier transform is evaluated.

beta
: [float] The inverse temperature $beta = 1/k_B T$.

moments
: [(m) float array_like, optional] High-frequency moments of *gf_izp*.

Returns

(N_tau) float np.ndarray
The Fourier transform of *gf_izp* for imaginary times *tau*.

See also:

iw2tau
Fourier transform from fermionic Matsubara frequencies.

_z2polegf
Function handling the fitting of *gf_izp*.

Notes

The algorithm performs in fact an analytic continuation instead of a Fourier integral. It is however only evaluated on the imaginary axis, so far the algorithm was observed to be stable

Examples

```
>>> BETA = 50
>>> izp, __ = gt.pade_frequencies(50, beta=BETA)
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
```

```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_izp = gt.pole_gf_z(izp, poles=poles, weights=weights)
>>> gf_ft = gt.fourier.izp2tau(izp, gf_izp, tau, beta=BETA)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
```

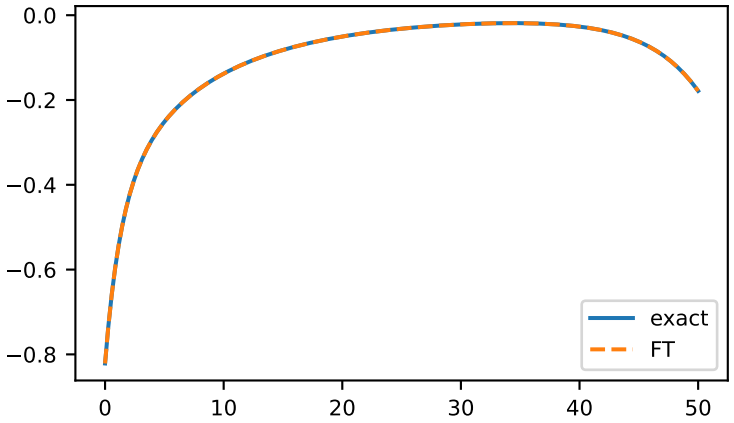
```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(tau, gf_tau, label='exact')
>>> __ = plt.plot(tau, gf_ft, '--', label='FT')
>>> __ = plt.legend()
>>> plt.show()
```

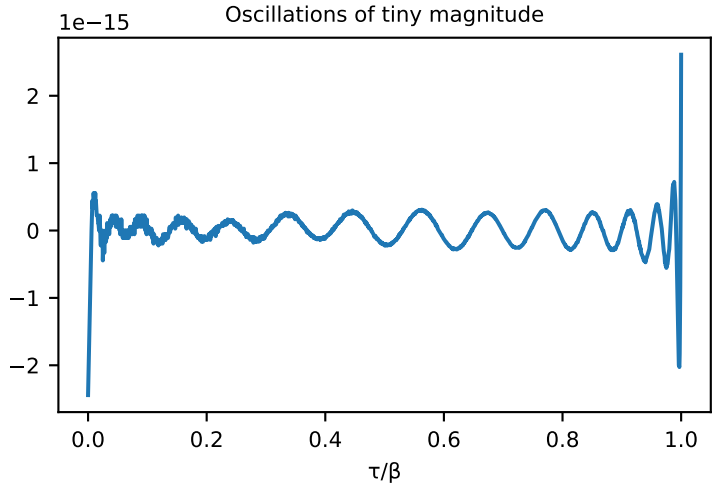
```
>>> __ = plt.title('Oscillations of tiny magnitude')
>>> __ = plt.plot(tau/BETA, gf_tau - gf_ft)
>>> __ = plt.xlabel('τ/β')
>>> plt.show()
```

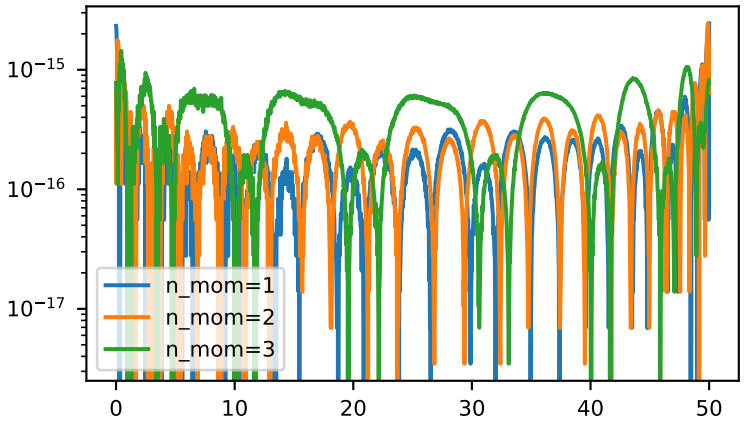
Results of *izp2tau* can be improved giving high-frequency moments.

```
>>> mom = np.sum(weights[:, np.newaxis] * poles[:, np.newaxis]**range(4), axis=0)
>>> for n in range(1, 4):
...     gf = gt.fourier.izp2tau(izp, gf_izp, tau, beta=BETA, moments=mom[:n])
...     __ = plt.plot(tau, abs(gf_tau - gf), label=f'n_mom={n}')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

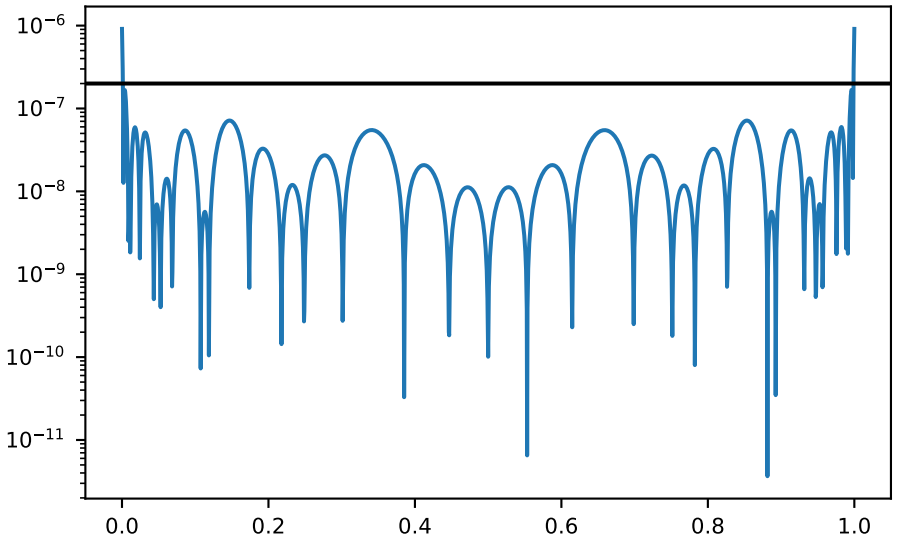
The method is resistant against noise:


0.0
−0.2
−0.4
−0.6
−0.8
0
10
20
30
40
50
exact
FT


Oscillations of tiny magnitude
1e−15
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0
τ/β




```
>>> magnitude = 2e-7
>>> noise = np.random.normal(scale=magnitude, size=gf_izp.size)
>>> gf = gt.fourier.izp2tau(izp, gf_izp + noise, tau, beta=BETA, moments=(1,))
>>> __ = plt.plot(tau/BETA, abs(gf_tau - gf))
>>> __ = plt.axhline(magnitude, color='black')
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```

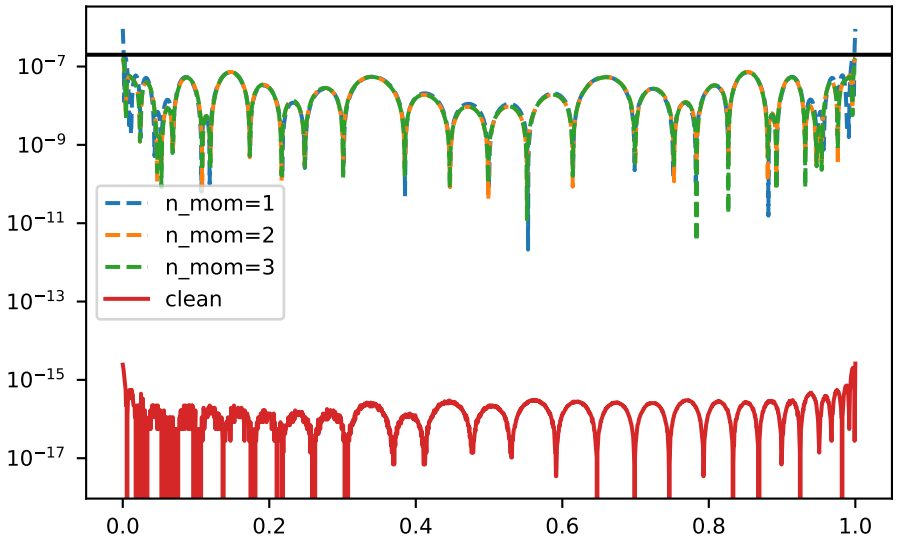



```
>>> for n in range(1, 4):
...     gf = gt.fourier.izp2tau(izp, gf_izp + noise, tau, beta=BETA,␣
↪moments=mom[:n])
...     __ = plt.plot(tau/BETA, abs(gf_tau - gf), '--', label=f'n_mom={n}')
>>> __ = plt.axhline(magnitude, color='black')
```

(continues on next page)

(continued from previous page)

```
>>> __ = plt.plot(tau/BETA, abs(gf_tau - gf_ft), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```



gftool.fourier.tau2iv

`gftool.fourier.`**`tau2iv`**(*gf_tau*, *beta*, *fourier=<function tau2iv_ft_lin>*)

Fourier transformation of the real Green's function *gf_tau*.

Fourier transformation of a bosonic imaginary-time Green's function to Matsubara domain. We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A() B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iv* is then Hermitian. This function removes the discontinuity $G_{AB}() - G_{AB}(0) = \langle [A, B]\rangle$.

TODO: if high-frequency moments are know, they should be stripped for increased accuracy.

Parameters

gf_tau
[(…, N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.

fourier
[{*tau2iv_ft_lin*, *tau2iv_dft*}, optional] Back-end to perform the actual Fourier transformation.

Returns

(…, (N_iv + 1)/2) complex np.ndarray
The Fourier transform of *gf_tau* for non-negative bosonic Matsubara frequencies i_n.

See also:

tau2iv_dft
Back-end: plain implementation using Riemann sum.

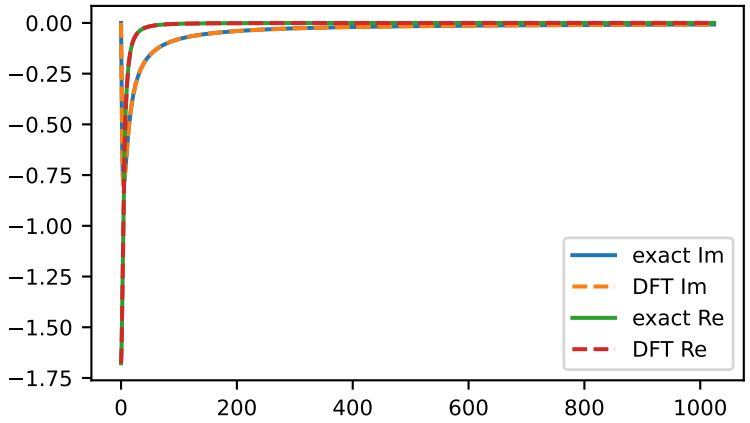
tau2iv_ft_lin
Back-end: Fourier integration using Filon's method.

Examples

```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> ivs = gt.matsubara_frequencies_b(range((tau.size+1)//2), beta=BETA)
```

```
>>> poles, weights = np.random.random(10), np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau_b(tau, poles=poles, weights=weights, beta=BETA)
>>> gf_ft = gt.fourier.tau2iv(gf_tau, beta=BETA)
>>> gf_tau.size, gf_ft.size
(2049, 1025)
>>> gf_iv = gt.pole_gf_z(ivs, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_iv.imag, label='exact Im')
>>> __ = plt.plot(gf_ft.imag, '--', label='DFT Im')
>>> __ = plt.plot(gf_iv.real, label='exact Re')
>>> __ = plt.plot(gf_ft.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.show()
```



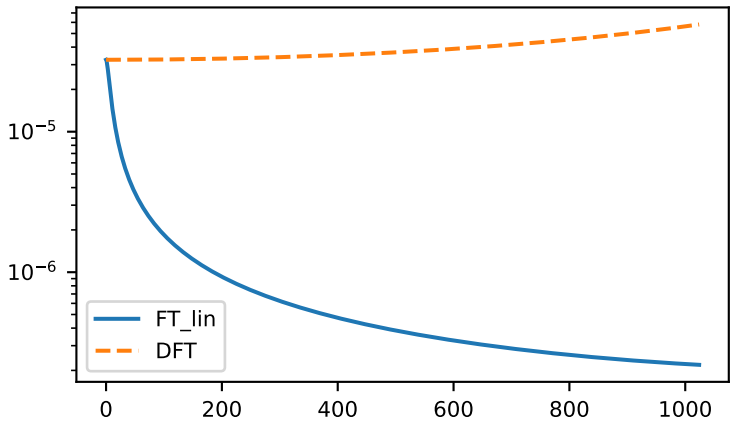
Accuracy of the different back-ends

```
>>> ft_lin, dft = gt.fourier.tau2iv_ft_lin, gt.fourier.tau2iv_dft
>>> gf_ft_lin = gt.fourier.tau2iv(gf_tau, beta=BETA, fourier=ft_lin)
>>> gf_dft = gt.fourier.tau2iv(gf_tau, beta=BETA, fourier=dft)
>>> __ = plt.plot(abs(gf_iv - gf_ft_lin), label='FT_lin')
>>> __ = plt.plot(abs(gf_iv - gf_dft), '--', label='DFT')
```

(continues on next page)

(continued from previous page)

```
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```



The methods are resistant against noise:

```
>>> magnitude = 5e-6
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> gf_ft_lin_noisy = gt.fourier.tau2iv(gf_tau + noise, beta=BETA, fourier=ft_lin)
>>> gf_dft_noisy = gt.fourier.tau2iv(gf_tau + noise, beta=BETA, fourier=dft)
>>> __ = plt.plot(abs(gf_iv - gf_ft_lin_noisy), '--', label='FT_lin')
>>> __ = plt.plot(abs(gf_iv - gf_dft_noisy), '--', label='DFT')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

gftool.fourier.tau2iv_dft

`gftool.fourier.`**`tau2iv_dft`**(*gf_tau*, *beta*)

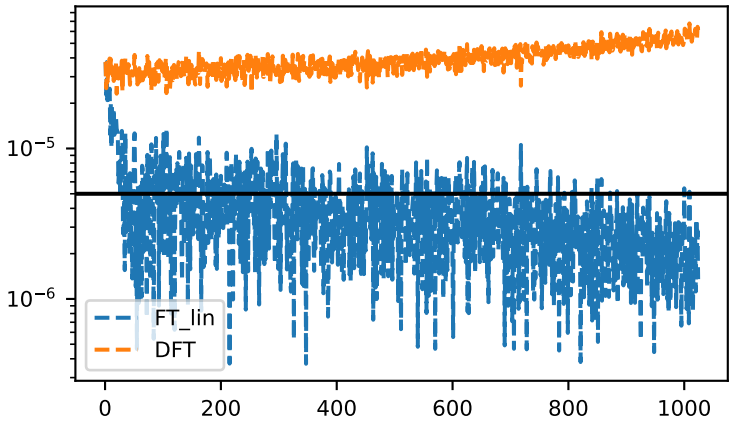
Discrete Fourier transform of the real Green's function *gf_tau*.

Fourier transformation of a bosonic imaginary-time Green's function to Matsubara domain. The Fourier integral is replaced by a Riemann sum giving a discrete Fourier transform (DFT). We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A() B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iv* is then Hermitian.

Parameters

gf_tau
[(…, N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.



Returns

(…, (N_iv + 1)/2) float np.ndarray
The Fourier transform of *gf_tau* for non-negative bosonic Matsubara frequencies i_n.

See also:

tau2iv_ft_lin
Fourier integration using Filon's method.

Examples

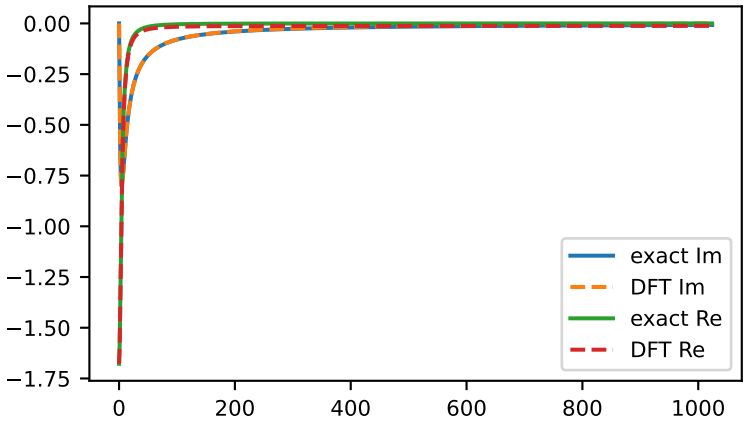
```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> ivs = gt.matsubara_frequencies_b(range((tau.size+1)//2), beta=BETA)
```

```
>>> poles, weights = np.random.random(10), np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
>>> gf_dft = gt.fourier.tau2iv_dft(gf_tau, beta=BETA)
>>> gf_tau.size, gf_dft.size
(2049, 1025)
>>> gf_iv = gt.pole_gf_z(ivs, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_iv.imag, label='exact Im')
>>> __ = plt.plot(gf_dft.imag, '--', label='DFT Im')
>>> __ = plt.plot(gf_iv.real, label='exact Re')
>>> __ = plt.plot(gf_dft.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.show()
```

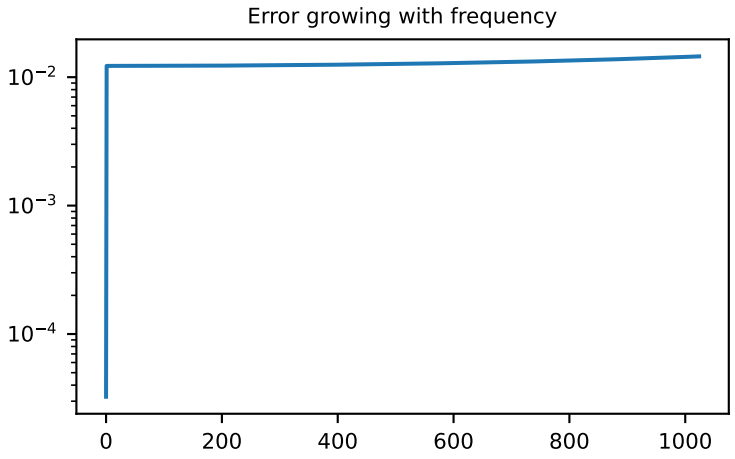
```
>>> __ = plt.title('Error growing with frequency')
>>> __ = plt.plot(abs(gf_iv - gf_dft))
```

(continues on next page)



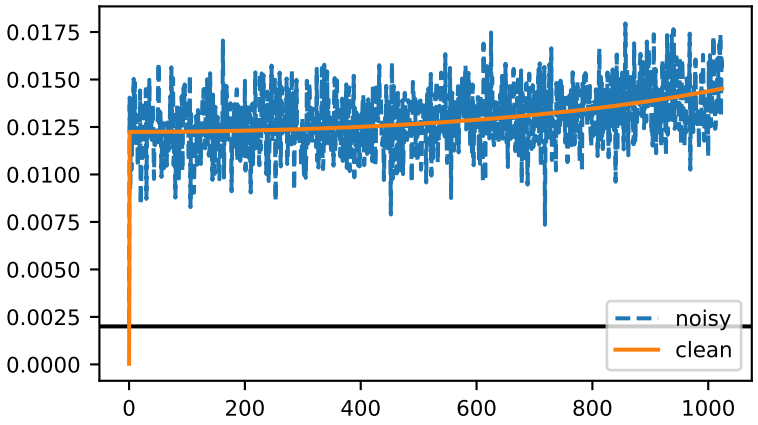
(continued from previous page)

```
>>> plt.yscale('log')
>>> plt.show()
```



The method is resistant against noise:

```
>>> magnitude = 2e-3
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> gf_dft_noisy = gt.fourier.tau2iv_dft(gf_tau + noise, beta=BETA)
>>> __ = plt.plot(abs(gf_iv - gf_dft_noisy), '--', label='noisy')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(abs(gf_iv - gf_dft), label='clean')
>>> __ = plt.legend()
>>> # plt.yscale('log')
>>> plt.show()
```



gftool.fourier.tau2iv_ft_lin

gftool.fourier.**tau2iv_ft_lin**(*gf_tau*, *beta*)

Fourier integration of the real Green's function *gf_tau*.

Fourier transformation of a bosonic imaginary-time Green's function to Matsubara domain. We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A() B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iv* is then Hermitian. Filon's method is used to calculated the Fourier integral

$$G^n = \int_0 dG() e^{i_n},$$

$G()$ is approximated by a linear spline. A linear approximation was chosen to be able to integrate noisy functions. Information on oscillatory integrations can be found e.g. in [filon1930] and [iserles2006].

Parameters

gf_tau
[(…, N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

(…, (N_iv + 1)/2) float np.ndarray
The Fourier transform of *gf_tau* for non-negative bosonic Matsubara frequencies i_n.

See also:

tau2iv_dft
Plain implementation using Riemann sum.

References

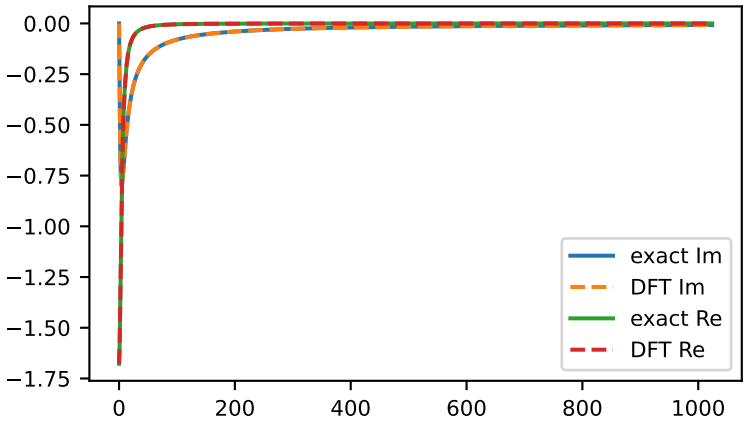
[filon1930], [iserles2006]

Examples

```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> ivs = gt.matsubara_frequencies_b(range((tau.size+1)//2), beta=BETA)
```

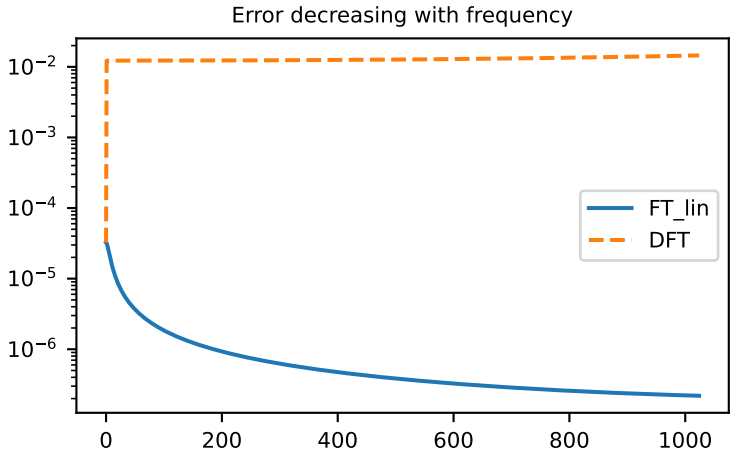
```
>>> poles, weights = np.random.random(10), np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau_b(tau, poles=poles, weights=weights, beta=BETA)
>>> gf_ft_lin = gt.fourier.tau2iv_ft_lin(gf_tau, beta=BETA)
>>> gf_tau.size, gf_ft_lin.size
(2049, 1025)
>>> gf_iv = gt.pole_gf_z(ivs, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_iv.imag, label='exact Im')
>>> __ = plt.plot(gf_ft_lin.imag, '--', label='DFT Im')
>>> __ = plt.plot(gf_iv.real, label='exact Re')
>>> __ = plt.plot(gf_ft_lin.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.show()
```

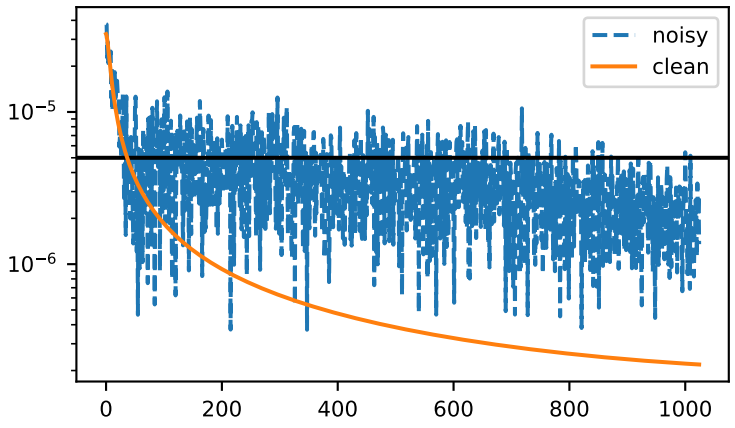



```
>>> __ = plt.title('Error decreasing with frequency')
>>> __ = plt.plot(abs(gf_iv - gf_ft_lin), label='FT_lin')
>>> gf_dft = gt.fourier.tau2iv_dft(gf_tau, beta=BETA)
>>> __ = plt.plot(abs(gf_iv - gf_dft), '--', label='DFT')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

The method is resistant against noise:




```
>>> magnitude = 5e-6
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> gf_ft_noisy = gt.fourier.tau2iv_ft_lin(gf_tau + noise, beta=BETA)
>>> __ = plt.plot(abs(gf_iv - gf_ft_noisy), '--', label='noisy')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(abs(gf_iv - gf_ft_lin), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```



gftool.fourier.tau2iw

gftool.fourier.**tau2iw**(*gf_tau*, *beta*, *n_pole=None*, *moments=None*, *fourier=<function tau2iw_ft_lin>*)

Fourier transform of the real Green's function *gf_tau*.

Fourier transformation of a fermionic imaginary-time Green's function to Matsubara domain. We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A() B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iw* is then Hermitian. If no explicit *moments* are given, this function removes $-G_{AB}() - G_{AB}(0) = \langle [A, B]\rangle$.

Parameters

gf_tau
[(…, N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.

n_pole
[int, optional] Number of poles used to fit *gf_tau*. Needs to be at least as large as the number of given moments *m* (default: no fitting is performed).

moments
[(…, m) float array_like, optional] High-frequency moments of *gf_iw*. If none are given, the first moment is chosen to remove the discontinuity at $= 0^{\pm}$.

fourier
[{*tau2iw_ft_lin*, *tau2iw_dft*}, optional] Back-end to perform the actual Fourier transformation.

Returns

(…, (N_iv + 1)/2) complex np.ndarray
The Fourier transform of *gf_tau* for non-negative fermionic Matsubara frequencies i_n.

See also:

tau2iw_ft_lin
Back-end: Fourier integration using Filon's method.

tau2iw_dft
Back-end: plain implementation using Riemann sum.

pole_gf_from_tau
Function handling the fitting of *gf_tau*.

Examples

```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> iws = gt.matsubara_frequencies(range((tau.size-1)//2), beta=BETA)
```

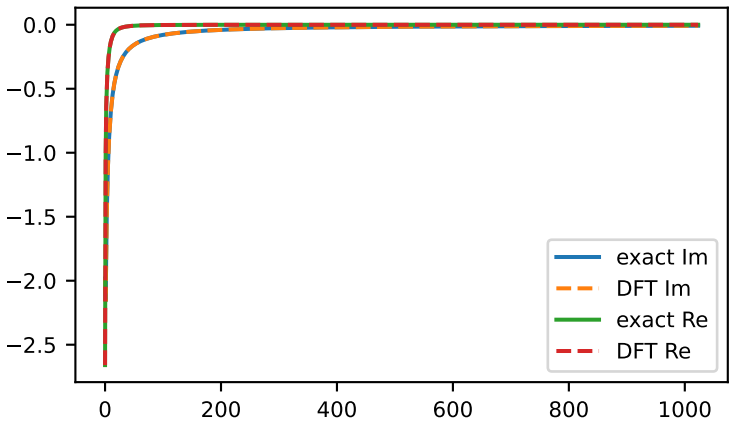
```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
>>> gf_ft = gt.fourier.tau2iw(gf_tau, beta=BETA)
>>> gf_tau.size, gf_ft.size
```

(continues on next page)

(continued from previous page)

```
(2049, 1024)
>>> gf_iw = gt.pole_gf_z(iws, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_iw.imag, label='exact Im')
>>> __ = plt.plot(gf_ft.imag, '--', label='DFT Im')
>>> __ = plt.plot(gf_iw.real, label='exact Re')
>>> __ = plt.plot(gf_ft.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.show()
```



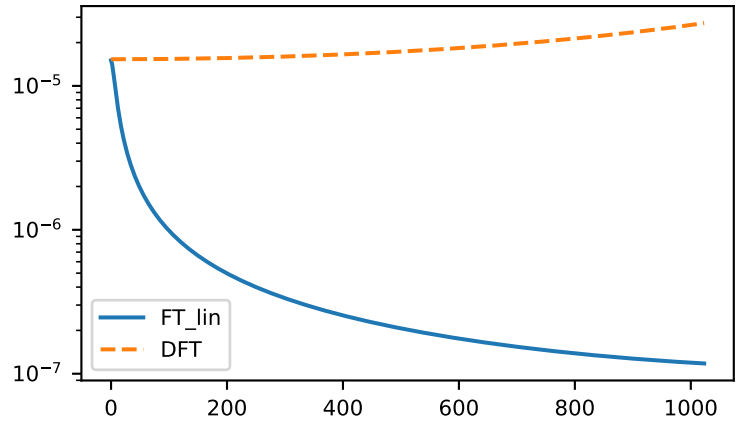
Accuracy of the different back-ends

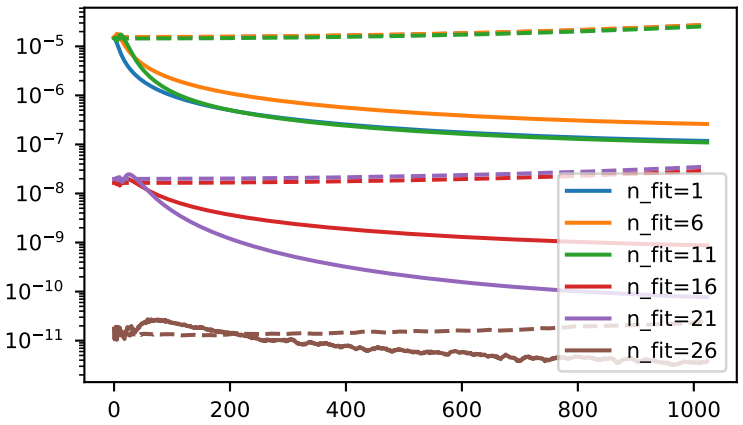
```
>>> ft_lin, dft = gt.fourier.tau2iw_ft_lin, gt.fourier.tau2iw_dft
>>> gf_ft_lin = gt.fourier.tau2iw(gf_tau, beta=BETA, fourier=ft_lin)
>>> gf_dft = gt.fourier.tau2iw(gf_tau, beta=BETA, fourier=dft)
>>> __ = plt.plot(abs(gf_iw - gf_ft_lin), label='FT_lin')
>>> __ = plt.plot(abs(gf_iw - gf_dft), '--', label='DFT')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

The accuracy can be further improved by fitting as suitable pole Green's function:

```
>>> for n, n_mom in enumerate(range(1, 30, 5)):
...     gf = gt.fourier.tau2iw(gf_tau, n_pole=n_mom, moments=(1,), beta=BETA,␣
↪fourier=ft_lin)
...     __ = plt.plot(abs(gf_iw - gf), label=f'n_fit={n_mom}', color=f'C{n}')
...     gf = gt.fourier.tau2iw(gf_tau, n_pole=n_mom, moments=(1,), beta=BETA,␣
↪fourier=dft)
...     __ = plt.plot(abs(gf_iw - gf), '--', color=f'C{n}')
>>> __ = plt.legend(loc='lower right')
>>> plt.yscale('log')
>>> plt.show()
```

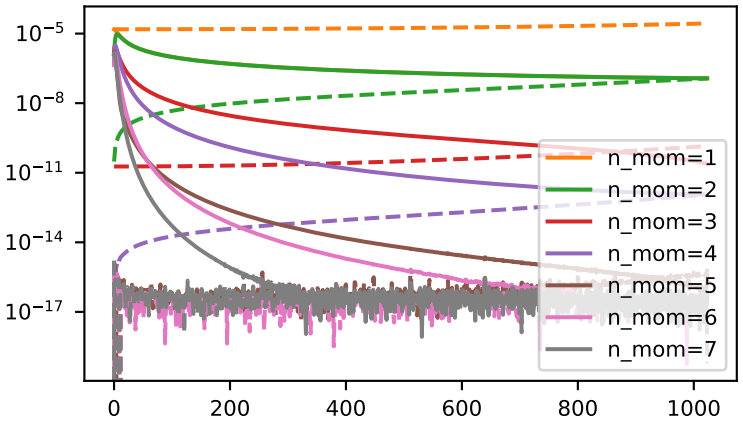
Results for DFT can be drastically improved giving high-frequency moments. The reason is, that lower large


10^{-5}
10^{-6}
10^{-7}
FT_lin
DFT
0
200
400
600
800
1000


10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{-11}
n_fit=1
n_fit=6
n_fit=11
n_fit=16
n_fit=21
n_fit=26
0
200
400
600
800
1000

frequencies, where FT_lin is superior, are treated by the moments instead of the Fourier transform.

```
>>> mom = np.sum(weights[:, np.newaxis] * poles[:, np.newaxis]**range(8), axis=0)
>>> for n in range(1, 8):
...     gf = gt.fourier.tau2iw(gf_tau, moments=mom[:n], beta=BETA, fourier=ft_lin)
...     __ = plt.plot(abs(gf_iw - gf), label=f'n_mom={n}', color=f'C{n}')
...     gf = gt.fourier.tau2iw(gf_tau, moments=mom[:n], beta=BETA, fourier=dft)
...     __ = plt.plot(abs(gf_iw - gf), '--', color=f'C{n}')
>>> __ = plt.legend(loc='lower right')
>>> plt.yscale('log')
>>> plt.show()
```

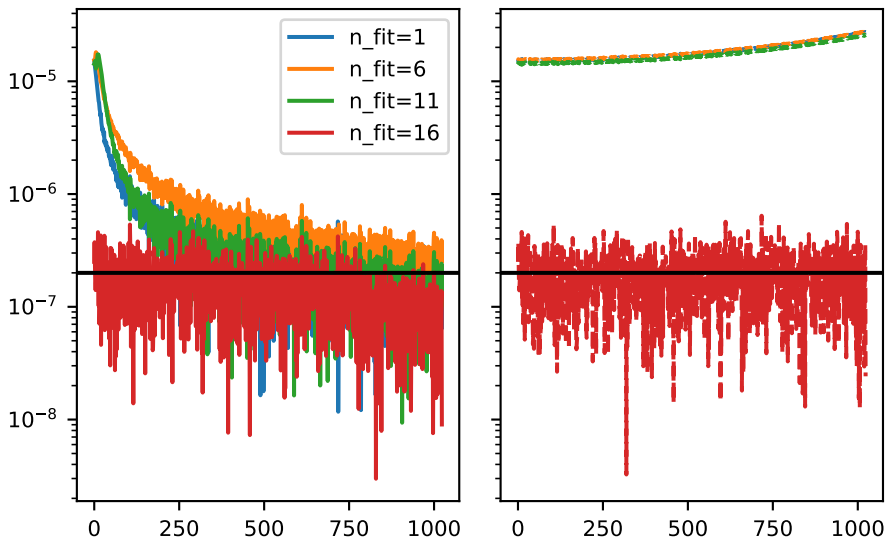


The method is resistant against noise:

```
>>> magnitude = 2e-7
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> __, axes = plt.subplots(ncols=2, sharey=True)
>>> for n, n_mom in enumerate(range(1, 20, 5)):
...     gf = gt.fourier.tau2iw(gf_tau + noise, n_pole=n_mom, moments=(1,),
...                            beta=BETA, fourier=ft_lin)
...     __ = axes[0].plot(abs(gf_iw - gf), label=f'n_fit={n_mom}', color=f'C{n}')
...     gf = gt.fourier.tau2iw(gf_tau + noise, n_pole=n_mom, moments=(1,),
...                            beta=BETA, fourier=dft)
...     __ = axes[1].plot(abs(gf_iw - gf), '--', color=f'C{n}')
>>> for ax in axes:
...     __ = ax.axhline(magnitude, color='black')
>>> __ = axes[0].legend()
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```

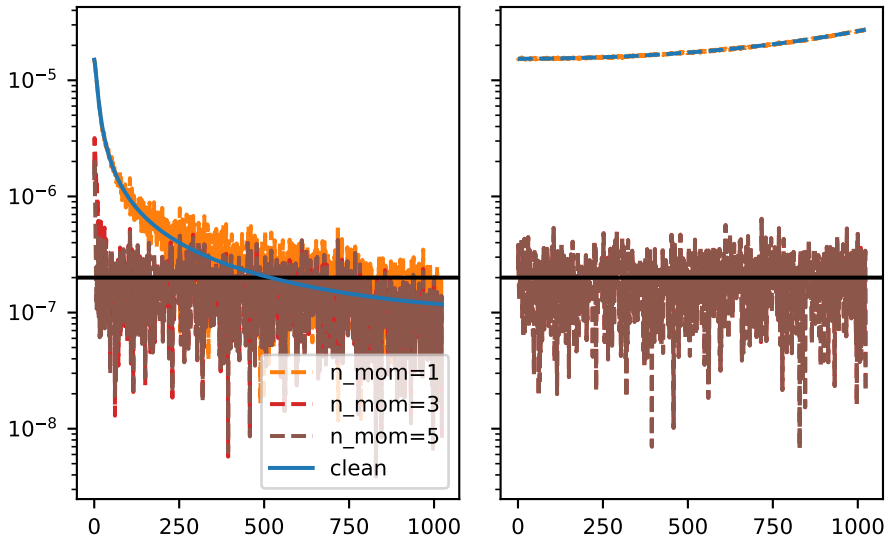
```
>>> __, axes = plt.subplots(ncols=2, sharey=True)
>>> for n in range(1, 7, 2):
...     gf = gt.fourier.tau2iw(gf_tau + noise, moments=mom[:n], beta=BETA,␣
↪fourier=ft_lin)
...     __ = axes[0].plot(abs(gf_iw - gf), '--', label=f'n_mom={n}', color=f'C{n}
↪')
```

(continues on next page)



(continued from previous page)

```
...     gf = gt.fourier.tau2iw(gf_tau + noise, moments=mom[:n], beta=BETA,␣
↪fourier=dft)
...     __ = axes[1].plot(abs(gf_iw - gf), '--', color=f'C{n}')
>>> for ax in axes:
...     __ = ax.axhline(magnitude, color='black')
>>> __ = axes[0].plot(abs(gf_iw - gf_ft_lin), label='clean')
>>> __ = axes[1].plot(abs(gf_iw - gf_dft), '--', label='clean')
>>> __ = axes[0].legend(loc='lower right')
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```



gftool.fourier.tau2iw_dft

gftool.fourier.**tau2iw_dft**(*gf_tau*, *beta*)

Discrete Fourier transform of the real Green's function *gf_tau*.

Fourier transformation of a fermionic imaginary-time Green's function to Matsubara domain. The Fourier integral is replaced by a Riemann sum giving a discrete Fourier transform (DFT). We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A()B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iw* is then Hermitian.

Parameters

gf_tau
[(…, N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

(…, (N_iw - 1)/2) float np.ndarray
The Fourier transform of *gf_tau* for positive fermionic Matsubara frequencies i_n.

See also:

tau2iw_ft_lin
Fourier integration using Filon's method.

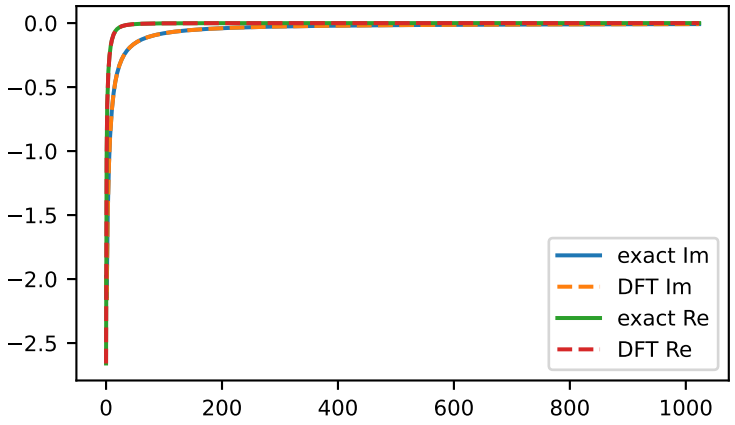
Examples

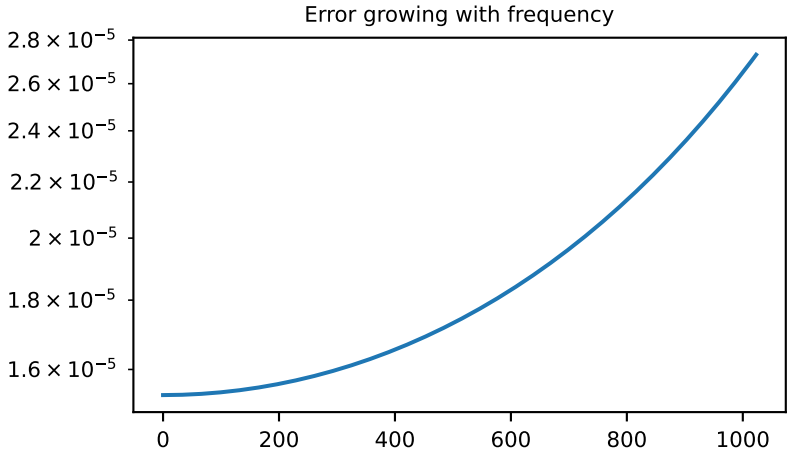
```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> iws = gt.matsubara_frequencies(range((tau.size-1)//2), beta=BETA)
```

```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
>>> # 1/z tail has to be handled manually
>>> gf_dft = gt.fourier.tau2iw_dft(gf_tau + .5, beta=BETA) + 1/iws
>>> gf_tau.size, gf_dft.size
(2049, 1024)
>>> gf_iw = gt.pole_gf_z(iws, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_iw.imag, label='exact Im')
>>> __ = plt.plot(gf_dft.imag, '--', label='DFT Im')
>>> __ = plt.plot(gf_iw.real, label='exact Re')
>>> __ = plt.plot(gf_dft.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.show()
```

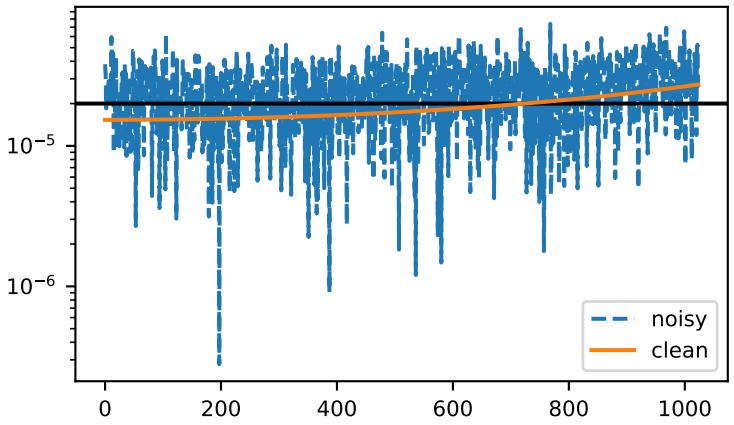
```
>>> __ = plt.title('Error growing with frequency')
>>> __ = plt.plot(abs(gf_iw - gf_dft))
>>> plt.yscale('log')
>>> plt.show()
```


0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
200
400
600
800
1000
exact Im
DFT Im
exact Re
DFT Re


Error growing with frequency
2.8 × 10−5
2.6 × 10−5
2.4 × 10−5
2.2 × 10−5
2 × 10−5
1.8 × 10−5
1.6 × 10−5
0
200
400
600
800
1000

The method is resistant against noise:

```
>>> magnitude = 2e-5
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> gf_dft_noisy = gt.fourier.tau2iw_dft(gf_tau + noise + .5, beta=BETA) + 1/iws
>>> __ = plt.plot(abs(gf_iw - gf_dft_noisy), '--', label='noisy')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(abs(gf_iw - gf_dft), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```



gftool.fourier.tau2iw_ft_lin

gftool.fourier.**tau2iw_ft_lin**(*gf_tau*, *beta*)

Fourier integration of the real Green's function *gf_tau*.

Fourier transformation of a fermionic imaginary-time Green's function to Matsubara domain. We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A() B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iw* is then Hermitian. Filon's method is used to calculated the Fourier integral

$$G^n = 0.5 \int\_ dG() e^{i_n},$$

$G()$ is approximated by a linear spline. A linear approximation was chosen to be able to integrate noisy functions. Information on oscillatory integrations can be found e.g. in [filon1930] and [iserles2006].

Parameters

gf_tau
[(…, N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

(..., (N_iw - 1)/2) float np.ndarray
The Fourier transform of *gf_tau* for positive fermionic Matsubara frequencies i_n.

See also:

tau2iw_dft
Plain implementation using Riemann sum.

References

[filon1930], [iserles2006]

Examples

```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> iws = gt.matsubara_frequencies(range((tau.size-1)//2), beta=BETA)
```

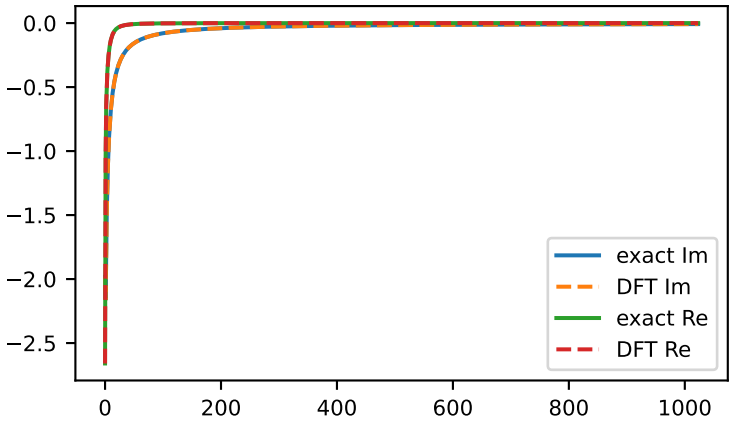
```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
>>> # 1/z tail has to be handled manually
>>> gf_ft_lin = gt.fourier.tau2iw_ft_lin(gf_tau + .5, beta=BETA) + 1/iws
>>> gf_tau.size, gf_ft_lin.size
(2049, 1024)
>>> gf_iw = gt.pole_gf_z(iws, poles=poles, weights=weights)
```

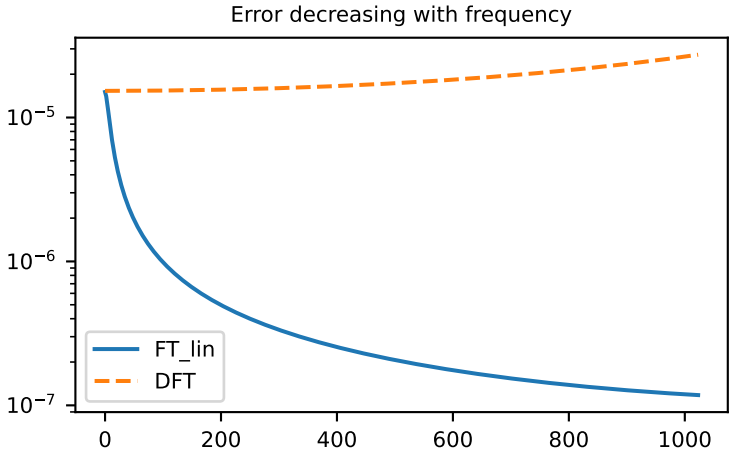
```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_iw.imag, label='exact Im')
>>> __ = plt.plot(gf_ft_lin.imag, '--', label='DFT Im')
>>> __ = plt.plot(gf_iw.real, label='exact Re')
>>> __ = plt.plot(gf_ft_lin.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.show()
```

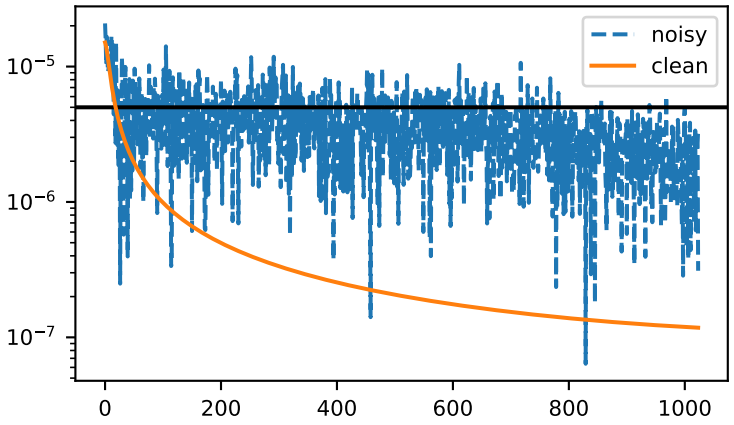
```
>>> __ = plt.title('Error decreasing with frequency')
>>> __ = plt.plot(abs(gf_iw - gf_ft_lin), label='FT_lin')
>>> gf_dft = gt.fourier.tau2iw_dft(gf_tau + .5, beta=BETA) + 1/iws
>>> __ = plt.plot(abs(gf_iw - gf_dft), '--', label='DFT')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

The method is resistant against noise:

```
>>> magnitude = 5e-6
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> gf_ft_noisy = gt.fourier.tau2iw_ft_lin(gf_tau + noise + .5, beta=BETA) + 1/iws
>>> __ = plt.plot(abs(gf_iw - gf_ft_noisy), '--', label='noisy')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(abs(gf_iw - gf_ft_lin), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```


0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
200
400
600
800
1000
exact Im
DFT Im
exact Re
DFT Re


Error decreasing with frequency
10^{-5}
10^{-6}
10^{-7}
0
200
400
600
800
1000
FT_lin
DFT



gftool.fourier.tau2izp

`gftool.fourier.`**`tau2izp`**(*gf_tau*, *beta*, *izp*, *moments=None*, *occ=False*, *weight=None*)

Fourier transform of the real Green's function *gf_tau* to *izp*.

Fourier transformation of a fermionic imaginary-time Green's function to fermionic imaginary Padé frequencies *izp*. We assume a real Green's function *gf_tau*, which is the case for commutator Green's functions $G_{AB}() = \langle A() B\rangle$ with $A = B^\dagger$. The Fourier transform *gf_iw* is then Hermitian. If no explicit *moments* are given, this function removes $-G_{AB}() - G_{AB}(0) = \langle [A, B]\rangle$.

TODO: this function is not vectorized yet.

Parameters

gf_tau
[(N_tau) float np.ndarray] The Green's function at imaginary times $\in [0,]$.

beta
[float] The inverse temperature $beta = 1/k_B T$.

izp
[(N_izp) complex np.ndarray] Complex Padé frequencies at which the Fourier transform is evaluated.

moments
[(m) float array_like, optional] High-frequency moments of *gf_iw*. If none are given, the first moment is chosen to remove the discontinuity at $= 0^{\pm}$.

occ
[float, optional] If given, fix occupation of Green's function to *occ* (default: False).

weight
[(..., N_tau) float np.ndarray, optional] Weight the values of *gf_tau*, can be provided to include uncertainty.

Returns

(N_izp) complex np.ndarray
The Fourier transform of *gf_tau* for given Padé frequencies *izp*.

See also:

tau2iw
Fourier transform to fermionic Matsubara frequencies.

pole_gf_from_tau
Function handling the fitting of *gf_tau*.

Notes

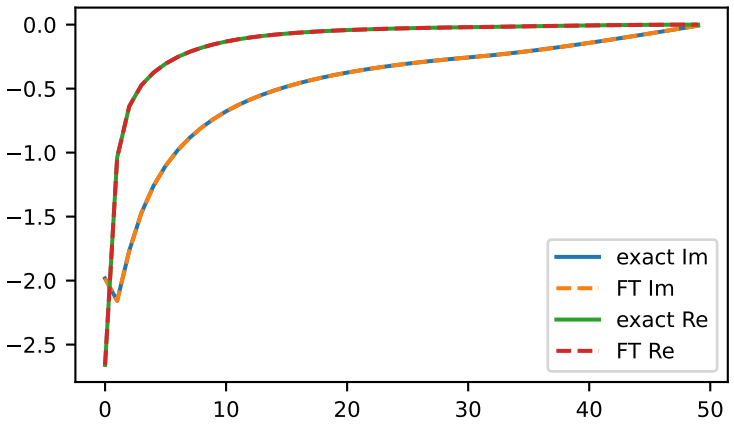
The algorithm performs in fact an analytic continuation instead of a Fourier integral. It is however only evaluated on the imaginary axis, so far the algorithm was observed to be stable

Examples

```
>>> BETA = 50
>>> tau = np.linspace(0, BETA, num=2049, endpoint=True)
>>> izp, __ = gt.pade_frequencies(50, beta=BETA)
```

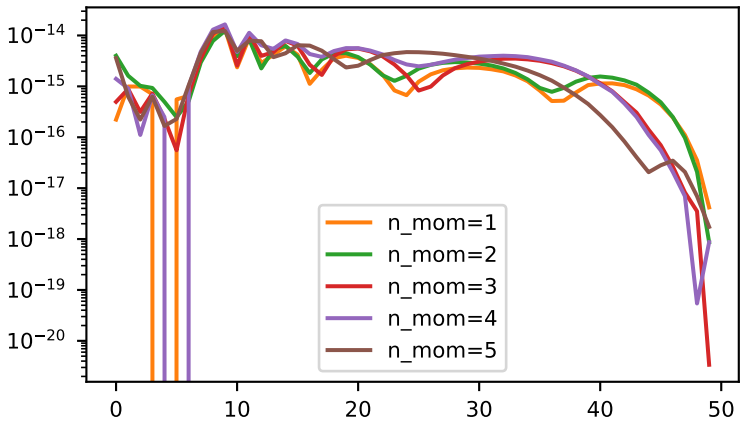
```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_tau = gt.pole_gf_tau(tau, poles=poles, weights=weights, beta=BETA)
>>> gf_ft = gt.fourier.tau2izp(gf_tau, beta=BETA, izp=izp)
>>> gf_izp = gt.pole_gf_z(izp, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(gf_izp.imag, label='exact Im')
>>> __ = plt.plot(gf_ft.imag, '--', label='FT Im')
>>> __ = plt.plot(gf_izp.real, label='exact Re')
>>> __ = plt.plot(gf_ft.real, '--', label='FT Re')
>>> __ = plt.legend()
>>> plt.show()
```



Results of *tau2izp* can be improved giving high-frequency moments.

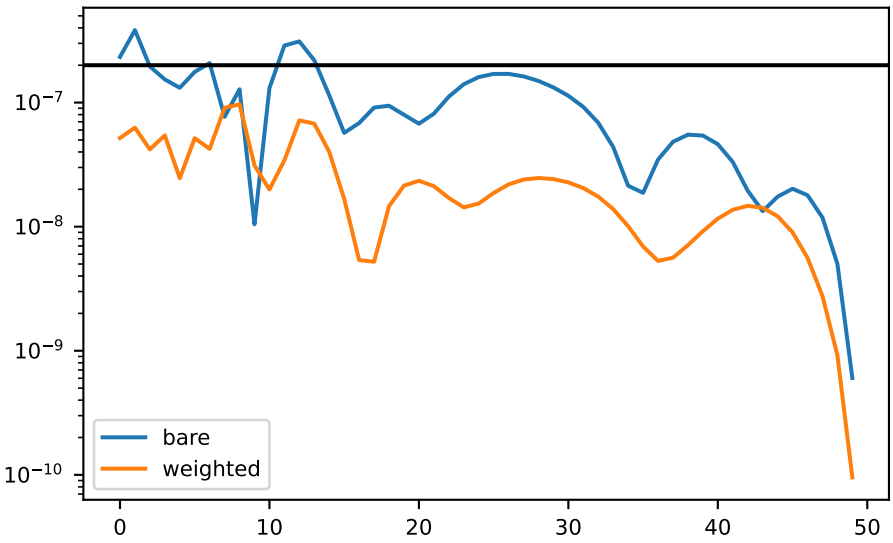
```
>>> mom = np.sum(weights[:, np.newaxis] * poles[:, np.newaxis]**range(6), axis=0)
>>> for n in range(1, 6):
...     gf = gt.fourier.tau2izp(gf_tau, izp=izp, moments=mom[:n], beta=BETA)
...     __ = plt.plot(abs(gf_izp - gf), label=f'n_mom={n}', color=f'C{n}')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```

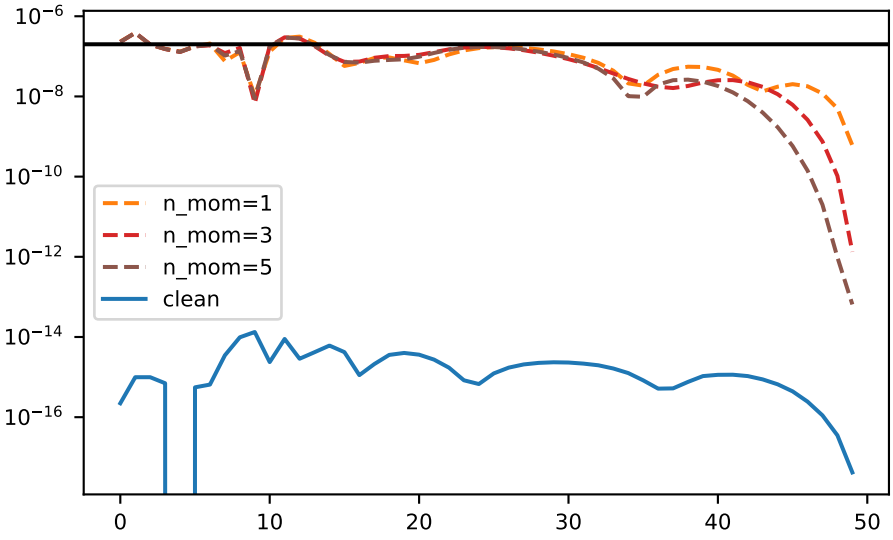


The method is resistant against noise, especially if there is knowledge of the noise:

```
>>> magnitude = 2e-7
>>> noise = np.random.normal(scale=magnitude, size=gf_tau.size)
>>> gf = gt.fourier.tau2izp(gf_tau + noise, izp=izp, moments=(1,), beta=BETA)
>>> __ = plt.plot(abs(gf_izp - gf), label='bare')
>>> gf = gt.fourier.tau2izp(gf_tau + noise, izp=izp, moments=(1,), beta=BETA,
...                         weight=abs(noise)**-0.5)
>>> __ = plt.plot(abs(gf_izp - gf), label='weighted')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```

```
>>> for n in range(1, 7, 2):
...     gf = gt.fourier.tau2izp(gf_tau + noise, izp=izp, moments=mom[:n],␣
↪beta=BETA)
...     __ = plt.plot(abs(gf_izp - gf), '--', label=f'n_mom={n}', color=f'C{n}')
>>> __ = plt.axhline(magnitude, color='black')
>>> __ = plt.plot(abs(gf_izp - gf_ft), label='clean')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```


10^{-7}
10^{-8}
10^{-9}
10^{-10}
bare
weighted
0
10
20
30
40
50


10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
n_mom=1
n_mom=3
n_mom=5
clean
0
10
20
30
40
50

gftool.fourier.tt2z

gftool.fourier.**tt2z**(*tt*, *gf_t*, *z*, *laplace=<function tt2z_lin>*, ***kwds*)

Laplace transform of the real-time Green's function *gf_t*.

Calculate the Laplace transform

$$G(z) = \int dt G(t) \exp(izt)$$

For the Laplace transform to be well defined, it should either be *tt>=0 and z.imag>=0* for the retarded Green's function, or *tt<=0 and z.imag<=0* for the advance Green's function.

The retarded (advanced) Green's function can in principle be evaluated for any frequency point z in the upper (lower) complex half-plane.

The accented contours for *tt* and z depend on the specific used back-end *laplace*.

Parameters

tt
[(Nt) float np.ndarray] The points for which the Green's function *gf_t* is given.

gf_t
[(…, Nt) complex np.ndarray] Green's function at time points *tt*.

z
[(…, Nz) complex np.ndarray] Frequency points for which the Laplace transformed Green's function should be evaluated.

laplace
[{*tt2z_lin*, *tt2z_trapz*, *tt2z_pade*, *tt2z_herm2*}, optional] Back-end to perform the actual Fourier transformation.

****kwds**
Key-word arguments forwarded to *laplace*.

Returns

(…, Nz) complex np.ndarray
Laplace transformed Green's function for complex frequencies z.

Raises

ValueError
If neither the condition for retarded or advanced Green's function is fulfilled.

See also:

tt2z_trapz
Back-end: approximate integral by trapezoidal rule.

tt2z_lin
Back-end: approximate integral by Filon's method.

tt2z_pade
Back-end: use Fourier-Padé algorithm.

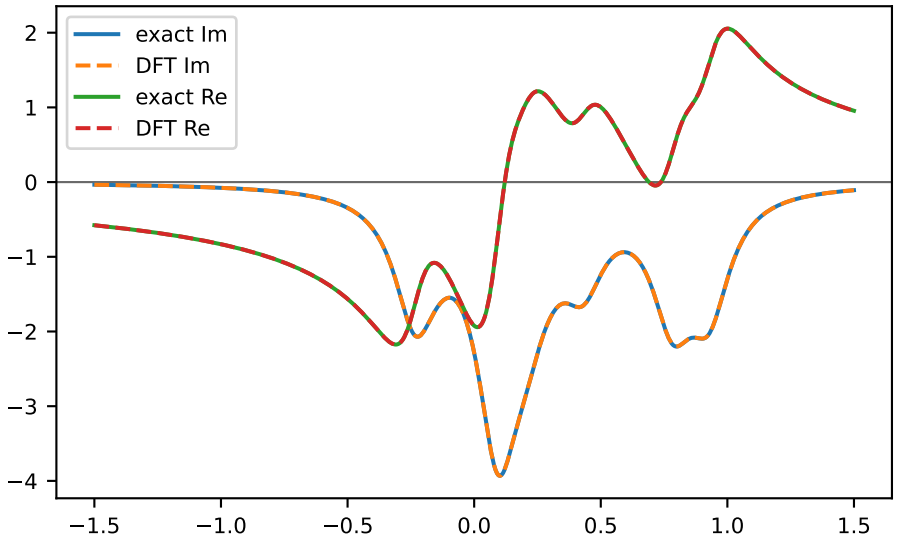
tt2z_herm2
Back-end: using square Hermite-Padé for Fourier.

Examples

```
>>> tt = np.linspace(0, 150, num=1501)
>>> ww = np.linspace(-1.5, 1.5, num=501) + 1e-1j
```

```
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> weights = np.random.random(10)
>>> weights = weights/np.sum(weights)
>>> gf_ret_t = gt.pole_gf_ret_t(tt, poles=poles, weights=weights)
>>> gf_ft = gt.fourier.tt2z(tt, gf_ret_t, z=ww)
>>> gf_ww = gt.pole_gf_z(ww, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.axhline(0, color='dimgray', linewidth=0.8)
>>> __ = plt.plot(ww.real, gf_ww.imag, label='exact Im')
>>> __ = plt.plot(ww.real, gf_ft.imag, '--', label='DFT Im')
>>> __ = plt.plot(ww.real, gf_ww.real, label='exact Re')
>>> __ = plt.plot(ww.real, gf_ft.real, '--', label='DFT Re')
>>> __ = plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



The function Laplace transform can be evaluated at abitrary contours, e.g. for a semi-ceircle in the the upper half-plane. Note, that close to the real axis the accuracy is bad, due to the truncation at *max(tt)*

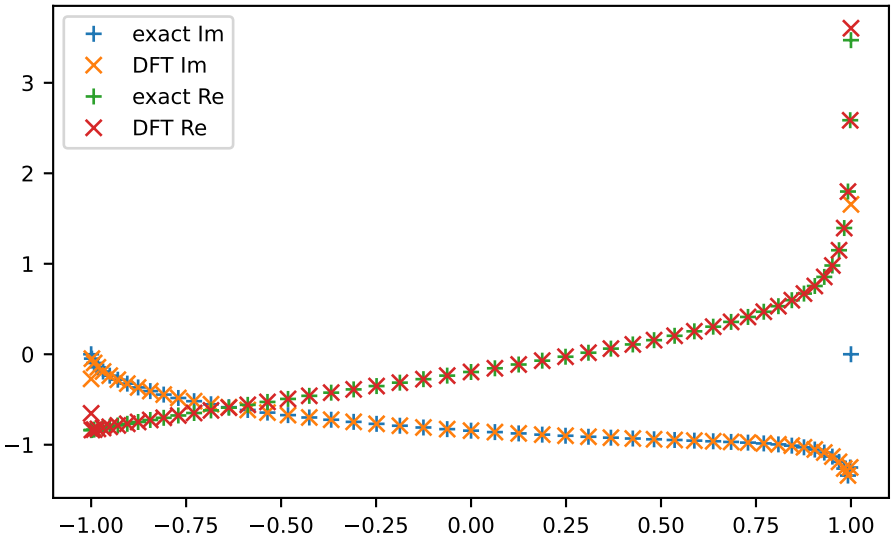
```
>>> z = np.exp(1j*np.pi*np.linspace(0, 1, num=51))
>>> gf_ft = gt.fourier.tt2z(tt, gf_ret_t, z=z)
>>> gf_z = gt.pole_gf_z(z, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(z.real, gf_z.imag, '+', label='exact Im')
>>> __ = plt.plot(z.real, gf_ft.imag, 'x', label='DFT Im')
>>> __ = plt.plot(z.real, gf_z.real, '+', label='exact Re')
>>> __ = plt.plot(z.real, gf_ft.real, 'x', label='DFT Re')
>>> __ = plt.legend()
```

(continues on next page)

(continued from previous page)

```
>>> plt.tight_layout()
>>> plt.show()
```



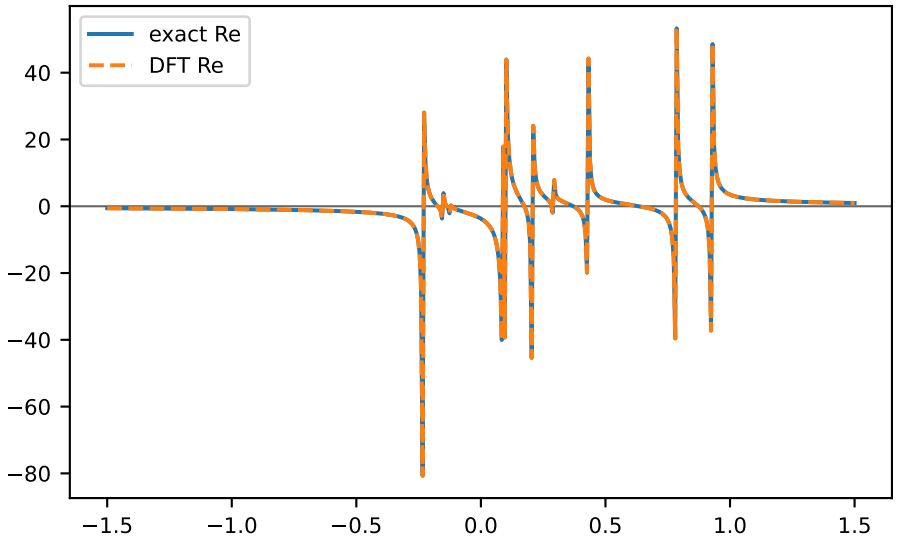
For small *max(tt)* close to the real axis, `tt2z_pade` is often the superior choice (it is taylored to resove poles):

```
>>> tt = np.linspace(0, 40, num=401)
>>> ww = np.linspace(-1.5, 1.5, num=501) + 1e-3j
>>> gf_ret_t = gt.pole_gf_ret_t(tt, poles=poles, weights=weights)
>>> gf_fp = gt.fourier.tt2z(tt, gf_ret_t, z=ww, laplace=gt.fourier.tt2z_pade)
>>> gf_ww = gt.pole_gf_z(ww, poles=poles, weights=weights)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='dimgray', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label='exact Re')
>>> _ = plt.plot(ww.real, gf_fp.real, '--', label='DFT Re')
>>> _ = plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

Accuracy of the different back-ends:

- For small *z.imag* or small *tt[-1]*, `tt2z_pade` performs better than standard transformations `tt2z_trapz` and `tt2z_lin`. It is especially suited to resolve poles. For large *tt.size*, spurious features can appear.
- `tt2z_herm2` further improves on `tt2z_pade` and can resolve square-root branch-cuts. Might be less stable as a wrong branch can be chosen.
- **`tt2z_trapz` vs `tt2z_lin`:**
  - For large *z.imag*, `tt2z_lin` performs better.
  - For intermediate *z.imag*, the quality depends on the relevant *z.real*. For small *z.real*, the error of `tt2z_trapz` is more uniform; for big *z.real*, `tt2z_lin` is a good approximation.
  - For small *z.imag*, the methods are almost identical, the truncation of *tt* dominates the error.




```
>>> tt = np.linspace(0, 150, num=1501)
>>> gf_ret_t = gt.pole_gf_ret_t(tt, poles=poles, weights=weights)
>>> import matplotlib.pyplot as plt
>>> for ii, eta in enumerate([1.0, 0.5, 0.1, 0.03]):
...     ww.imag = eta
...     gf_ww = gt.pole_gf_z(ww, poles=poles, weights=weights)
...     gf_trapz = gt.fourier.tt2z(tt, gf_ret_t, z=ww, laplace=gt.fourier.tt2z_
↪trapz)
...     gf_lin = gt.fourier.tt2z(tt, gf_ret_t, z=ww, laplace=gt.fourier.tt2z_lin)
...     __ = plt.plot(ww.real, abs((gf_ww - gf_trapz)/gf_ww),
...                   label=f"z.imag={eta}", color=f"C{ii}")
...     __ = plt.plot(ww.real, abs((gf_ww - gf_lin)/gf_ww), '--', color=f"C{ii}")
...     __ = plt.legend()
>>> plt.yscale('log')
>>> plt.tight_layout()
>>> plt.show()
```

gftool.fourier.tt2z_herm2

gftool.fourier.**tt2z_herm2**(*tt*, *gf_t*, *z*, *herm2=<bound method Hermite2.from_taylor of <class 'gftool.hermpade.Hermite2'>>*, *quad='trapz'*, ***kwds*)

Square Fourier-Padé transform of the real-time Green's function *gf_t*.

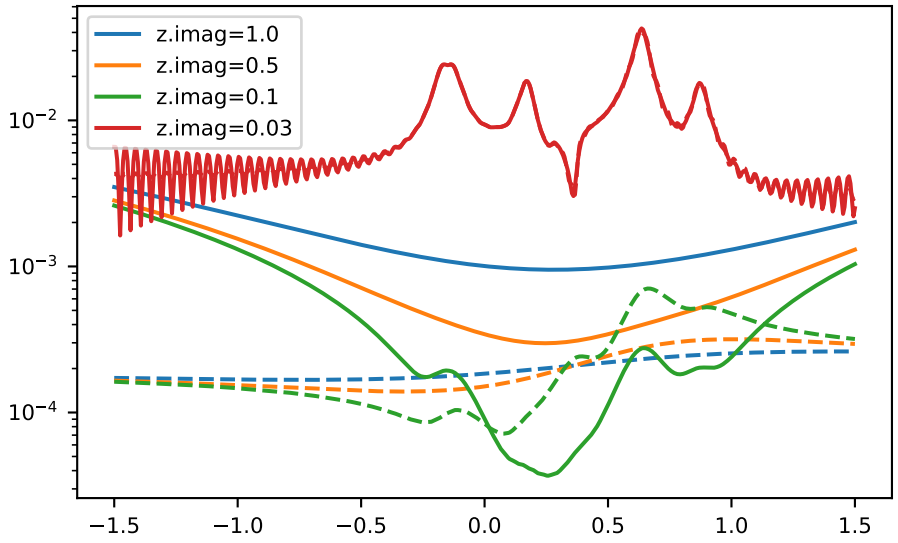
Uses a square Hermite-Padé approximant for the transform. The function requires an equidistant *tt*.

Parameters

tt
: [(Nt) float np.ndarray] The equidistant points for which the Green's function *gf_t* is given.

gf_t
: [(…, Nt) complex np.ndarray] Green's function at time points *tt*.

z
: [(…, Nz) complex np.ndarray] Frequency points for which the Laplace transformed Green's function should be evaluated.



Returns

(…, Nz) complex np.ndarray
Laplace transformed Green's function for complex frequencies *z*.

Other Parameters

herm2
[{gftool.hermpade.Hermite2.from_taylor, gftool.hermpade.Hermite2.from_taylor_lstsq}]
Hermite-Padé algorithm that is used.

quad
[{'trapz', 'simps'}] Quadrature to discretize the Laplace integral.

****kwds**
Optional key-word arguments passed to *herm2*.

Raises

ValueError
If the time points *tt* are not equidistant.

See also:

gftool.hermpade.Hermite2
tt2z_pade
Fourier-Padé using regular rational Padé approximant.

tt2z_trapz
Plain implementation using trapezoidal rule.

tt2z_lin
Laplace integration using Filon's method.

gftool.fourier.tt2z_lin

gftool.fourier.**tt2z_lin**(*tt*, *gf_t*, *z*)

Laplace transform of the real-time Green's function *gf_t*.

Filon's method is used to calculate the Laplace integral

$$G(z) = \int dt G(t) \exp(izt),$$

$G(t)$ is approximated by a linear spline. The function currently requires an equidistant *tt*. Information on oscillatory integrations can be found e.g. in [filon1930] and [iserles2006].

Parameters

tt
[(Nt) float np.ndarray] The equidistant points for which the Green's function *gf_t* is given.

gf_t
[(…, Nt) complex np.ndarray] Green's function at time points *tt*.

z
[(…, Nz) complex np.ndarray] Frequency points for which the Laplace transformed Green's function should be evaluated.

Returns

(…, Nz) complex np.ndarray
Laplace transformed Green's function for complex frequencies *z*.

Raises

ValueError
If the time points *tt* are not equidistant.

See also:

tt2z_trapz
Plain implementation using trapezoidal rule.

Notes

If *numexpr* is available, it is used for the significant speed up it provides for transcendental equations. Internally the sum is evaluated as a matrix product to leverage the speed-up of BLAS.

References

[filon1930], [iserles2006]

gftool.fourier.tt2z_lpz

gftool.fourier.**tt2z_lpz**(*tt*, *gf_t*, *z*, *order=None*, *quad='trapz'*, ***kwds*)

Linear prediction Z-transform of the real-time Green's function *gf_t*.

gftool.fourier.tt2z_pade

gftool.fourier.**tt2z_pade**(*tt*, *gf_t*, *z*, *degree=-1*, *pade=<function pade>*, *quad='trapz'*, ***kwds*)

Fourier-Padé transform of the real-time Green's function *gf_t*.

The function requires an equidistant *tt*.

Parameters

tt
[(Nt) float np.ndarray] The equidistant points for which the Green's function *gf_t* is given.

gf_t
[(…, Nt) complex np.ndarray] Green's function at time points *tt*.

z
[(…, Nz) complex np.ndarray] Frequency points for which the Laplace transformed Green's function should be evaluated.

Returns

(…, Nz) complex np.ndarray
Laplace transformed Green's function for complex frequencies z.

Other Parameters

degree
[int, optional] Asymptotic degree d of the Green's function $G(z) \sim z^d$ for $abs(z) \to \infty$ (default: -1).

pade
[{gftool.hermpade.pade, gftool.hermpade.pader}] Padé algorithm that is used.

quad
[{'trapz', 'simps'}] Quadrature to discretize the Laplace integral.

****kwds**
Optional key-word arguments passed to `pade`.

Raises

ValueError
If the time points *tt* are not equidistant.

See also:

gftool.hermpade.pade
gftool.hermpade.pader
tt2z_herm2
Fourier-Padé using square Hermite-Padé approximant.

tt2z_trapz
Plain implementation using trapezoidal rule.

tt2z_lin
Laplace integration using Filon's method.

gftool.fourier.tt2z_simps

gftool.fourier.**tt2z_simps**(*tt*, *gf_t*, *z*)

Laplace transform of the real-time Green's function *gf_t*.

Approximate the Laplace integral using the Simpson rule.

Parameters

tt
[(Nt) float np.ndarray] The equidistant points for which the Green's function *gf_t* is given.

gf_t
[(…, Nt) complex np.ndarray] Green's function at time points *tt*.

z
[(…, Nz) complex np.ndarray] Frequency points for which the Laplace transformed Green's function should be evaluated.

Returns

(…, Nz) complex np.ndarray
Laplace transformed Green's function for complex frequencies *z*.

See also:

tt2z_trapz
Plain implementation using trapezoidal rule.

tt2z_lin
Laplace integration using Filon's method.

Notes

If *numexpr* is available, it is used for the significant speed up it provides for transcendental equations. Internally the sum is evaluated as a matrix product to leverage the speed-up of BLAS.

gftool.fourier.tt2z_trapz

gftool.fourier.**tt2z_trapz**(*tt*, *gf_t*, *z*)

Laplace transform of the real-time Green's function *gf_t*.

Approximate the Laplace integral by trapezoidal rule:

$$G(z) = \int dt G(t) \exp(izt) \approx \sum_{k=1}^{N} [G(t_{k-1}) \exp(izt_{k-1}) + G(t_k) \exp(izt_k)] t_k/2$$

The function can handle any input discretization *tt*.

Parameters

tt
[(Nt) float np.ndarray] The points for which the Green's function *gf_t* is given.

gf_t
[(…, Nt) complex np.ndarray] Green's function at time points *tt*.

z
[(…, Nz) complex np.ndarray] Frequency points for which the Laplace transformed Green's function should be evaluated.

Returns

(…, Nz) complex np.ndarray
Laplace transformed Green's function for complex frequencies *z*.

See also:

tt2z_lin
Laplace integration using Filon's method.

Notes

The function is equivalent to the one-liner *np.trapz(np.exp(1j*z[:, None]*tt)*gf_t, x=tt)*. If *numexpr* is available, it is used for the significant speed up it provides for transcendental equations. Internally the sum is evaluated as a matrix product to leverage the speed-up of BLAS.

3.1.5 gftool.hermpade

Hermite-Padé approximants from Taylor expansion.

See [fasondini2019] for practical applications and [baker1996] for the extensive theoretical basis.

We present the example from [fasondini2019] showing the approximations. We consider the cubic root `f(z) = (1 + z)**(1/3)`, the radius of convergence of its series is 1.

Taylor series

Obviously the Taylor series fails for *z<=1* as it cannot represent a pole, but also for larger *z>=1* it fails:

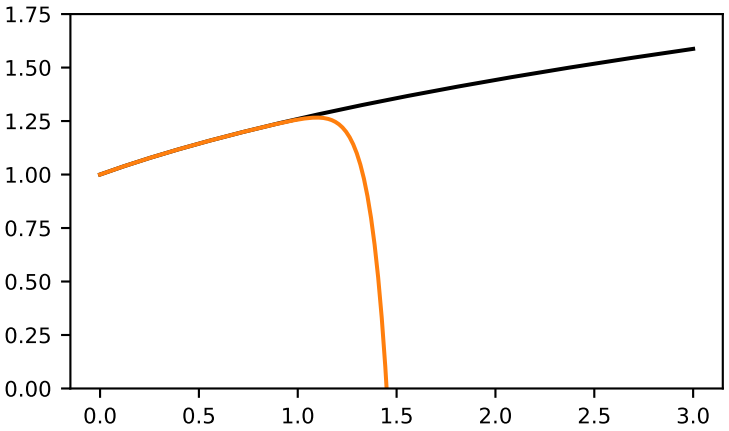
```
>>> from scipy.special import binom
>>> an = binom(1/3, np.arange(17))  # Taylor of (1+x)**(1/3)
>>> def f(z):
...     return np.emath.power(1+z, 1/3)
>>> taylor = np.polynomial.Polynomial(an)
```

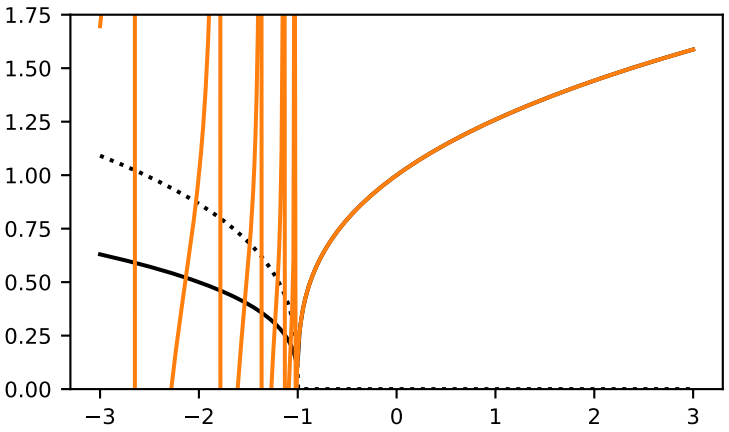
```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 3, num=500)
>>> __ = plt.plot(x, f(x), color='black')
>>> __ = plt.plot(x, taylor(x), color='C1')
>>> __ = plt.ylim(0, 1.75)
>>> plt.show()
```

Padé approximant

The Padé approximant can be used to improve the Taylor expansion and expands the applicability beyond the radius of convergence:

```
>>> x = np.linspace(-3, 3, num=501)
>>> pade = gt.hermpade.pade(an, num_deg=8, den_deg=8)
>>> __ = plt.plot(x, f(x).real, color='black')
>>> __ = plt.plot(x, f(x).imag, ':', color='black')
>>> __ = plt.plot(x, pade.eval(x), color='C1')
>>> __ = plt.ylim(0, 1.75)
>>> plt.show()
```

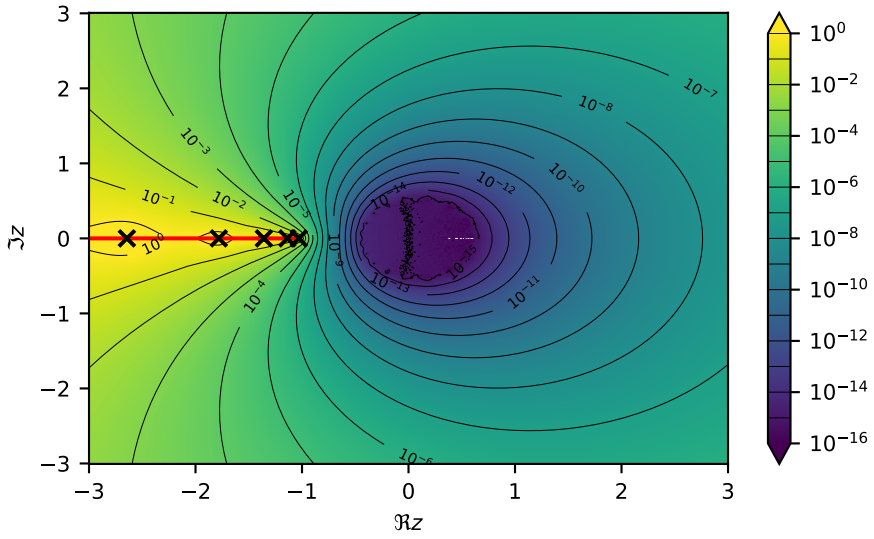

1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0


1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−3
−2
−1
0
1
2
3

The Padé approximant provides a global approximation. For negative values, however, the Padé approximant still fails, as it cannot accurately represent a branch cut. The Padé approximant is suitable for simple poles and tries to approximate the branch-cut by a finite number of poles. It is instructive to plot the error in the complex plane:

```
>>> y = np.linspace(-3, 3, num=501)
>>> z = x[:, None] + 1j*y[None, :]
>>> error = abs(pade.eval(z) - f(z))
>>> poles = pade.denom.roots()
```

```
>>> import matplotlib as mpl
>>> fmt = mpl.ticker.LogFormatterMathtext()
>>> __ = fmt.create_dummy_axis()
>>> norm = mpl.colors.LogNorm(vmin=1e-16, vmax=1)
>>> __ = plt.pcolormesh(x, y, error.T, shading='nearest', norm=norm)
>>> cbar = plt.colorbar(extend='both')
>>> levels = np.logspace(-15, 0, 16)
>>> cont = plt.contour(x, y, error.T, colors='black', linewidths=0.25, levels=levels)
>>> __ = plt.clabel(cont, cont.levels, fmt=fmt, fontsize='x-small')
>>> for ll in levels:
...     __ = cbar.ax.axhline(ll, color='black', linewidth=0.25)
>>> __ = plt.hlines(0, xmin=x[0], xmax=-1, color='red')  # branch cut
>>> __ = plt.scatter(poles.real, poles.imag, color='black', marker='x', zorder=2)  #␣
↪poles
>>> __ = plt.xlabel(r"$\Re z$")
>>> __ = plt.ylabel(r"$\Im z$")
>>> __ = plt.xlim(x[0], x[-1])
>>> plt.tight_layout()
>>> plt.gca().set_rasterization_zorder(1.5)  # avoid excessive files
>>> plt.show()
```



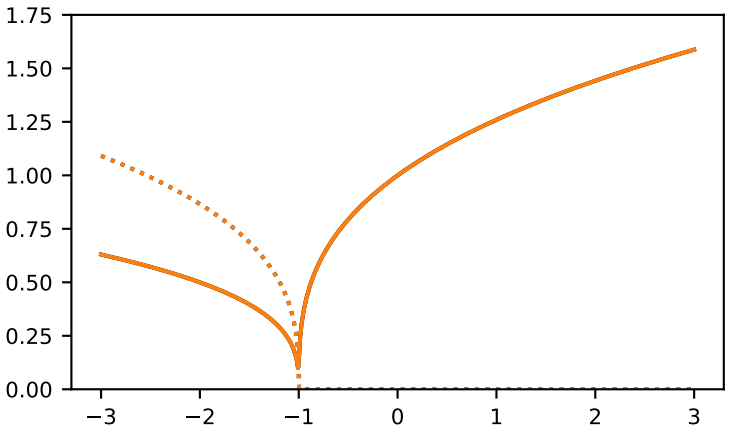
Away from the branch-cut, indicated by the red line, the Padé approximant is a reasonable approximation. The crosses indicate the simple poles of the Padé approximant.

Quadratic Hermite-Padé approximant

A further improvement is provided by the quadratic Hermite-Padé approximant, which can represent square-root branch cuts:

```
>>> herm2 = gt.hermpade.Hermite2.from_taylor(an, deg_p=5, deg_q=5, deg_r=5)
```

```
>>> __ = plt.plot(x, f(x).real, color='black')
>>> __ = plt.plot(x, f(x).imag, ':', color='black')
>>> __ = plt.plot(x, herm2.eval(x + 1e-16j).real, color='C1')
>>> __ = plt.plot(x, herm2.eval(x + 1e-16j).imag, ':', color='C1')
>>> __ = plt.ylim(0, 1.75)
>>> plt.show()
```



It nicely approximates the function almost everywhere. However, there is ambiguity which branch to choose, thus we had to add the shift `1e-16j` by hand, to get the correct branch on the real axis. Let's compare the error to the (linear) Padé approximant:

```
>>> __ = plt.plot(x, abs(pade.eval(x) - f(x)), label="Padé")
>>> __ = plt.plot(x, abs(herm2.eval(x + 1e-16j) - f(x)), label="Herm2")
>>> __ = plt.plot(x, abs(herm2.eval(x) - f(x)), label="wrong branch")
>>> __ = plt.yscale('log')
>>> __ = plt.legend()
>>> plt.show()
```

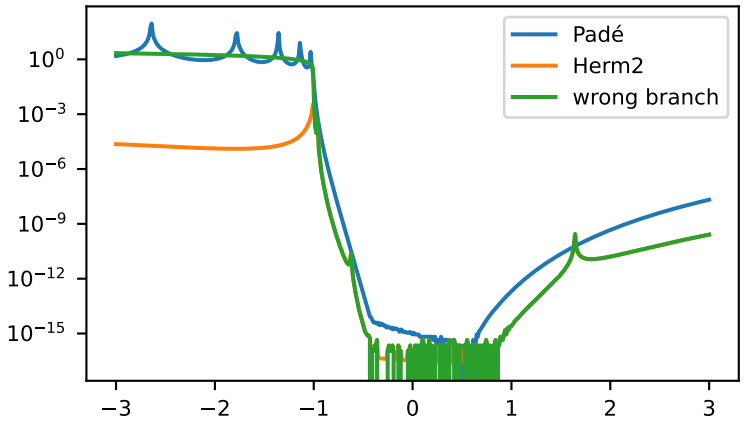
The correct branch nicely approximates the function everywhere, but even the wrong branch performs better than Padé.

Let's also compare the quality of the approximants in the complex plane:

```
>>> error2 = np.abs(herm2.eval(z) - f(z))
```

```
>>> __, axes = plt.subplots(ncols=2, sharex=True, sharey=True)
>>> __ = axes[0].set_title("Padé")
>>> __ = axes[1].set_title("Herm2")
>>> levels = np.logspace(-15, 0, 16)
>>> for ax, err in zip(axes, [error, error2]):
```

(continues on next page)



(continued from previous page)

```
...     pcm = ax.pcolormesh(x, y, err.T, shading='nearest', norm=norm)
...     cont = ax.contour(x, y, err.T, colors='black', linewidths=0.25, levels=levels)
...     __ = ax.clabel(cont, cont.levels, fmt=fmt, fontsize='x-small')
...     __ = ax.set_xlabel(r"$\Re z$")
...     __ = ax.hlines(0, xmin=x[0], xmax=-1, color='red')  # branch cut
...     ax.set_rasterization_zorder(1.5)
>>> __ = axes[0].scatter(poles.real, poles.imag, color='black', marker='x', zorder=2)␣
↪ # poles
>>> __ = plt.xlim(x[0], x[-1])
>>> __ = axes[0].set_ylabel(r"$\Im z$")
>>> plt.tight_layout()
>>> cbar = plt.colorbar(pcm, extend='both', ax=axes, fraction=0.08, pad=0.02)
>>> cbar.ax.tick_params(labelsize='x-small')
>>> for ll in levels:
...     __ = cbar.ax.axhline(ll, color='black', linewidth=0.25)
>>> plt.show()
```

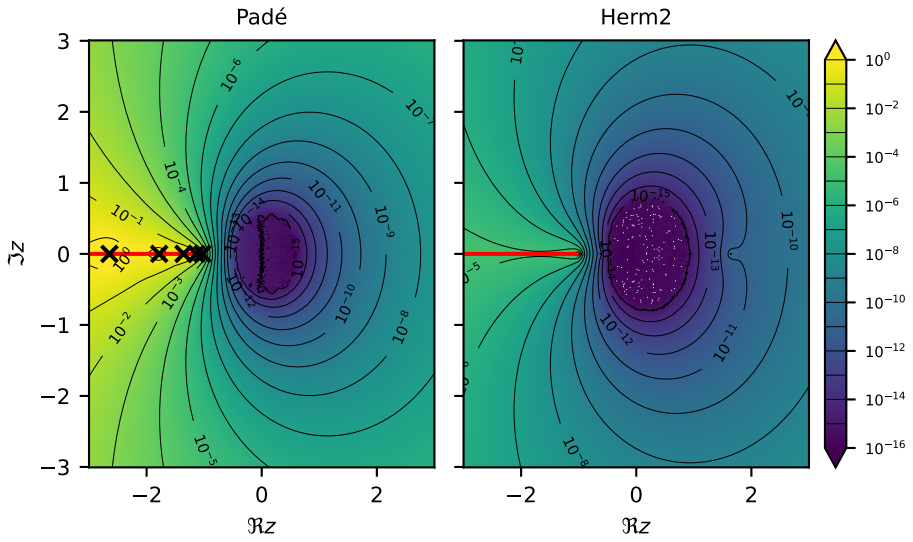
Note, however, the quadratic Hermite-Padé approximant contains the ambiguity which branch to choose. The heuristic can fail and should therefore be checked.

Alternative example: logarithmic branch cut

We also show the results for a logarithmic branch cut, showing that the results hold not only for algebraic branch cuts. Let's consider the approximations for the logarithm `f(z) = np.log(1 + z)`, whose series has a radius of convergence of *1*:

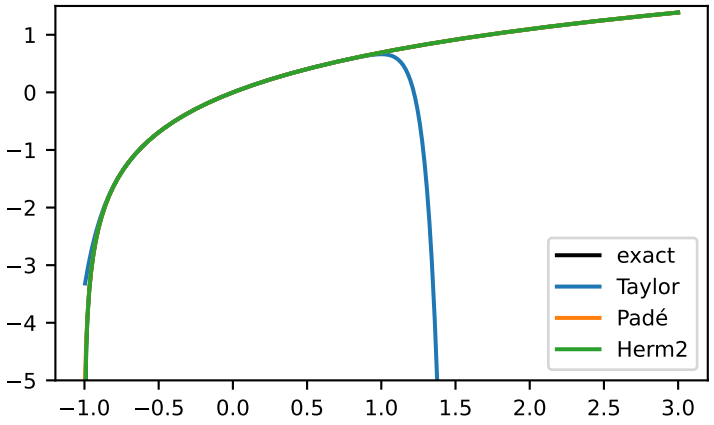
```
>>> an = np.r_[0, (-1)**np.arange(16)/np.arange(1, 17)]  # Taylor of ln(1+x)
>>> def f(z):
...     return np.emath.log(1 + z)
>>> taylor = np.polynomial.Polynomial(an)
>>> pade = gt.hermpade.pade(an, num_deg=8, den_deg=8)
>>> herm2 = gt.hermpade.Hermite2.from_taylor(an, deg_p=5, deg_q=5, deg_r=5)
```

Again, we see that the Taylor series fails for larger $z>=1$, the (linear) Padé and the quadratic Hermite-Padé, on the other



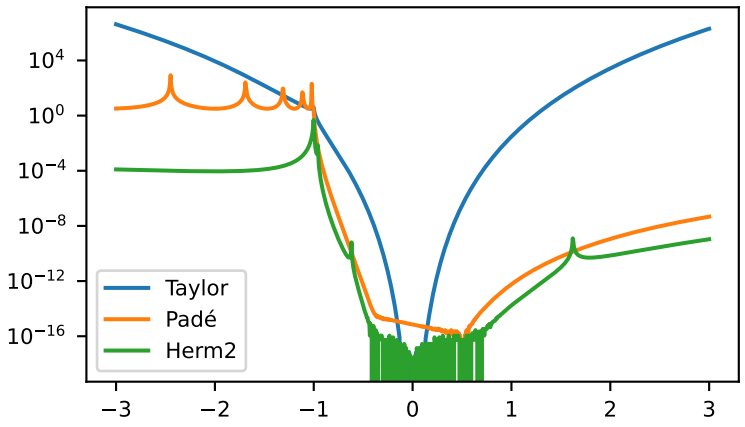
hand, yield good results also for large values.

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 3, num=1001)[1:]
>>> __ = plt.plot(x, f(x), color='black', label="exact")
>>> __ = plt.plot(x, taylor(x), label="Taylor")
>>> __ = plt.plot(x, pade.eval(x), label="Padé")
>>> __ = plt.plot(x, herm2.eval(x), label="Herm2")
>>> __ = plt.ylim(-5, 1.5)
>>> __ = plt.legend()
>>> plt.show()
```



Plotting the error, again we see that the Taylor series is only valied for small values of z, and Padé fails to approximate the branch cut well. The quadratic Hermite-Padé approximant is best for (almost) all values.

```
>>> x = np.linspace(-3, 3, num=1001)
>>> __ = plt.plot(x, abs(taylor(x) - f(x)), label="Taylor")
>>> __ = plt.plot(x, abs(pade.eval(x) - f(x)), label="Padé")
>>> __ = plt.plot(x, abs(herm2.eval(x) - f(x)), label="Herm2")
>>> __ = plt.yscale('log')
>>> __ = plt.legend()
>>> plt.show()
```



Plotting the error in the complex plain shows that Padé fails to resolve the branch cut but is else a good approximation globally. The branch-cut is indicated by the red line, the crosses mark the poles of Padé. The Hermite-Padé algorithm yields good results also in the vicinity of the branch cut.

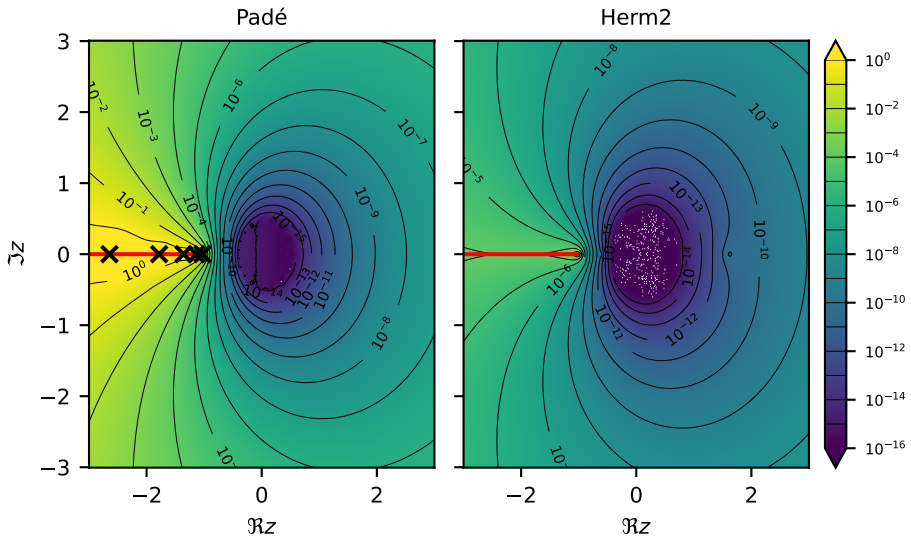
```
>>> x = np.linspace(-3, 3, num=501)
>>> y = np.linspace(-3, 3, num=501)
>>> z = x[:, None] + 1j*y[None, :]
>>> error = np.abs(pade.eval(z) - f(z))
>>> error2 = np.abs(herm2.eval(z) - f(z))
```

```
>>> import matplotlib as mpl
>>> __, axes = plt.subplots(ncols=2, sharex=True, sharey=True)
>>> __ = axes[0].set_title("Padé")
>>> __ = axes[1].set_title("Herm2")
>>> norm = mpl.colors.LogNorm(vmin=1e-16, vmax=1)
>>> for ax, err in zip(axes, [error, error2]):
...     pcm = ax.pcolormesh(x, y, err.T, shading='nearest', norm=norm)
...     cont = ax.contour(x, y, err.T, colors='black', linewidths=0.25, levels=levels)
...     __ = ax.clabel(cont, cont.levels, fmt=fmt, fontsize='x-small')
...     __ = ax.set_xlabel(r"$\Re z$")
...     __ = ax.hlines(0, xmin=x[0], xmax=-1, color='red')  # branch cut
...     ax.set_rasterization_zorder(1.5)
>>> __ = axes[0].scatter(poles.real, poles.imag, color='black', marker='x', zorder=2)␣
↪ # poles
>>> __ = plt.xlim(x[0], x[-1])
>>> __ = axes[0].set_ylabel(r"$\Im z$")
>>> plt.tight_layout()
>>> cbar = plt.colorbar(pcm, extend='both', ax=axes, fraction=0.08, pad=0.02)
```

(continues on next page)

(continued from previous page)

```
>>> cbar.ax.tick_params(labelsize='x-small')
>>> for ll in levels:
...     __ = cbar.ax.axhline(ll, color='black', linewidth=0.25)
>>> plt.show()
```



References

API

Functions

hermite2(an, p_deg, q_deg, r_deg)	Return the polynomials p, q, r for the quadratic Hermite-Padé approximant.
hermite2_lstsq(an, p_deg, q_deg, r_deg[, ...])	Return the polynomials p, q, r for the quadratic Hermite-Padé approximant.
pade(an, num_deg, den_deg[, fast])	Return the [*num_deg*/*den_deg*] Padé approximant to the polynomial *an*.
pade_lstsq(an, num_deg, den_deg[, rcond, fix_q])	Return the [*num_deg*/*den_deg*] Padé approximant to the polynomial *an*.
pader(an, num_deg, den_deg[, rcond])	Robust version of Padé approximant to polynomial *an*.

gftool.hermpade.hermite2

gftool.hermpade.**hermite2**(*an*, *p_deg: int*, *q_deg: int*, *r_deg: int*) → Tuple[Polynomial, Polynomial, Polynomial]

Return the polynomials *p*, *q*, *r* for the quadratic Hermite-Padé approximant.

The polynomials fulfill the equation

$$p(x) + q(x)f(x) + r(x)f^2(x) = (x^{N_p+N_q+N_r+2})$$

where $f(x)$ is the function with Taylor coefficients *an*, and N_x are the degrees of the polynomials. The approximant has two branches

$$F^{\pm}(z) = [-q(z) \pm \sqrt{q^2(z) - 4p(z)r(z)}]/2r(z)$$

Parameters

an
[(L,) array_like] Taylor series coefficients representing polynomial of order `L-1`.

p_deg, q_deg, r_deg
[int] The order of the polynomials of the quadratic Hermite-Padé approximant. The sum must be at most `p_deg + q_deg + r_deg + 2 <= L`.

Returns

p, q, r
[Polynom] The polynomials *p*, *q*, and *r* building the quadratic Hermite-Padé approximant.

See also:

Hermite2
High-level interface, guessing the correct branch.

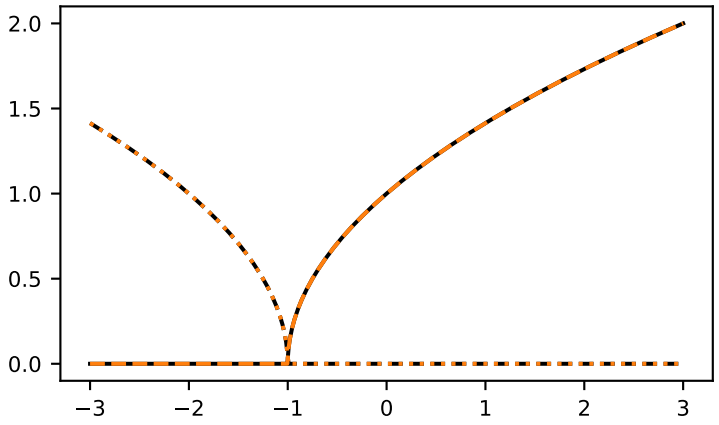
Examples

The quadratic Hermite-Padé approximant can reproduce the square root `f(z) = (1 + z)**(1/2)`:

```
>>> from scipy.special import binom
>>> an = binom(1/2, np.arange(5+5+5+2))  # Taylor of (1+x)**(1/2)
>>> x = np.linspace(-3, 3, num=500)
>>> fx = np.emath.power(1+x, 1/2)
```

```
>>> p, q, r = gt.hermpade.hermite2(an, 5, 5, 5)
>>> px, qx, rx = p(x), q(x), r(x)
>>> pos_branch = (-qx + np.emath.sqrt(qx**2 - 4*px*rx)) / (2*rx)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(x, fx.real, label='exact', color='black')
>>> __ = plt.plot(x, fx.imag, '--', color='black')
>>> __ = plt.plot(x, pos_branch.real, '--', label='Herm2', color='C1')
>>> __ = plt.plot(x, pos_branch.imag, ':', color='C1')
>>> plt.show()
```



gftool.hermpade.hermite2_lstsq

gftool.hermpade.**hermite2_lstsq**(*an*, *p_deg: int*, *q_deg: int*, *r_deg: int*, *rcond=None*, *fix_qr=None*) → Tuple[Polynomial, Polynomial, Polynomial]

Return the polynomials *p*, *q*, *r* for the quadratic Hermite-Padé approximant.

Same as *hermite2*, however all elements of *an* are taken into account. Instead of finding the null-vector of the underdetermined system, the parameter `q.coeff[0]=1` is fixed and the system is solved truncating small singular values.

The polynomials fulfill the equation

$$p(x) + q(x)f(x) + r(x)f^2(x) = (x^{N_p+N_q+N_r+2})$$

where $f(x)$ is the function with Taylor coefficients *an*, and N_x are the degrees of the polynomials. The approximant has two branches

$$F^{\pm}(z) = [-q(z) \pm \sqrt{q^2(z) - 4p(z)r(z)}]/2r(z)$$

Parameters

an
: [(L,) array_like] Taylor series coefficients representing polynomial of order `L-1`.

p_deg, q_deg, r_deg
: [int] The order of the polynomials of the quadratic Hermite-Padé approximant. The sum must be at most `p_deg + q_deg + r_deg + 2 <= L`.

rcond
: [float, optional] Cut-off ratio for small singular values. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value (default: machine precision times maximum of matrix dimensions).

fix_qr
: [int, optional] The coefficient which is fixed to 1. The values `0 <= fix_qr <= q_deg` corresponds to the coefficients of the polynomial *q*, the values `q_deg + 1 <= fix_qr <= q_deg + r_deg + 1` correspond to the coefficients of the polynomial *r*.

Returns

p, q, r
[Polynom] The polynomials *p*, *q*, and *r* building the quadratic Hermite-Padé approximant.

See also:

hermite2
Hermite2
High-level interface, guessing the correct branch.

numpy.linalg.lstsq

gftool.hermpade.pade

`gftool.hermpade.`**`pade`**(*an*, *num_deg: int*, *den_deg: int*, *fast=False*) → *RatPol*

Return the [*num_deg*/*den_deg*] Padé approximant to the polynomial *an*.

Parameters

an
[(L,) array_like] Taylor series coefficients representing polynomial of order `L-1`.

num_deg, den_deg
[int] The order of the return approximating numerator/denominator polynomial. The sum must be at most `L`: `L >= num_deg + den_deg + 1`.

fast
[bool, optional] If *fast*, use faster `solve_toeplitz` algorithm. Else use QR and calculate null-vector (default: False).

Returns

RatPol
The rational polynomial with numerator `RatPol.numer`, and denominator `RatPol.denom`.

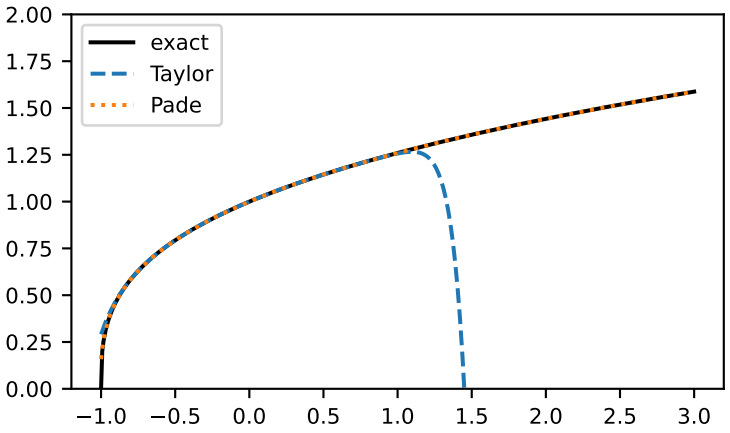
Examples

Let's approximate the cubic root `f(z) = (1 + z)**(1/3)` by the `[8/8]` Padé approximant:

```
>>> from scipy.special import binom
>>> an = binom(1/3, np.arange(8+8+1))  # Taylor of (1+x)**(1/3)
>>> x = np.linspace(-1, 3, num=500)
>>> fx = np.emath.power(1+x, 1/3)
```

```
>>> pade = gt.hermpade.pade(an, num_deg=8, den_deg=8)
```

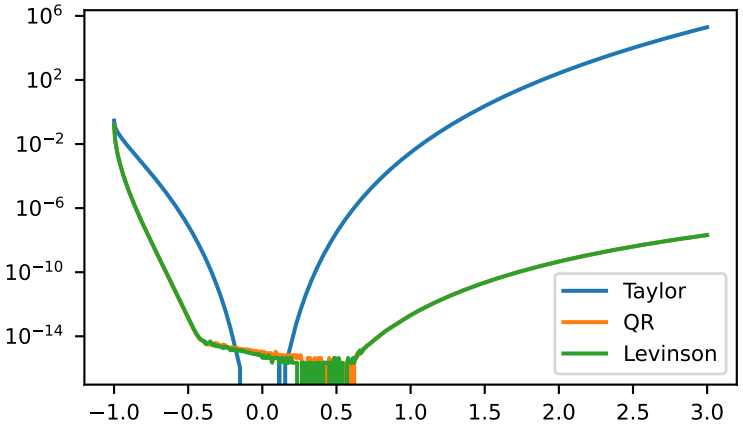
```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(x, fx, label='exact', color='black')
>>> __ = plt.plot(x, np.polynomial.Polynomial(an)(x), '--', label='Taylor')
>>> __ = plt.plot(x, pade.eval(x), ':', label='Pade')
>>> __ = plt.ylim(ymin=0, ymax=2)
>>> __ = plt.legend(loc='upper left')
>>> plt.show()
```



The Padé approximation is able to approximate the function even for larger `x`.

Using `fast=True`, the Toeplitz structure is used to evaluate the *pade* faster using Levinson recursion. This might, however, be less accurate in some cases.

```
>>> padef = gt.hermpade.pade(an, num_deg=8, den_deg=8, fast=True)
>>> __ = plt.plot(x, abs(np.polynomial.Polynomial(an)(x) - fx), label='Taylor')
>>> __ = plt.plot(x, abs(pade.eval(x) - fx), label='QR')
>>> __ = plt.plot(x, abs(padef.eval(x) - fx), label='Levinson')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```



gftool.hermpade.pade_lstsq

gftool.hermpade.**pade_lstsq**(*an*, *num_deg: int*, *den_deg: int*, *rcond=None*, *fix_q=None*) → *RatPol*

Return the [*num_deg*/*den_deg*] Padé approximant to the polynomial *an*.

Same as `pade`, however all elements of *an* are taken into account. Instead of finding the null-vector of the underdetermined system, the parameter `RatPol.denom.coeff[0]=1` is fixed and the system is solved truncating small singular values.

Parameters

an
[(L,) array_like] Taylor series coefficients representing polynomial of order `L-1`.

num_deg, den_deg
[int] The order of the return approximating numerator/denominator polynomial. The sum must be at most `L`: `L >= num_deg + den_deg + 1`.

rcond
[float, optional] Cut-off ratio for small singular values for the denominator polynomial. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value (default: machine precision times *den_deg*).

Returns

RatPol
The rational polynomial with numerator `RatPol.numer`, and denominator `RatPol.denom`.

See also:

pade
numpy.linalg.lstsq

gftool.hermpade.pader

gftool.hermpade.**pader**(*an*, *num_deg: int*, *den_deg: int*, *rcond: float = 1e-14*) → *RatPol*

Robust version of Padé approximant to polynomial *an*.

Implements more or less [gonnet2013]. The degrees *num_deg* and *den_deg* are automatically reduced to obtain a robust solution.

Parameters

an
[(L,) array_like] Taylor series coefficients representing polynomial of order `L-1`.

num_deg, den_deg
[int] The order of the return approximating numerator/denominator polynomial. The sum must be at most `L`: `L >= n + m + 1`. Depending on *rcond* the degrees can be reduced.

rcond
[float, optional] Cut-off ratio for small singular values. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value (default: 1e-14). The default is appropriate for round error due to machine precision.

Returns

RatPol
The rational polynomial with numerator `RatPol.numer`, and denominator `RatPol.denom`.

See also:

pade

References

[gonnet2013]

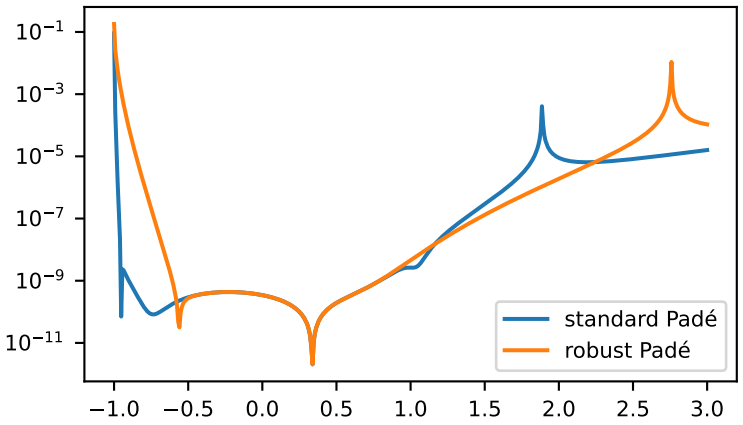
Examples

The robust version can avoid over fitting for high-order Padé approximants. Choosing an appropriate *rcond*, is however a delicate task in practice. We consider an example with random noise on the Taylor coefficients *an*:

```
>>> from scipy.special import binom
>>> deg = 50
>>> an = binom(1/3, np.arange(2*deg + 1))  # Taylor of (1+x)**(1/3)
>>> an += np.random.default_rng().normal(scale=1e-9, size=2*deg + 1)
>>> x = np.linspace(-1, 3, num=500)
>>> fx = np.emath.power(1+x, 1/3)
```

```
>>> pade = gt.hermpade.pade(an, num_deg=deg, den_deg=deg)
>>> pader = gt.hermpade.pader(an, num_deg=deg, den_deg=deg, rcond=1e-8)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(x, abs(pade.eval(x) - fx), label='standard Padé')
>>> __ = plt.plot(x, abs(pader.eval(x) - fx), label='robust Padé')
>>> plt.yscale('log')
>>> __ = plt.legend()
>>> plt.show()
```



Classes

Hermite2(p, q, r, pade)	Quadratic Hermite-Padé approximant with branch selection according to Padé.

gftool.hermpade.Hermite2

class gftool.hermpade.**Hermite2**(*p: Polynomial*, *q: Polynomial*, *r: Polynomial*, *pade:* RatPol)

Quadratic Hermite-Padé approximant with branch selection according to Padé.

A function $f(z)$ with known Taylor coefficients *an* is approximated using

$$p(z) + q(z)f(z) + r(z)f^2(z) = (z^{N_p+N_q+N_r+2})$$

where $f(z)$ is the function with Taylor coefficients *an*, and N_x are the degrees of the polynomials. The approximant has two branches

$$F^{\pm}(z) = [-q(z) \pm \sqrt{q^2(z) - 4p(z)r(z)}]/2r(z)$$

The function *Hermite2.eval* chooses the branch which is locally closer to the Padé approximant, as proposed by [fasondini2019].

Parameters

p, q, r
[Polynom] The polynomials.

pade
[RatPol] The Padé approximant.

References

[fasondini2019]

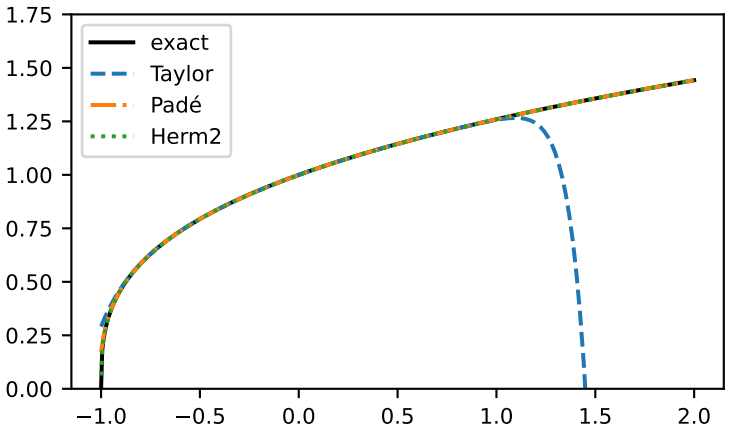
Examples

Let's approximate the cubic root `f(z) = (1 + z)**(1/3)` by the `[5/5/5]` quadratic Hermite-Padé approximant:

```
>>> from scipy.special import binom
>>> an = binom(1/3, np.arange(5+5+5+2))  # Taylor of (1+x)**(1/3)
>>> x = np.linspace(-1, 2, num=500)
>>> fx = np.emath.power(1+x, 1/3)
```

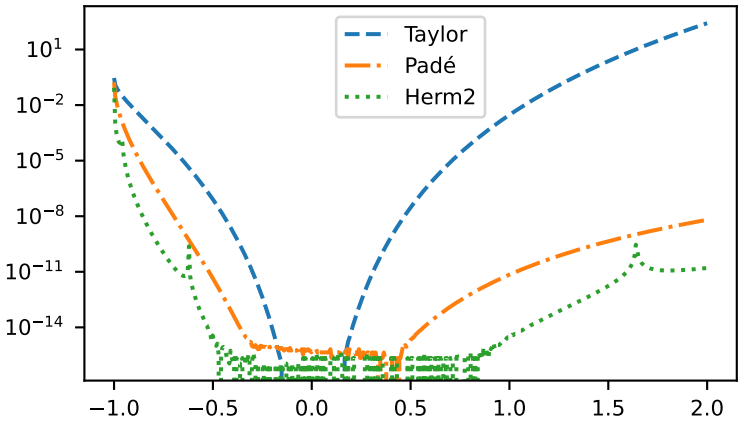
```
>>> herm = gt.hermpade.Hermite2.from_taylor(an, 5, 5, 5)
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(x, fx, label='exact', color='black')
>>> __ = plt.plot(x, np.polynomial.Polynomial(an)(x), '--', label='Taylor')
>>> __ = plt.plot(x, herm.pade.eval(x), '-.', label='Padé')
>>> __ = plt.plot(x, herm.eval(x).real, ':', label='Herm2')
>>> __ = plt.ylim(ymin=0, ymax=1.75)
>>> __ = plt.legend(loc='upper left')
>>> plt.show()
```



The improvement becomes more clear showing the error:

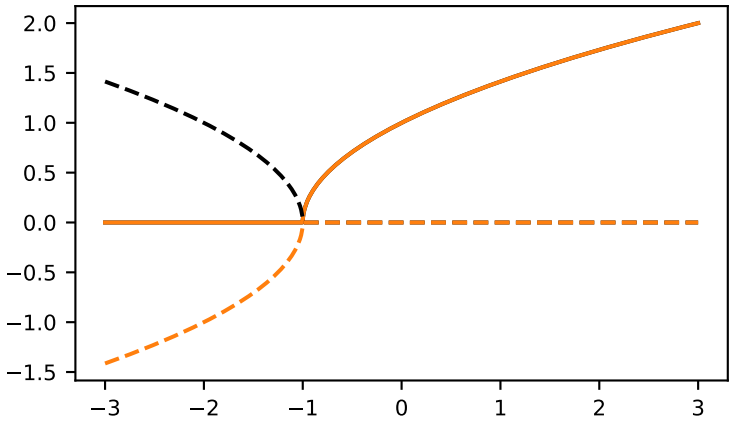
```
>>> __ = plt.plot(x, abs(np.polynomial.Polynomial(an)(x) - fx), '--', label=
↪'Taylor')
>>> __ = plt.plot(x, abs(herm.pade.eval(x) - fx), '-.', label='Padé')
>>> __ = plt.plot(x, abs(herm.eval(x) - fx), ':', label='Herm2')
>>> __ = plt.legend()
>>> plt.yscale('log')
>>> plt.show()
```



Mind, that the prediction of the correct branch is far from safe:

```
>>> an = binom(1/2, np.arange(8+8+1))  # Taylor of (1+x)**(1/2)
>>> x = np.linspace(-3, 3, num=500)
>>> fx = np.emath.power(1+x, 1/2)
>>> herm = gt.hermpade.Hermite2.from_taylor(an, 5, 5, 5)
```

```
>>> __ = plt.plot(x, fx.real, label='exact', color='black')
>>> __ = plt.plot(x, herm.eval(x).real, label='Square', color='C1')
>>> __ = plt.plot(x, fx.imag, '--', color='black')
>>> __ = plt.plot(x, herm.eval(x).imag, '--', color='C1')
>>> plt.show()
```



The positive branch, however, yields the exact result:

```
>>> p_branch, __ = herm.eval_branches(x)
>>> np.allclose(p_branch, fx, rtol=1e-14, atol=1e-14)
True
```

__init__ (*p: Polynomial*, *q: Polynomial*, *r: Polynomial*, *pade:* RatPol) → None

Methods

__init__(p, q, r, pade)	
eval(z)	Evaluate square approximant choosing branch based on Padé.
eval_branches(z)	Evaluate the two branches.
from_taylor(an, deg_p, deg_q, deg_r)	Construct quadratic Hermite-Padé from Taylor expansion *an*.
from_taylor_lstsq(an, deg_p, deg_q, deg_r[, ...])	Construct quadratic Hermite-Padé from Taylor expansion *an*.

gftool.hermpade.Hermite2.__init__

Hermite2.**__init__**(*p: Polynomial*, *q: Polynomial*, *r: Polynomial*, *pade:* RatPol) → None

gftool.hermpade.Hermite2.eval

Hermite2.**eval**(*z*)

Evaluate square approximant choosing branch based on Padé.

gftool.hermpade.Hermite2.eval_branches

Hermite2.**eval_branches**(*z*) → Tuple[ndarray, ndarray]

Evaluate the two branches.

gftool.hermpade.Hermite2.from_taylor

classmethod Hermite2.**from_taylor**(*an*, *deg_p: int*, *deg_q: int*, *deg_r: int*) → *Hermite2*

Construct quadratic Hermite-Padé from Taylor expansion *an*.

gftool.hermpade.Hermite2.from_taylor_lstsq

classmethod Hermite2.**from_taylor_lstsq**(*an*, *deg_p: int*, *deg_q: int*, *deg_r: int*, *rcond=None*, *fix_qr=None*) → *Hermite2*

Construct quadratic Hermite-Padé from Taylor expansion *an*.

Attributes

p
q
r
pade

gftool.hermpade.Hermite2.p

Hermite2.**p**: **Polynomial**

gftool.hermpade.Hermite2.q

`Hermite2.q: Polynomial`

gftool.hermpade.Hermite2.r

`Hermite2.r: Polynomial`

gftool.hermpade.Hermite2.pade

`Hermite2.pade: RatPol`

3.1.6 gftool.lattice

Collection of different lattices and their Green's functions.

The lattices are described by a tight binding Hamiltonian

$$H = t\sum_{\langle i,j\rangle} c_i^\dagger c_j,$$

where t is the hopping amplitude or integral. Mind the sign, often tight binding Hamiltonians are instead defined with a negative sign in front of t.

The Hamiltonian can be diagonalized

$$H = \sum_{k} \varepsilon_k c_k^\dagger c_k.$$

Typical quantities provided for the different lattices are:

gf_z
: The one-particle Green's function

$$G_{ii}(z) = \langle\langle c_i | c_i^\dagger \rangle\rangle(z) = 1/N \sum_k \frac{1}{z - \varepsilon_k}.$$

dos
: The density of states (DOS)

$$DOS(\varepsilon) = 1/N \sum_k \delta(\varepsilon - \varepsilon_k).$$

dos_moment
: The moments of the DOS

$$\varepsilon^{(m)} = \int d\varepsilon\, DOS(\varepsilon)\, \varepsilon^m$$

Submodules

bethe	Bethe lattice with infinite coordination number.
bethez	Bethe lattice for general coordination number Z.
box	Green's function corresponding to a box DOS.
onedim	1D lattice.
square	2D square lattice.
rectangular	2D rectangular lattice.
lieb	2D Lieb lattice.
triangular	2D triangular lattice.
honeycomb	2D honeycomb lattice.
kagome	2D Kagome lattice.
sc	3D simple cubic (sc) lattice.
bcc	3D body-centered cubic (bcc) lattice.
fcc	3D face-centered cubic (fcc) lattice.

gftool.lattice.bethe

Bethe lattice with infinite coordination number.

This is in fact no real lattice, but a tree. It corresponds to a semi-circular DOS.

half_bandwidth
The half_bandwidth corresponds to a scaled nearest neighbor hopping of $t=D/2$

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting Bethe lattice for infinite coordination number.
dos_moment(m, half_bandwidth)	Calculate the m th moment of the Bethe DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting Bethe lattice for infinite coordination number.
gf_d1_z(z, half_bandwidth)	First derivative of local Green's function of Bethe lattice for infinite coordination number.
gf_d2_z(z, half_bandwidth)	Second derivative of local Green's function of Bethe lattice for infinite coordination number.
gf_ret_t(tt, half_bandwidth[, center])	Retarded-time local Green's function of Bethe lattice with Z=∞.
gf_z(z, half_bandwidth)	Local Green's function of Bethe lattice for infinite coordination number.
gf_z_inv(gf, half_bandwidth)	Inverse of local Green's function of Bethe lattice for infinite coordination number.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the Bethe lattice.

gftool.lattice.bethe.dos

gftool.lattice.bethe.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting Bethe lattice for infinite coordination number.

Parameters

eps
[float array_like or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.bethe.dos_mp
Multi-precision version suitable for integration.

References

[economou2006]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.bethe.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```

gftool.lattice.bethe.dos_moment

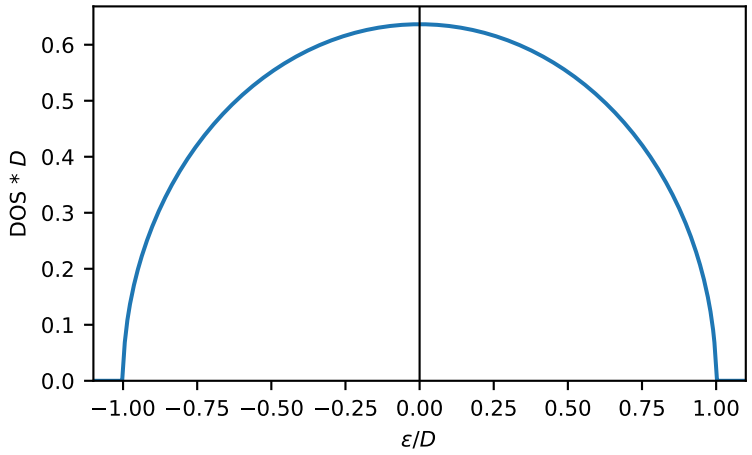
gftool.lattice.bethe.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the Bethe DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.



half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice.

Returns

float
The *m* th moment of the Bethe DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.bethe.dos

gftool.lattice.bethe.dos_mp

gftool.lattice.bethe.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting Bethe lattice for infinite coordination number.

This function is particularly suited to calculate integrals of the form $\int dDOS() f()$.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.bethe.dos
Vectorized version suitable for array evaluations.

References

[economou2006]

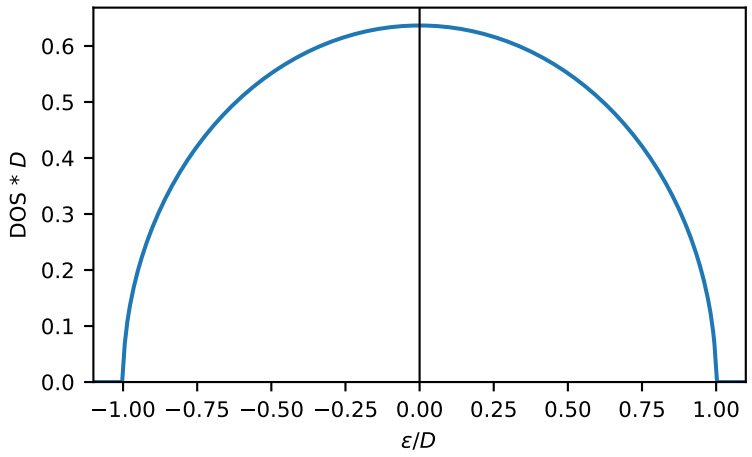
Examples

Calculate integrals:

```
>>> from mpmath import mp
>>> mp.quad(gt.lattice.bethe.dos_mp, [-1, 1])
mpf('1.0')
```

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos_mp = [gt.lattice.bethe.dos_mp(ee, half_bandwidth=1) for ee in eps]
>>> dos_mp = np.array(dos_mp, dtype=np.float64)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.bethe.gf_d1_z

gftool.lattice.bethe.**gf_d1_z**(*z*, *half_bandwidth*)

First derivative of local Green's function of Bethe lattice for infinite coordination number.

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the derivative of the Green's function.

See also:

gftool.lattice.bethe.gf_z

gftool.lattice.bethe.gf_d2_z

gftool.lattice.bethe.**gf_d2_z**(*z*, *half_bandwidth*)

Second derivative of local Green's function of Bethe lattice for infinite coordination number.

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the Green's function.

See also:

gftool.lattice.bethe.gf_z

gftool.lattice.bethe.gf_ret_t

gftool.lattice.bethe.**gf_ret_t**(*tt*, *half_bandwidth*, *center=0*)

Retarded-time local Green's function of Bethe lattice with Z=∞.

$$G(t) = -2j * (t) * J_1(Dt)/(Dt)$$

where D is the half bandwidth and $J_1(t)$ is the Bessel function of first kind.

Parameters

tt
[float array_like or float] Green's function is evaluated at time *tt*.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

center
[float] Position of the center of the Bethe DOS. This parameter is **not** given in units of *half_bandwidth*.

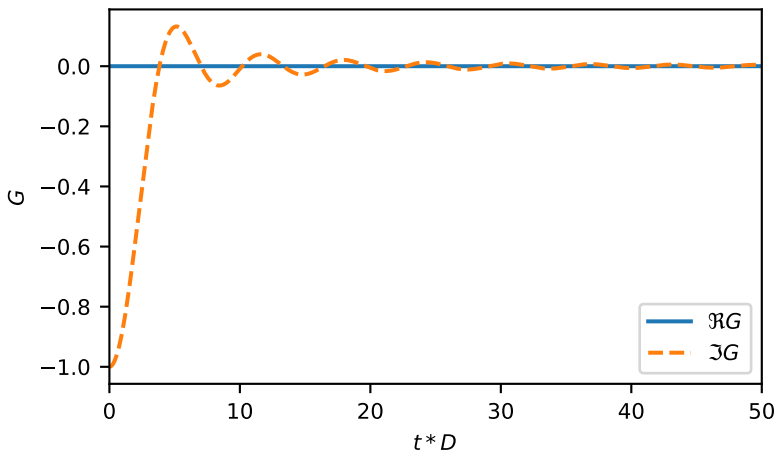
Returns

complex np.ndarray or complex
Value of the retarded-time Bethe Green's function.

Examples

```
>>> tt = np.linspace(0, 50, 1500)
>>> gf_tt = gt.lattice.bethe.gf_ret_t(tt, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(tt, gf_tt.real, label=r"$\Re G$")
>>> _ = plt.plot(tt, gf_tt.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$t*D$")
>>> _ = plt.ylabel(r"$G$")
>>> _ = plt.xlim(left=tt.min(), right=tt.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.bethe.gf_z

gftool.lattice.bethe.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of Bethe lattice for infinite coordination number.

$$G(z) = 2(z - s\sqrt{z^2 - D^2})/D^2$$

where D is the half bandwidth and $s = sgn[]$. See [georges1996].

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the Bethe Green's function.

References

[georges1996]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.bethe.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```

gftool.lattice.bethe.gf_z_inv

gftool.lattice.bethe.**gf_z_inv**(*gf*, *half_bandwidth*)

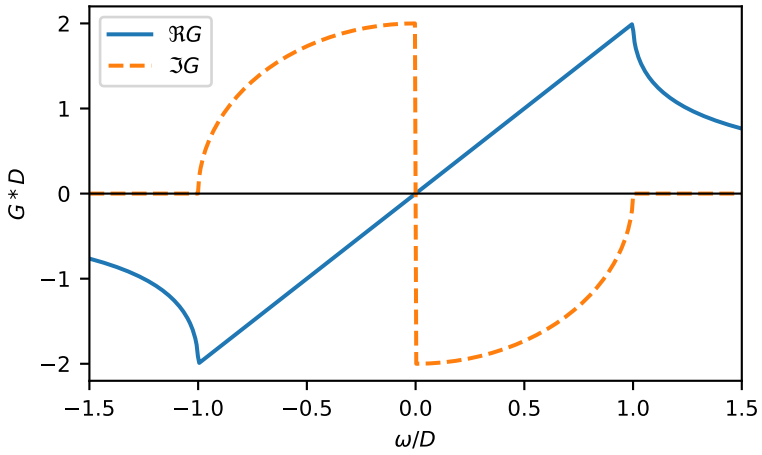
Inverse of local Green's function of Bethe lattice for infinite coordination number.

$$R(G) = (D/2)^2 G + 1/G$$

where $R(z) = G^{-1}(z)$ is the inverse of the Green's function.

Parameters

gf
[complex array_like or complex] Value of the local Green's function.



half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
The inverse of the Bethe Green's function *gf_z(gf_z_inv(g, D), D)=g*.

See also:

gftool.lattice.bethe.gf_z

References

[georges1996]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500) + 1e-4j
>>> gf_ww = gt.lattice.bethe.gf_z(ww, half_bandwidth=1)
>>> np.allclose(ww, gt.lattice.bethe.gf_z_inv(gf_ww, half_bandwidth=1))
True
```

gftool.lattice.bethe.hilbert_transform

gftool.lattice.bethe.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the Bethe lattice.

The Hilbert transform is defined as:

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex array_like or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.bethe.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D:

$$2t = D$$

gftool.lattice.bethez

Bethe lattice for general coordination number Z.

In the limit of infinite coordination number $Z=\infty$, this becomes `gftool.lattice.bethe`, in the opposite limit of minimal coordination number $Z=2$`, this is `gftool.lattice.onedim`.

API

Functions

`dos`(eps, half_bandwidth, coordination)	DOS of non-interacting Bethe lattice for *coordination*.
`gf_z`(z, half_bandwidth, coordination)	Local Green's function of Bethe lattice for *coordination*.

gftool.lattice.bethez.dos

gftool.lattice.bethez.**dos**(*eps*, *half_bandwidth*, *coordination*)

DOS of non-interacting Bethe lattice for *coordination*.

Parameters

eps
[float ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0.

coordination
[int] Coordination number of the Bethe lattice.

Returns

float ndarray or float
The value of the DOS.

See also:

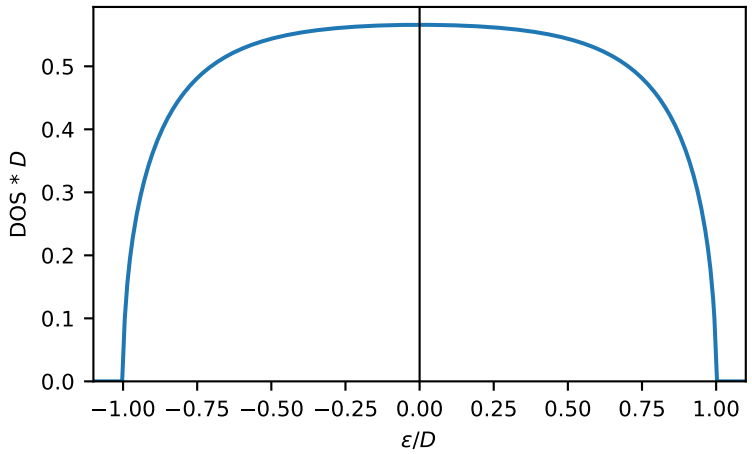
gftool.lattice.bethe.dos
Case for *coordination=np.infty*.

gftool.lattice.onedim.dos
Case for *coordination=2*.

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.bethez.dos(eps, half_bandwidth=1, coordination=9)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.bethez.gf_z

gftool.lattice.bethez.**gf_z**(*z*, *half_bandwidth*, *coordination*)

Local Green's function of Bethe lattice for *coordination*.

$$G(z) = 2(Z-1)/z/((Z-2) + Z\sqrt{1 - D^2/z^2})$$

where D is the *half_bandwidth* and Z the`coordination`. See [economou2006].

Parameters

z
[complex ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice.

coordination
[int] Coordination number of the Bethe lattice.

Returns

complex ndarray or complex
Value of the Bethe Green's function.

See also:

gftool.lattice.bethe.gf_z
Case for *coordination=np.infty*.

gftool.lattice.onedim.gf_z
Case for *coordination=2*.

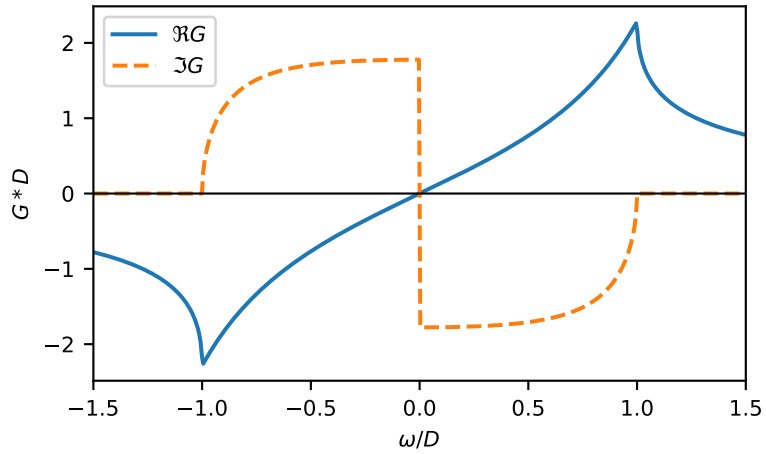
References

[economou2006]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.bethez.gf_z(ww, half_bandwidth=1, coordination=9)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.box

Green's function corresponding to a box DOS.

This doesn't correspond to any real lattice. It is mostly meant as very simple test case, for which we have analytic expressions.

API

Functions

dos(eps, half_bandwidth)	Box-shaped DOS.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the box DOS.
gf_ret_t(tt, half_bandwidth[, center])	Local retarded-time local Green's function corresponding to a box DOS.
gf_z(z, half_bandwidth)	Local Green's function corresponding to a box DOS.

gftool.lattice.box.dos

gftool.lattice.box.**dos**(*eps*, *half_bandwidth*)

Box-shaped DOS.

Parameters

eps
[float array_like or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0.

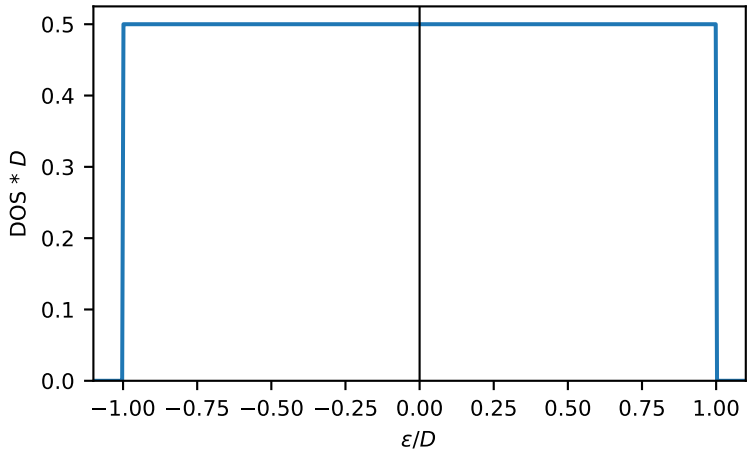
Returns

float np.ndarray or float
The value of the DOS.

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.box.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.box.dos_moment

gftool.lattice.box.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the box DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the box lattice.

Returns

float
The *m* th moment of the box DOS.

See also:

gftool.lattice.box.dos

gftool.lattice.box.gf_ret_t

gftool.lattice.box.**gf_ret_t**(*tt*, *half_bandwidth*, *center=0*)

Local retarded-time local Green's function corresponding to a box DOS.

$$G(t) = -1j(t)\sin(Dt)/Dt$$

where D is the half bandwidth.

Parameters

tt
[float array_like or float] Green's function is evaluated at time *tt*.

half_bandwidth
[float] Half-bandwidth of the box DOS.

center
[float] Position of the center of the box DOS. This parameter is **not** given in units of *half_bandwidth*.

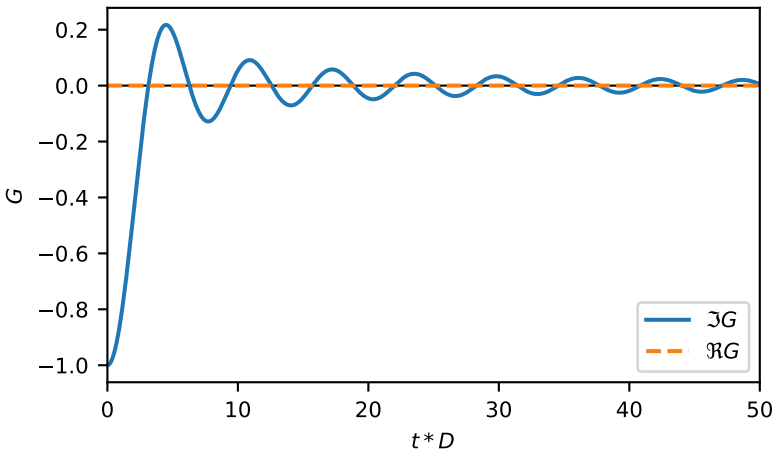
Returns

complex np.ndarray or complex
Value of the retarded-time Green's function corresponding to a box DOS.

Examples

```
>>> tt = np.linspace(0, 50, 1500)
>>> gf_tt = gt.lattice.box.gf_ret_t(tt, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(tt, gf_tt.imag, label=r"$\Im G$")
>>> _ = plt.plot(tt, gf_tt.real, '--', label=r"$\Re G$")
>>> _ = plt.xlabel(r"$t*D$")
>>> _ = plt.ylabel(r"$G$")
>>> _ = plt.xlim(left=tt.min(), right=tt.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.box.gf_z

gftool.lattice.box.**gf_z**(*z*, *half_bandwidth*)

Local Green's function corresponding to a box DOS.

$$G(z) = \ln(\frac{z+D}{z-D})/2D$$

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the box DOS.

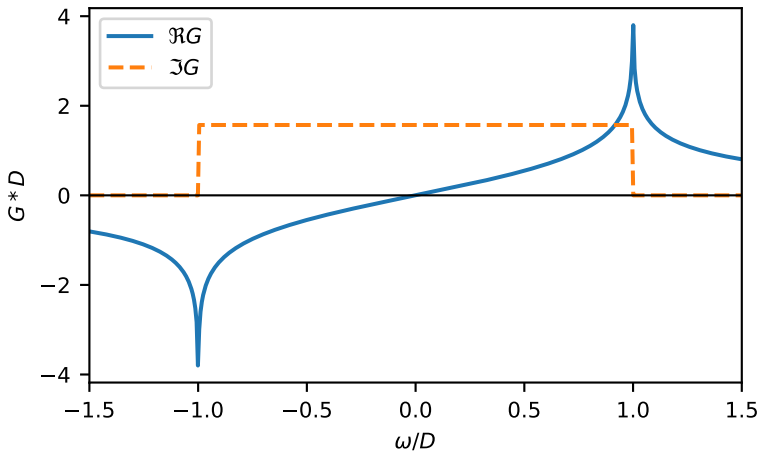
Returns

complex np.ndarray or complex
Value of the Green's function corresponding to a box DOS.

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.box.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.onedim

1D lattice.

The dispersion of the 1D lattice is given by

$$_k = 2t\cos(k)$$

which takes values in $_k \in [-2t, +2t] = [-D, +D]$.

half_bandwidth
The half_bandwidth corresponds to a nearest neighbor hopping of *t=D/2*

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 1D lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the 1D DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 1D lattice.
gf_z(z, half_bandwidth)	Local Green's function of the 1D lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the 1D lattice.

gftool.lattice.onedim.dos

gftool.lattice.onedim.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 1D lattice.

Diverges at the band-edges *abs(eps) = half_bandwidth*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.onedim.dos_mp
Multi-precision version suitable for integration.

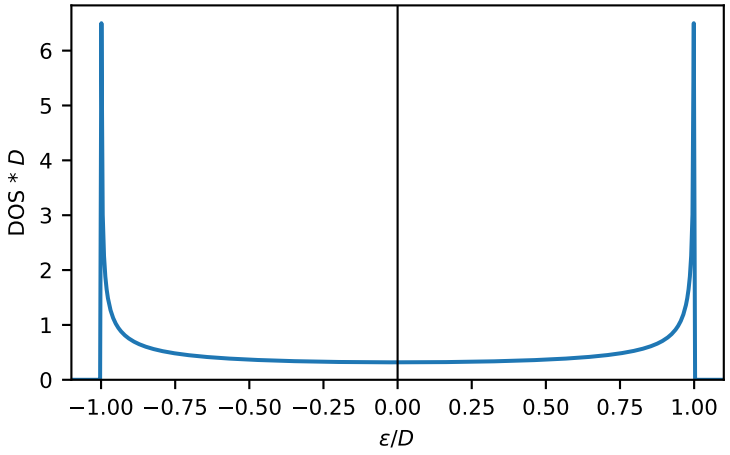
References

[economou2006]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=501)
>>> dos = gt.lattice.onedim.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.onedim.dos_moment

gftool.lattice.onedim.**dos_moment**(*m*, *half_bandwidth*)

Calculate the m th moment of the 1D DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 1D lattice.

Returns

float
The m th moment of the 1D DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments m.

See also:

gftool.lattice.onedim.dos

gftool.lattice.onedim.dos_mp

gftool.lattice.onedim.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 1D lattice.

Diverges at the band-edges *abs(eps) = half_bandwidth*.

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider removing singularities, e.g. split the integral

$$\int^0 dDOS()[f() - f(-D)] + \int_0 dDOS()[f() - f(+D)] + [f(-D) + f(+D)]/2$$

or symmetrize the integral.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_likeloat] Half-bandwidth of the DOS, DOS(|*eps*| > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.onedim.dos
Vectorized version suitable for array evaluations.

References

[economou2006]

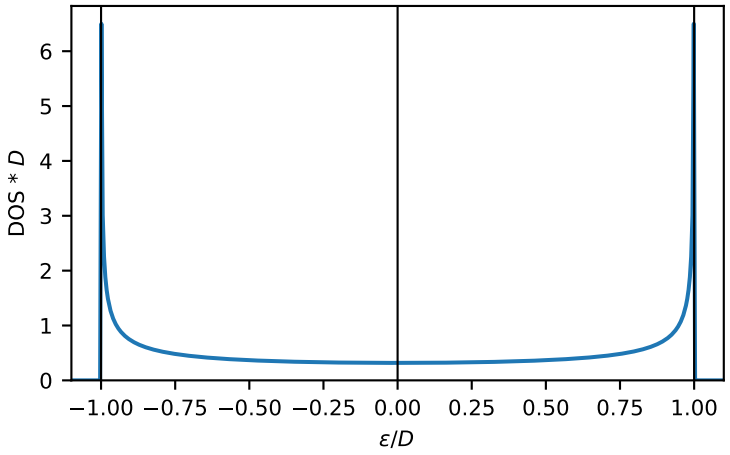
Examples

Calculate integrals (the 1D DOS needs higher accuracy for accurate results):

```
>>> from mpmath import mp
>>> with mp.workdps(35, normalize_output=True):
...     norm = mp.quad(gt.lattice.onedim.dos_mp, [-1, +1])
>>> norm
mpf('1.0')
```

```
>>> eps = np.linspace(-1.1, 1.1, num=501)
>>> dos_mp = [gt.lattice.onedim.dos_mp(ee, half_bandwidth=1) for ee in eps]
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos_mp)
>>> for pos in (-1, 0, +1):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.onedim.gf_z

gftool.lattice.onedim.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 1D lattice.

$$G(z) = \frac{1}{2}\int_{-} \frac{d}{z - D\cos()}$$

where D is the half bandwidth. The integral can be evaluated in the complex plane along the unit circle. See [economou2006].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the 1D lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t=D/2$.

Returns

complex np.ndarray or complex
Value of the square lattice Green's function.

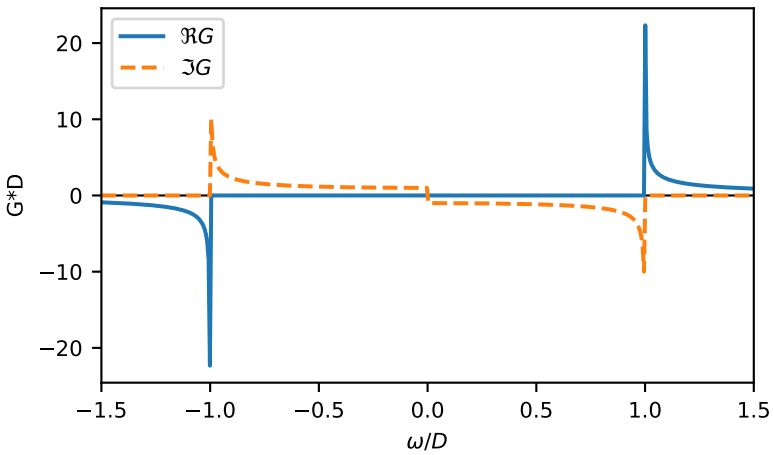
References

[economou2006]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.onedim.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel("G*D")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.onedim.hilbert_transform

gftool.lattice.onedim.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the 1D lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 1D lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.onedim.gf_z

Notes

Relation between nearest neighbor hopping *t* and half-bandwidth *D*

$$2t = D$$

gftool.lattice.square

2D square lattice.

The dispersion of the 2D square lattice is given by

$$_{k_x,k_y} = 2t[\cos(k_x) + \cos(k_y)]$$

which takes values in $_{k_x,k_y} \in [-4t, +4t] = [-D, +D]$.

half_bandwidth
The half_bandwidth corresponds to a nearest neighbor hopping of *t=D/4*

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 2D square lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the square DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 2D square lattice.
gf_z(z, half_bandwidth)	Local Green's function of the 2D square lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the square lattice.
stress_trafo(xi, half_bandwidth)	Single pole integration over the stress tensor function.

gftool.lattice.square.dos

gftool.lattice.square.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D square lattice.

Has a van Hove singularity (logarithmic divergence) at *eps = 0*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(|*eps*| > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/4*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.square.dos_mp
Multi-precision version suitable for integration.

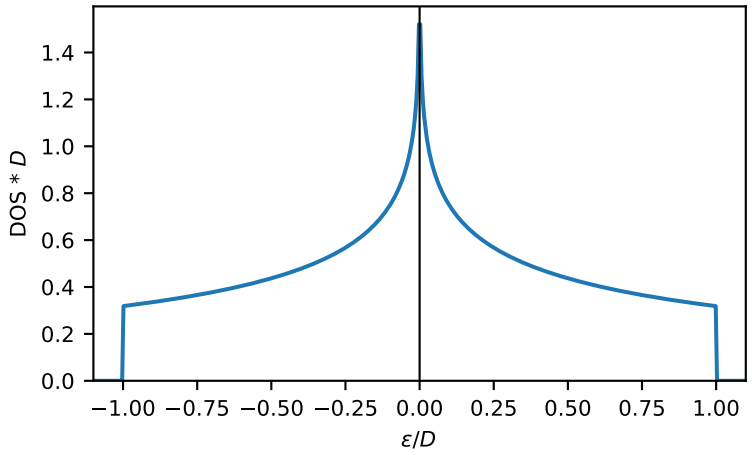
References

[economou2006]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.square.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.square.dos_moment

gftool.lattice.square.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the square DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D square lattice.

Returns

float
The *m* th moment of the 2D square DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.square.dos

gftool.lattice.square.dos_mp

gftool.lattice.square.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 2D square lattice.

Has a van Hove singularity (logarithmic divergence) at *eps = 0*.

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider using $\int dDOS()[f() - f(0)] + f(0)$ to avoid the singularity.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/4*.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.square.dos
Vectorized version suitable for array evaluations.

References

[economou2006]

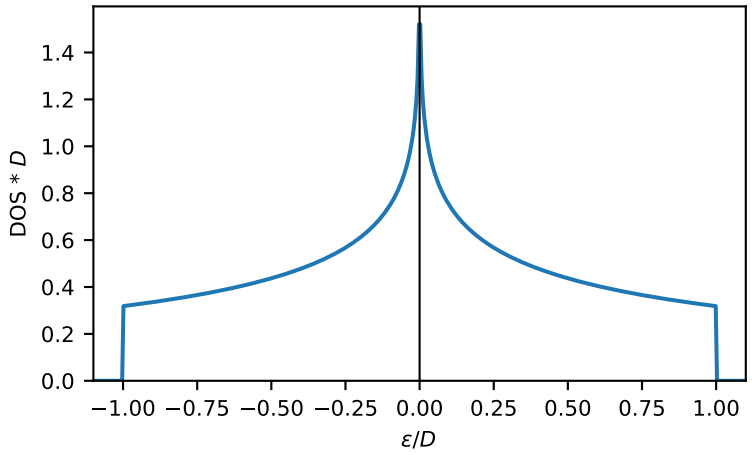
Examples

Calculate integrals:

```
>>> from mpmath import mp
>>> mp.quad(gt.lattice.square.dos_mp, [-1, 0, 1])
mpf('1.0')
```

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos_mp = [gt.lattice.square.dos_mp(ee, half_bandwidth=1) for ee in eps]
>>> dos_mp = np.array(dos_mp, dtype=np.float64)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.square.gf_z

gftool.lattice.square.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D square lattice.

$$G(z) = \frac{2}{z}\int_0^{\nicefrac{\pi}{2}} \frac{d}{\sqrt{1-(D/z)^2\cos^2}}$$

where D is the half bandwidth and the integral is the complete elliptic integral of first kind. See [economou2006].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the square lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/4*.

Returns

complex np.ndarray or complex
Value of the square lattice Green's function.

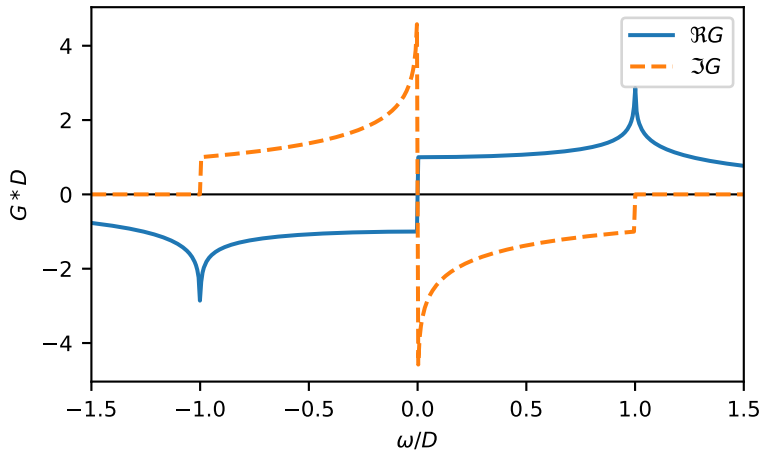
References

[economou2006]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.square.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.square.hilbert_transform

gftool.lattice.square.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the square lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D square lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.square.gf_z

Notes

Relation between nearest neighbor hopping *t* and half-bandwidth *D*

$$4t = D$$

gftool.lattice.square.stress_trafo

gftool.lattice.square.**stress_trafo**(*xi*, *half_bandwidth*)
Single pole integration over the stress tensor function.

In analogy to the Hilbert transformation, we define the stress tensor transformation as

$$T() = \int d\tilde{}_{xx}()/(-)$$

with the stress tensor function

$$\tilde{}_{xx}() := \sum_k \partial^2/\partial k_x^2(- _k) = -0.5 * * DOS()$$

Parameters

xi
[complex or complex array_like] Point of evaluation of the transformation.

half_bandwidth
[float] Half-bandwidth of the square lattice.

Returns

complex np.ndarray
Stress tensor function at *xi*.

References

[arsenault2013]

gftool.lattice.rectangular

2D rectangular lattice.

The dispersion of the 2D rectangular lattice is given by

$$_{k_x,k_y} = 2t[\cos(k_x) + \cos(k_y)]$$

where is the *scale*. The DOS has a singularity at $2t(- 1) = D(- 1)/(+ 1)$.

half_bandwidth
The half-bandwidth *D* corresponds to a nearest neighbor hopping of *t=D/2/(scale + 1)*

scale
Relative scale of the different hopping *t_1 = scale*t_2*.

API

Functions

dos(eps, half_bandwidth, scale)	DOS of non-interacting 2D rectangular lattice.
gf_z(z, half_bandwidth, scale)	Local Green's function of the 2D rectangular lattice.
hilbert_transform(xi, half_bandwidth, scale)	Hilbert transform of non-interacting DOS of the rectangular lattice.

gftool.lattice.rectangular.dos

gftool.lattice.rectangular.**dos**(*eps*, *half_bandwidth*, *scale*)

DOS of non-interacting 2D rectangular lattice.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(|*eps*| > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/2/(scale + 1)$.

scale
[float] Relative scale of the different hoppings $t_1 = scale * t_2$. *scale=1* corresponds to the square lattice.

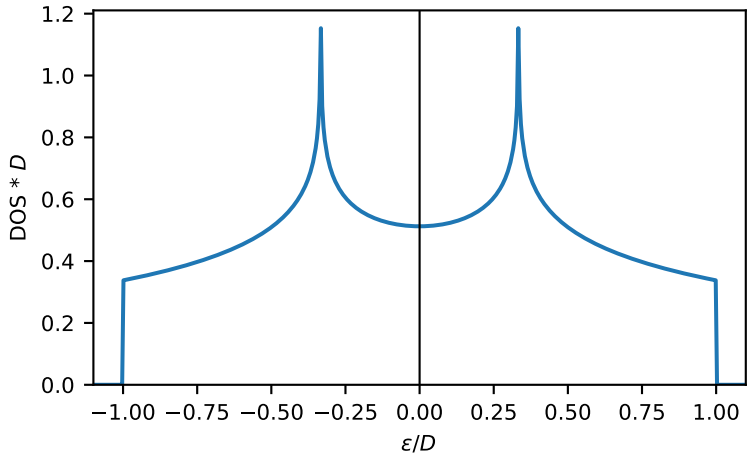
Returns

float np.ndarray or float
The value of the DOS.

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.rectangular.dos(eps, half_bandwidth=1, scale=2)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.rectangular.gf_z

gftool.lattice.rectangular.**gf_z**(*z*, *half_bandwidth*, *scale*)

Local Green's function of the 2D rectangular lattice.

$$G(z) = \frac{1}{}\int_0 \frac{d}{\sqrt{(z - \cos)^2 - 1}}$$

where is the *scale*, the hopping is chosen *t=1*, the *half_bandwidth* is $D = 2(1 +)$. The integral is the complete elliptic integral of first kind. See [morita1971].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the rectangular lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/2/(scale + 1)$.

scale
[float] Relative scale of the different hoppings $t_1 = scale * t_2$. *scale=1* corresponds to the square lattice.

Returns

complex np.ndarray or complex
Value of the rectangular lattice Green's function.

See also:

gftool.lattice.square.gf_z
Green's function in the limit *scale=1*.

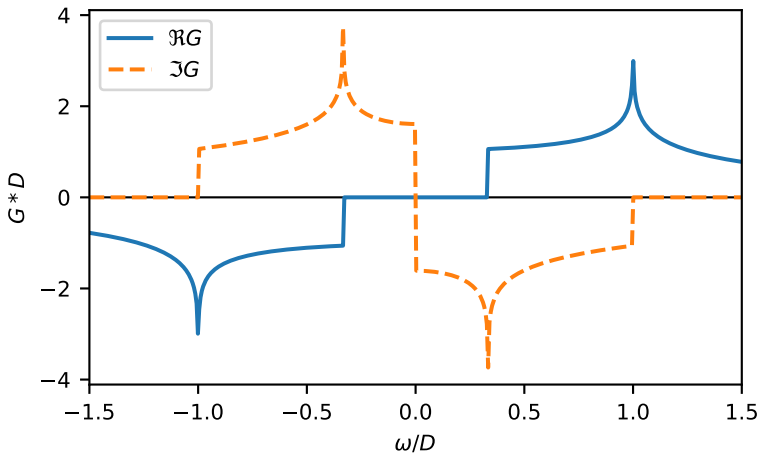
References

[morita1971]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500, dtype=complex)
>>> gf_ww = gt.lattice.rectangular.gf_z(ww, half_bandwidth=1, scale=2)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.rectangular.hilbert_transform

gftool.lattice.rectangular.**hilbert_transform**(*xi*, *half_bandwidth*, *scale*)

Hilbert transform of non-interacting DOS of the rectangular lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi

[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D rectangular lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/2/(scale + 1)$.

scale
[float] Relative scale of the different hoppings $t_1 = scale * t_2$. *scale=1* corresponds to the square lattice.

Returns

complex np.ndarray or complex
Hilbert transform at *xi*.

See also:

gftool.lattice.rectangular.gf_z

Notes

Relation between nearest neighbor hopping *t*, scale γ and half-bandwidth *D*

$$2(+1)t = D$$

gftool.lattice.lieb

2D Lieb lattice.

The lieb lattice can be decomposed into *square* and a dispersionless flat band.

half_bandwidth
The half-bandwidth *D* corresponds to a nearest neighbor hopping of *t=D * 2**1.5*

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 2D Lieb lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the Lieb DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 2D lieb lattice.
gf_z(z, half_bandwidth)	Local Green's function of the 2D Lieb lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the lieb lattice.

gftool.lattice.lieb.dos

gftool.lattice.lieb.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D Lieb lattice.

The delta-peak at *eps=0* is **ommited** and must be treated seperately! Without it, the DOS integrates to *2/3*.

Besides the delta-peak, the DOS diverges at *eps=±half_bandwidth/2**0.5*.

The Green's function and therefore the DOS of the 2D Lieb lattice can be expressed in terms of the 2D square lattice *gftool.lattice.square.dos*, see [kogan2021].

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D * 2**1.5*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.lieb.dos_mp
Multi-precision version suitable for integration.

gftool.lattice.square.dos

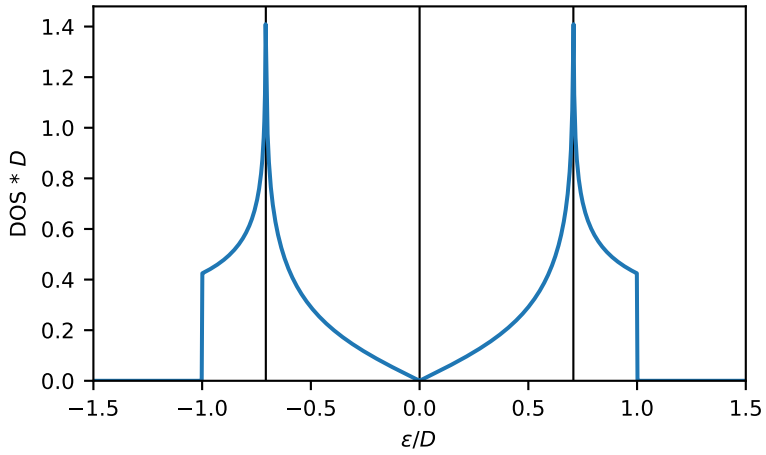
References

[kogan2021]

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=1001)
>>> dos = gt.lattice.lieb.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-2**-0.5, 0, +2**-0.5):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.lieb.dos_moment

gftool.lattice.lieb.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the Lieb DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D Lieb lattice.

Returns

float
The *m* th moment of the 2D Lieb DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.lieb.dos

gftool.lattice.lieb.dos_mp

gftool.lattice.lieb.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 2D lieb lattice.

The delta-peak at *eps=0* is **ommited** and must be treated seperately! Without it, the DOS integrates to *2/3*.

Besides the delta-peak, the DOS diverges at *eps=±half_bandwidth/2**0.5*.

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider removing singularities, e.g. split the integral

$$\int^{0} dDOS()[f() - f(-D/\sqrt{2})] + \int_{0} dDOS()[f() - f(+D/\sqrt{3})] \\ +[f(-D/\sqrt{2}) + f(0) + f(+D/\sqrt{2})]/3$$

where D is the *half_bandwidth*, or symmetrize the integral.

The Green's function and therefore the DOS of the 2D Lieb lattice can be expressed in terms of the 2D square lattice *gftool.lattice.square.dos*, see [kogan2021].

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D * 2**1.5*.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.lieb.dos
Vectorized version suitable for array evaluations.

gftool.lattice.square.dos_mp

References

[kogan2021]

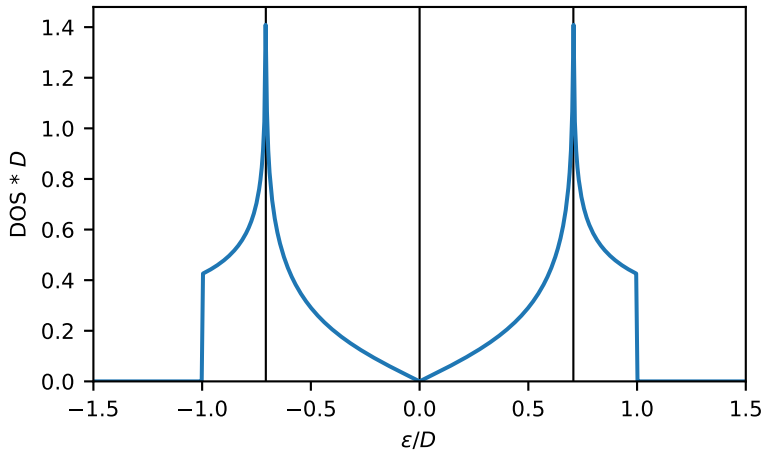
Examples

Calculated integrals

```
>>> from mpmath import mp
>>> mp.identify(mp.quad(gt.lattice.lieb.dos_mp, [-1, -2**-0.5, 0, 2**-0.5, 1]))
'(2/3)'
```

```
>>> eps = np.linspace(-1.5, 1.5, num=501)
>>> dos_mp = [gt.lattice.lieb.dos_mp(ee, half_bandwidth=1) for ee in eps]
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-2**-0.5, 0, +2**-0.5):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.lieb.gf_z

gftool.lattice.lieb.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D Lieb lattice.

The Green's function of the 2D Lieb lattice can be expressed in terms of the 2D square lattice *gftool.lattice.square.gf_z*, and a non-dispersive peak, see [kogan2021].

The Green's function has singularities for *z/half_bandwidth in [-2**-0.5, 0, 2**-0.5]*.

Parameters

z
: [complex np.ndarray or complex] Green's function is evaluated at complex frequency *z*.

half_bandwidth
: [float] Half-bandwidth of the DOS of the Lieb lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D * 2**1.5*.

Returns

complex np.ndarray or complex
: Value of the Lieb lattice Green's function.

See also:

gftool.lattice.square.gf_z

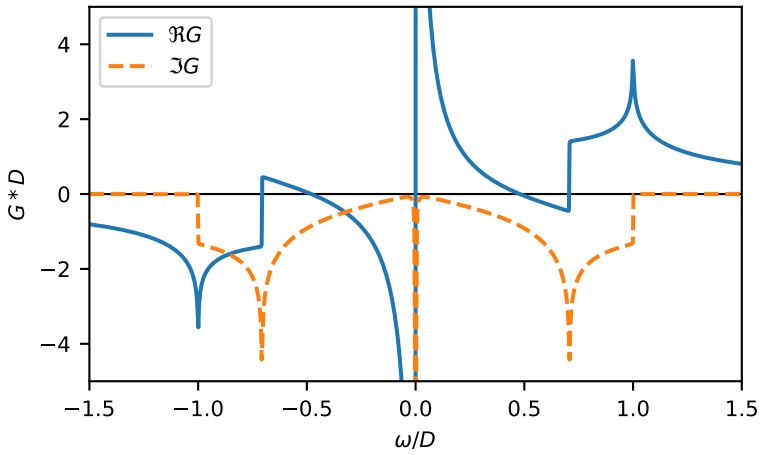
References

[kogan2021]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=1001, dtype=complex) + 1e-4j
>>> gf_ww = gt.lattice.lieb.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.ylim(bottom=-5.0, top=5.0)
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.lieb.hilbert_transform

gftool.lattice.lieb.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the lieb lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D lieb lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.lieb.gf_z

Notes

Relation between nearest neighbor hopping *t* and half-bandwidth *D*

$$t = D2^{3/2}$$

gftool.lattice.triangular

2D triangular lattice.

The dispersion of the 2D triangular lattice is given by

$$_{k_x,k_y} = t[\cos(2k_x) + 2\cos(k_x)\cos(k_y)]$$

which takes values $_{k_x,k_y} \in [-1.5t, 3t] = [-2D/3, 4D/3]$.

half_bandwidth
The half-bandwidth *D* corresponds to a nearest neighbor hopping of *t=4D/9*.

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 2D triangular lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the triangular DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 2D triangular lattice.
gf_z(z, half_bandwidth)	Local Green's function of the 2D triangular lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the triangular lattice.

gftool.lattice.triangular.dos

gftool.lattice.triangular.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D triangular lattice.

The DOS diverges at *-4/9*half_bandwidth*. The DOS is evaluated as complete elliptic integral of first kind, see [kogan2021].

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(*eps* < -2/3`half_bandwidth`) = 0, DOS(4/3`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 4D/9$.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.triangular.dos_mp
Multi-precision version suitable for integration.

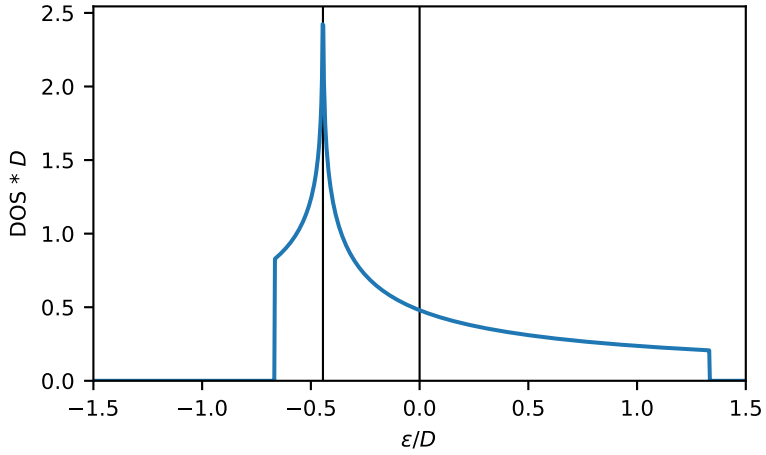
References

[kogan2021]

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=1000)
>>> dos = gt.lattice.triangular.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(-4/9, color='black', linewidth=0.8)
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.triangular.dos_moment

gftool.lattice.triangular.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the triangular DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D triangular lattice.

Returns

float
The *m* th moment of the 2D triangular DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.triangular.dos

gftool.lattice.triangular.dos_mp

gftool.lattice.triangular.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 2D triangular lattice.

The DOS diverges at *-4/9*half_bandwidth*.

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider using $\int dDOS()[f() - f(-4/9)] + f(-4/9)$ to avoid the singularity.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(*eps* < -2/3`half_bandwidth`) = 0, DOS(4/3`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 4D/9$.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.triangular.dos
Vectorized version suitable for array evaluations.

References

[kogan2021]

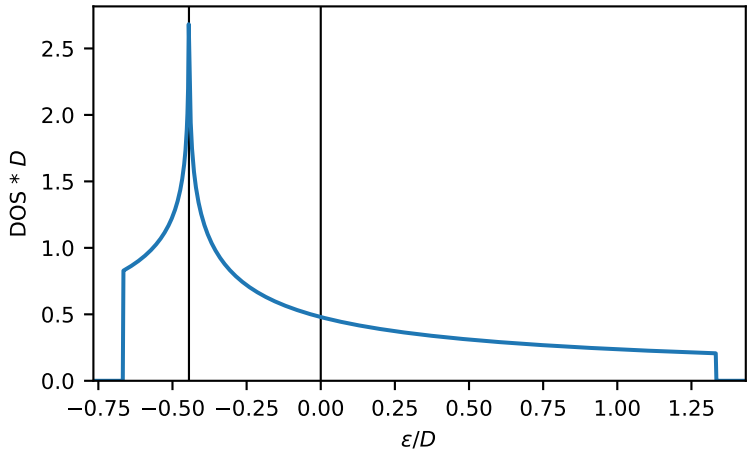
Examples

Calculate integrals:

```
>>> from mpmath import mp
>>> mp.quad(gt.lattice.triangular.dos_mp, [-2/3, -4/9, 4/3])
mpf('1.0')
```

```
>>> eps = np.linspace(-2/3 - 0.1, 4/3 + 0.1, num=1000)
>>> dos_mp = [gt.lattice.triangular.dos_mp(ee, half_bandwidth=1) for ee in eps]
>>> dos_mp = np.array(dos_mp, dtype=np.float64)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(-4/9, color='black', linewidth=0.8)
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.triangular.gf_z

gftool.lattice.triangular.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D triangular lattice.

Note, that the spectrum is asymmetric and in $[-2D/3, 4D/3]$, where D is the half-bandwidth. The Green's function is evaluated as complete elliptic integral of first kind, see [horiguchi1972].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the triangular lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 4D/9$.

Returns

complex np.ndarray or complex
Value of the triangular lattice Green's function.

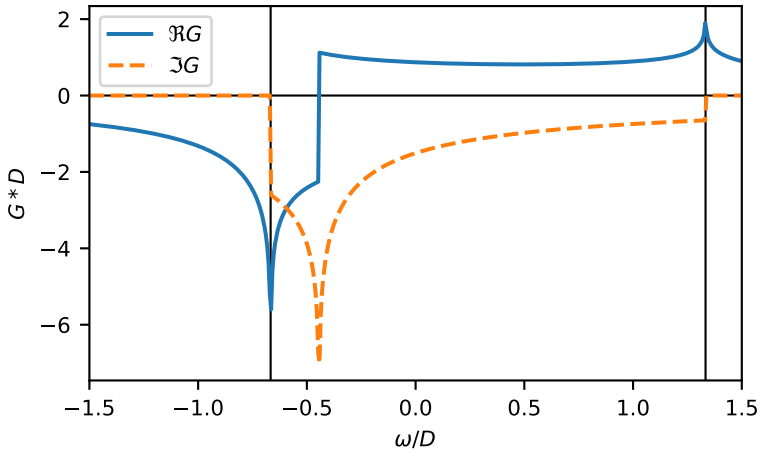
References

[horiguchi1972]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500, dtype=complex) + 1e-64j
>>> gf_ww = gt.lattice.triangular.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.axvline(-2/3, color='black', linewidth=0.8)
>>> _ = plt.axvline(+4/3, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.triangular.hilbert_transform

gftool.lattice.triangular.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the triangular lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D triangular lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.triangular.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$9t = 4D$$

gftool.lattice.honeycomb

2D honeycomb lattice.

The honeycomb lattice can be decomposed into *triangular* sublattices.

half_bandwidth
The half-bandwidth *D* corresponds to a nearest neighbor hopping of *t=2D/3*.

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 2D honeycomb lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the honeycomb DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 2D honeycomb lattice.
gf_z(z, half_bandwidth)	Local Green's function of the 2D honeycomb lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the honeycomb lattice.

gftool.lattice.honeycomb.dos

gftool.lattice.honeycomb.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D honeycomb lattice.

The DOS diverges at *eps*=±*half_bandwidth/3*. The Green's function and therefore the DOS of the 2D honeycomb lattice can be expressed in terms of the 2D triangular lattice *gftool.lattice.triangular.dos*, see [horiguchi1972].

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.honeycomb.dos_mp
Multi-precision version suitable for integration.

gftool.lattice.triangular.dos

References

[horiguchi1972]

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=501)
>>> dos = gt.lattice.honeycomb.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-1/3, 0, +1/3):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```

gftool.lattice.honeycomb.dos_moment

gftool.lattice.honeycomb.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the honeycomb DOS.

The moments are defined as $\int d^m DOS()$.

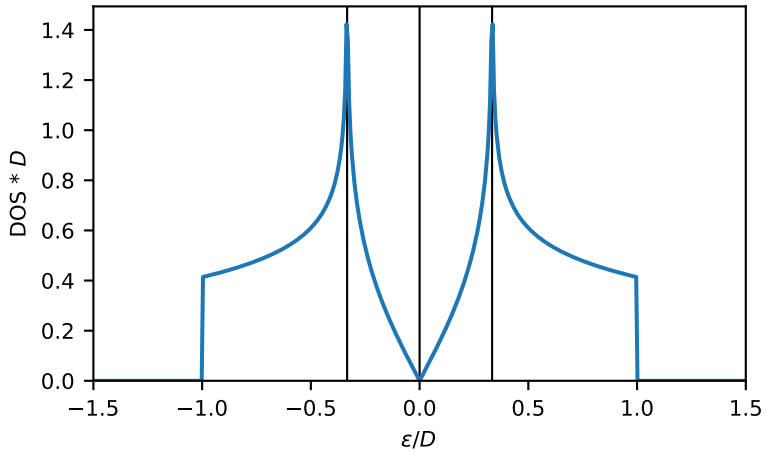
Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D honeycomb lattice.

Returns

float
The *m* th moment of the 2D honeycomb DOS.



Raises

NotImplementedError
Currently only implemented for a few specific moments m.

See also:

gftool.lattice.honeycomb.dos

gftool.lattice.honeycomb.dos_mp

gftool.lattice.honeycomb.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 2D honeycomb lattice.

The DOS diverges at *eps*=±*half_bandwidth/3*.

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider removing singularities, e.g. split the integral

$$\int^{0} dDOS()[f() - f(-D/3)] + \int_{0} dDOS()[f() - f(+D/3)] + [f(-D/3) + f(+D/3)]/2$$

where D is the *half_bandwidth*, or symmetrize the integral.

The Green's function and therefore the DOS of the 2D honeycomb lattice can be expressed in terms of the 2D triangular lattice *gftool.lattice.triangular.dos_mp*, see [horiguchi1972].

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.honeycomb.dos
Vectorized version suitable for array evaluations.

gftool.lattice.triangular.dos_mp

References

[horiguchi1972]

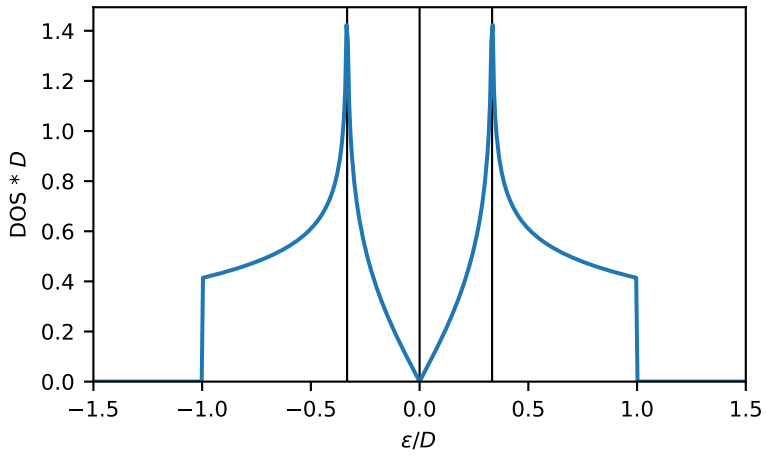
Examples

Calculated integrals

```
>>> from mpmath import mp
>>> mp.quad(gt.lattice.honeycomb.dos_mp, [-1, -1/3, 0, +1/3, +1])
mpf('1.0')
```

```
>>> eps = np.linspace(-1.5, 1.5, num=501)
>>> dos_mp = [gt.lattice.honeycomb.dos_mp(ee, half_bandwidth=1) for ee in eps]
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-1/3, 0, +1/3):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.honeycomb.gf_z

gftool.lattice.honeycomb.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D honeycomb lattice.

The Green's function of the 2D honeycomb lattice can be expressed in terms of the 2D triangular lattice *gftool.lattice.triangular.gf_z*, see [horiguchi1972].

The Green's function has singularities at z=±*half_bandwidth/3*.

Parameters

z
: [complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
: [float] Half-bandwidth of the DOS of the honeycomb lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

complex np.ndarray or complex
: Value of the honeycomb lattice Green's function.

See also:

gftool.lattice.triangular.gf_z

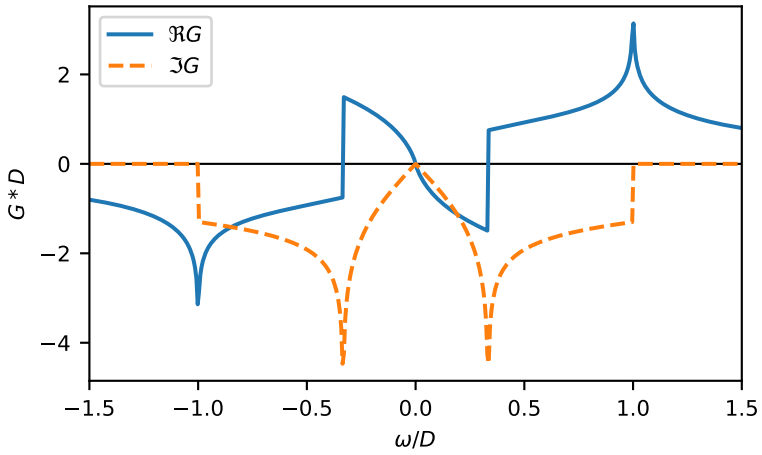
References

[horiguchi1972]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=501, dtype=complex) + 1e-64j
>>> gf_ww = gt.lattice.honeycomb.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.honeycomb.hilbert_transform

gftool.lattice.honeycomb.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the honeycomb lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D honeycomb lattice.

Returns

complex ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.honeycomb.gf_z

Notes

Relation between nearest neighbor hopping *t* and half-bandwidth *D*

$$3t/2 = D$$

gftool.lattice.kagome

2D Kagome lattice.

The DOS is finite in the interval $[-2D/3, 4D/3]$, where D is the half-bandwidth.

The kagome lattice can be decomposed into `triangular` and a dispersionless flat band. The dispersive part looks like the `honeycomb` lattice.

half_bandwidth
The half-bandwidth *D* corresponds to a nearest neighbor hopping of *t=2D/3*

API

Functions

`dos`(eps, half_bandwidth)	DOS of non-interacting 2D kagome lattice.
`dos_moment`(m, half_bandwidth)	Calculate the *m* th moment of the kagome DOS.
`dos_mp`(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 2D kagome lattice.
`gf_z`(z, half_bandwidth)	Local Green's function of the 2D kagome lattice.
`hilbert_transform`(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the kagome lattice.

gftool.lattice.kagome.dos

gftool.lattice.kagome.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D kagome lattice.

The delta-peak at *eps=-2*half_bandwidth/3* is **ommited** and must be treated seperately! Without it, the DOS integrates to *2/3*.

Besides the delta-peak, the DOS diverges at *eps=0* and *eps=2*half_bandwidth/3*.

The Green's function and therefore the DOS of the 2D kagome lattice can be expressed in terms of the 2D triangular lattice `gftool.lattice.triangular.dos`, see [kogan2021]. Omitting the non-dispersive peak, it corresponds to `gftool.lattice.honeycomb.dos` shifted by *half_bandwidth/3*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(*eps* < -2/3`half_bandwidth`) = 0, DOS(4/3`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.kagome.dos_mp
Multi-precision version suitable for integration.

gftool.lattice.triangular.dos
gftool.lattice.honeycomb.dos

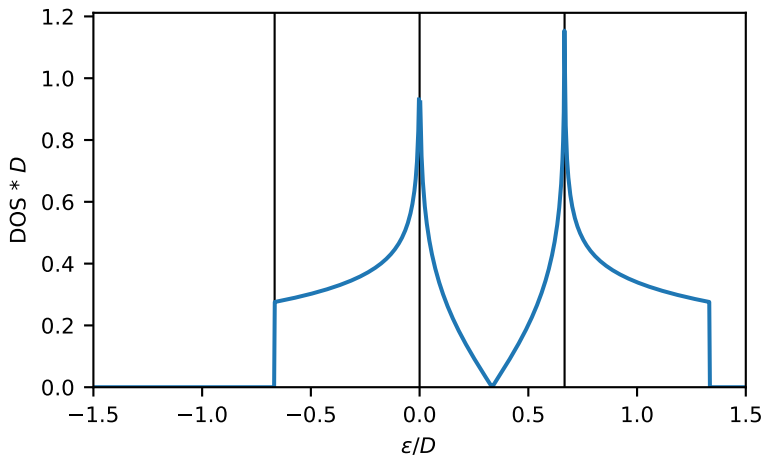
References

[varm2013], [kogan2021]

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=1001)
>>> dos = gt.lattice.kagome.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-2/3, 0, +2/3):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.kagome.dos_moment

gftool.lattice.kagome.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the kagome DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D kagome lattice.

Returns

float
The *m* th moment of the 2D kagome DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.kagome.dos

gftool.lattice.kagome.dos_mp

gftool.lattice.kagome.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 2D kagome lattice.

The delta-peak at *eps=-2*half_bandwidth/3* is **ommited** and must be treated seperately! Without it, the DOS integrates to *2/3*.

Besides the delta-peak, the DOS diverges at *eps=0* and *eps=2*half_bandwidth/3*.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(*eps* < -2/3`half_bandwidth`) = 0, DOS(4/3`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.kagome.dos_mp
Vectorized version suitable for array evaluations.

gftool.lattice.triangular.dos_mp
gftool.lattice.honeycomb.dos_mp

References

[varm2013], [kogan2021]

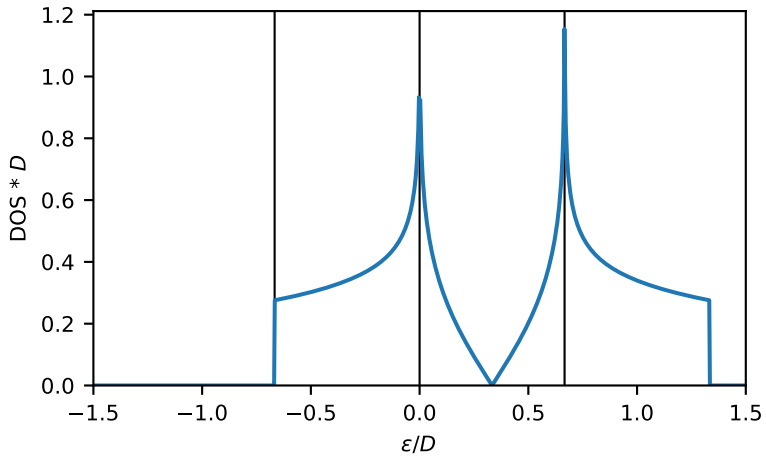
Examples

Calculated integrals

```
>>> from mpmath import mp
>>> mp.identify(mp.quad(gt.lattice.kagome.dos_mp, [-2/3, 0, 1/3, 2/3, 4/3]))
'(2/3)'
```

```
>>> eps = np.linspace(-1.5, 1.5, num=1001)
>>> dos_mp = [gt.lattice.kagome.dos(ee, half_bandwidth=1) for ee in eps]
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-2/3, 0, +2/3):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.kagome.gf_z

gftool.lattice.kagome.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D kagome lattice.

The Green's function of the 2D kagome lattice can be expressed in terms of the 2D triangular lattice *gftool.lattice.triangular.gf_z*, and a non-dispersive peak, see [kogan2021]. Omitting the non-dispersive peak, it corresponds to *gftool.lattice.honeycomb.gf_z* shifted by *half_bandwidth/3*.

The Green's function has singularities for *z/half_bandwidth in [-2/3, 0, 2/3]*.

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the kagome lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

complex np.ndarray or complex
Value of the kagome lattice Green's function.

See also:

gftool.lattice.triangular.gf_z

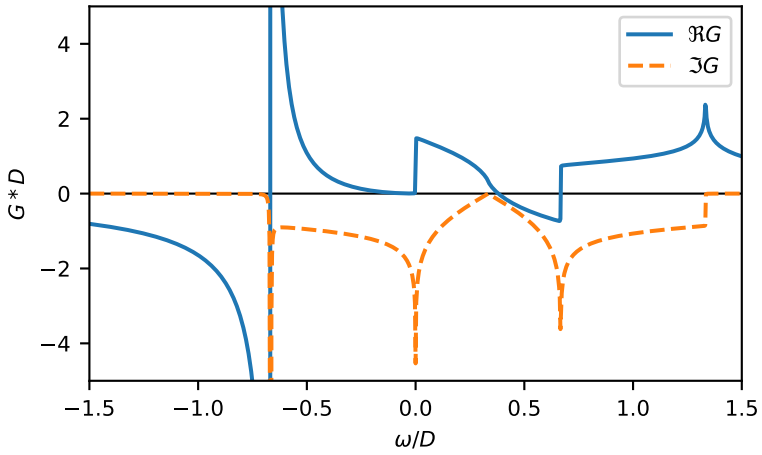
References

[varm2013], [kogan2021]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=1001, dtype=complex) + 1e-4j
>>> gf_ww = gt.lattice.kagome.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.ylim(bottom=-5, top=5)
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.kagome.hilbert_transform

gftool.lattice.kagome.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the kagome lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D kagome lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.kagome.gf_z

Notes

Relation between nearest neighbor hopping *t* and half-bandwidth *D*

$$3t/2 = D$$

gftool.lattice.sc

3D simple cubic (sc) lattice.

The dispersion of the 3D simple cubic lattice is given by

$$_{k_x,k_y,k_z} = 2t[\cos(k_x) + \cos(k_y) + \cos(k_z)]$$

which takes values in $_{k_x,k_y,k_z} \in [-6t, +6t] = [-D, +D]$.

half_bandwidth
The half_bandwidth corresponds to a nearest neighbor hopping of *t=D/6*

API

Functions

dos(eps[, half_bandwidth])	Local Green's function of 3D simple cubic lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the simple cubic DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 3D simple cubic lattice.
gf_z(z[, half_bandwidth])	Local Green's function of 3D simple cubic lattice.
gf_z_mp(z[, half_bandwidth])	Multi-precision Green's function of non-interacting 3D simple cubic lattice.
hilbert_transform(xi[, half_bandwidth])	Hilbert transform of non-interacting DOS of the simple cubic lattice.

gftool.lattice.sc.dos

gftool.lattice.sc.**dos**(*eps*, *half_bandwidth=1*)

Local Green's function of 3D simple cubic lattice.

Has a van Hove singularity (continuous but not differentiable) at *abs(eps) = D/3*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS of the simple cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/6$.

Returns

float np.ndarray or float
The value of the DOS.

Notes

Around `eps=0` the expansion Eq. (5.4) using Eq. (7.37) from [joyce1973] is used. Otherwise, it is identical to `-gf_z.imag/np.pi`

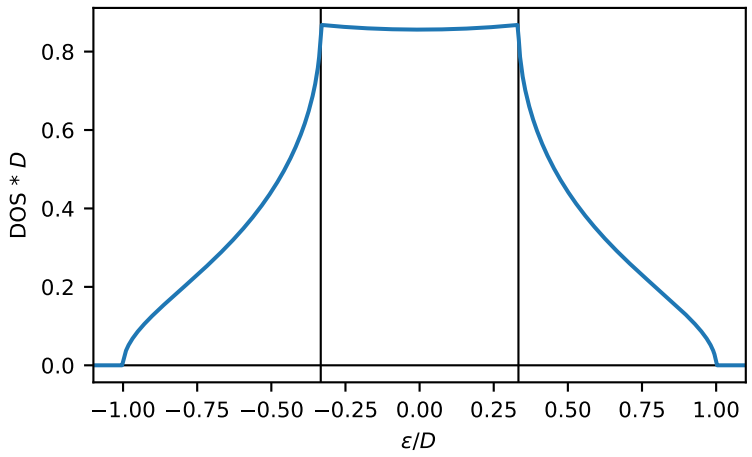
References

[economou2006], [joyce1973], [katsura1971]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=501)
>>> dos = gt.lattice.sc.dos(eps)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color="black", linewidth=0.8)
>>> _ = plt.axvline(-1/3, color="black", linewidth=0.8)
>>> _ = plt.axvline(+1/3, color="black", linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.sc.dos_moment

gftool.lattice.sc.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the simple cubic DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D simple cubic lattice.

Returns

float
The *m* th moment of the 3D simple cubic DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.sc.dos

gftool.lattice.sc.dos_mp

gftool.lattice.sc.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 3D simple cubic lattice.

Has a van Hove singularity (continuous but not differentiable) at *abs(eps) = D/3*.

Implements Eq. 7.37 from [joyce1973] for the special case of *eps = 0*, otherwise calls *gf_z_mp*.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS of the simple cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/6$.

Returns

mpmath.mpf
The value of the DOS.

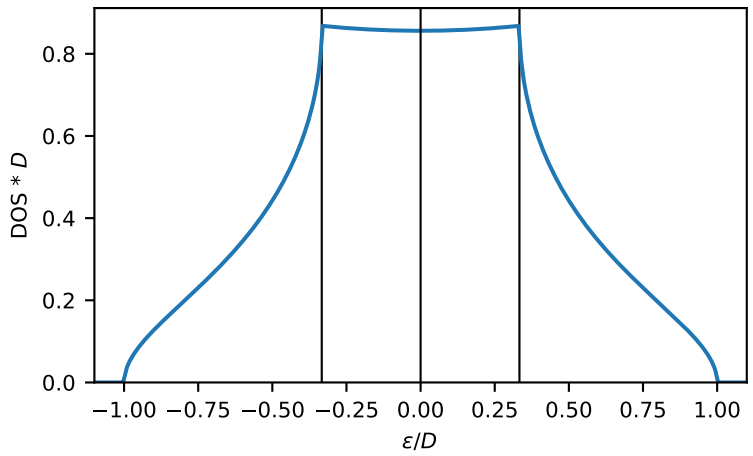
References

[economou2006], [joyce1973], [katsura1971]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=501)
>>> dos_mp = [gt.lattice.sc.dos_mp(ee, half_bandwidth=1) for ee in eps]
>>> dos_mp = np.array(dos_mp, dtype=np.float64)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(1/3, color="black", linewidth=0.8)
>>> _ = plt.axvline(-1/3, color="black", linewidth=0.8)
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color="black", linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.sc.gf_z

gftool.lattice.sc.**gf_z**(*z*, *half_bandwidth=1*)

Local Green's function of 3D simple cubic lattice.

Has a van Hove singularity (continuous but not differentiable) at $z = \pm D/3$.

Implements equations (1.24 - 1.26) from [delves2001].

Parameters

z

[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the simple cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/6$.

Returns

complex np.ndarray or complex
Value of the simple cubic Green's function at complex energy z.

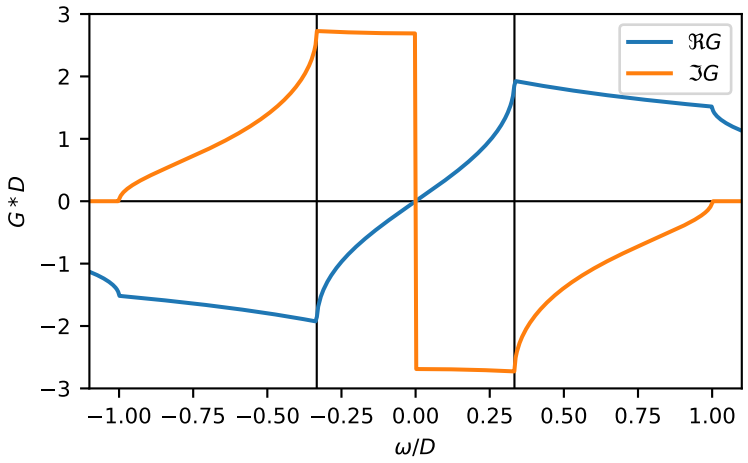
References

[economou2006], [delves2001]

Examples

```
>>> ww = np.linspace(-1.1, 1.1, num=500)
>>> gf_ww = gt.lattice.sc.gf_z(ww)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color="black", linewidth=0.8)
>>> _ = plt.axvline(-1/3, color="black", linewidth=0.8)
>>> _ = plt.axvline(+1/3, color="black", linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.sc.gf_z_mp

gftool.lattice.sc.**gf_z_mp**(*z*, *half_bandwidth=1*)

Multi-precision Green's function of non-interacting 3D simple cubic lattice.

Has a van Hove singularity (continuous but not differentiable) at $z = \pm D/3$.

Implements equations (1.24 - 1.26) from [delves2001].

Parameters

z
[mpmath.mpc or mpc_like] Green's function is evaluated at complex frequency z.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS of the simple cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/6$.

Returns

mpmath.mpc
Value of the Green's function at complex energy z.

References

[economou2006], [delves2001]

Examples

```
>>> ww = np.linspace(-1.1, 1.1, num=500)
>>> gf_ww = np.array([gt.lattice.sc.gf_z_mp(wi) for wi in ww])
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color="black", linewidth=0.8)
>>> _ = plt.axvline(-1/3, color="black", linewidth=0.8)
>>> _ = plt.axvline(+1/3, color="black", linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.astype(complex).real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.astype(complex).imag, label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```

gftool.lattice.sc.hilbert_transform

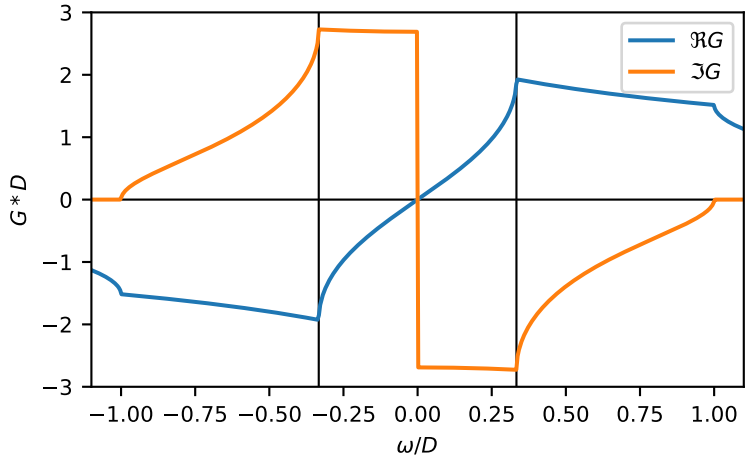
gftool.lattice.sc.**hilbert_transform**(*xi*, *half_bandwidth=1*)

Hilbert transform of non-interacting DOS of the simple cubic lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.



Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D simple cubic lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.sc.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$6t = D$$

gftool.lattice.bcc

3D body-centered cubic (bcc) lattice.

The dispersion of the 3D body-centered cubic lattice is given by

$$_{k_x,k_y,k_z} = 8t\cos(k_x)\cos(k_y)\cos(k_z)$$

which takes values in $_{k_x,k_y,k_z} \in [-8t, +8t] = [-D, +D]$.

half_bandwidth
The half_bandwidth corresponds to a nearest neighbor hopping of *t=D/8*

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 3D body-centered cubic lattice.
dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the body-centered cubic DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 3D body-centered lattice.
gf_z(z, half_bandwidth)	Local Green's function of 3D body-centered cubic (bcc) lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the body-centered cubic lattice.

gftool.lattice.bcc.dos

gftool.lattice.bcc.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 3D body-centered cubic lattice.

Has a van Hove singularity (logarithmic divergence) at *eps = 0*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.bcc.dos_mp
Multi-precision version suitable for integration.

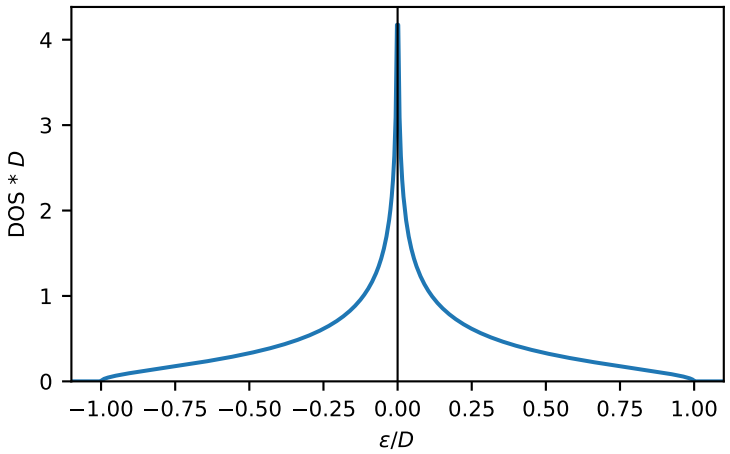
References

[morita1971]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.bcc.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.bcc.dos_moment

gftool.lattice.bcc.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the body-centered cubic DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D body-centered cubic lattice.

Returns

float
The *m* th moment of the 3D body-centered cubic DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments m.

See also:

gftool.lattice.bcc.dos

gftool.lattice.bcc.dos_mp

gftool.lattice.bcc.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 3D body-centered lattice.

Has a van Hove singularity (logarithmic divergence) at *eps = 0*.

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider using $\int dDOS()[f() - f(0)] + f(0)$ to avoid the singularity.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.bcc.dos
Vectorized version suitable for array evaluations.

References

[morita1971]

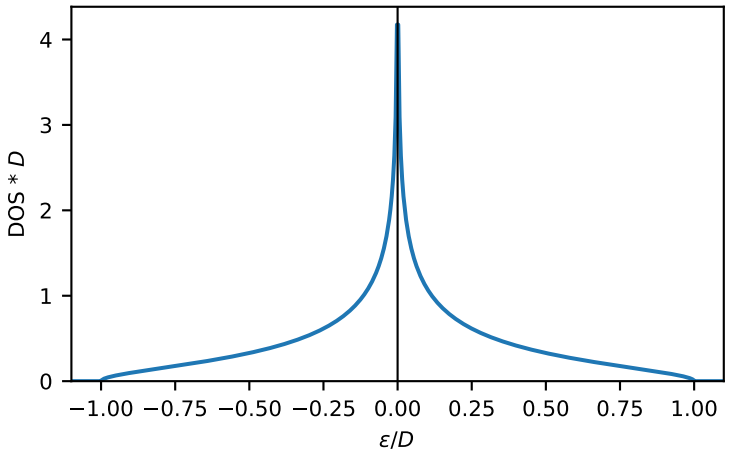
Examples

Calculate integrals:

```
>>> from mpmath import mp
>>> mp.quad(gt.lattice.bcc.dos_mp, [-1, 0, 1])
mpf('1.0')
```

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos_mp = [gt.lattice.bcc.dos_mp(ee, half_bandwidth=1) for ee in eps]
>>> dos_mp = np.array(dos_mp, dtype=np.float64)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.bcc.gf_z

gftool.lattice.bcc.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of 3D body-centered cubic (bcc) lattice.

Has a van Hove singularity at *z=0* (divergence).

Implements equations (2.1) and (2.4) from [morita1971]

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency *z*.

half_bandwidth
[float] Half-bandwidth of the DOS of the body-centered cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

complex np.ndarray or complex
Value of the body-centered cubic Green's function at complex energy *z*.

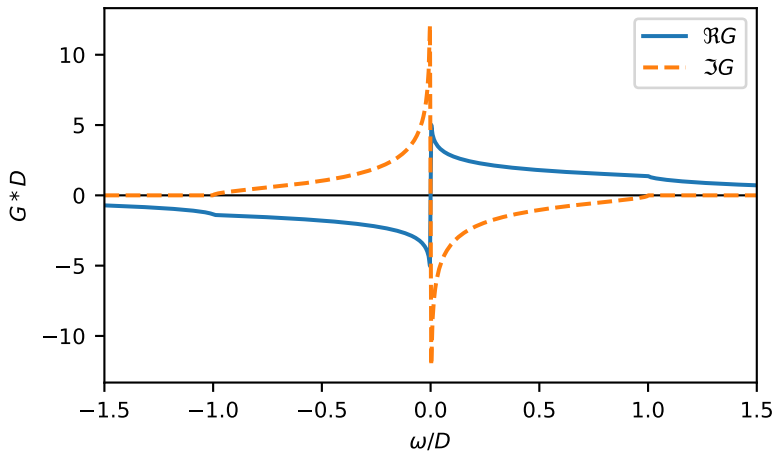
References

[morita1971]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.bcc.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.bcc.hilbert_transform

gftool.lattice.bcc.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the body-centered cubic lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi

[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D body-centered cubic lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.bcc.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$8t = D$$

gftool.lattice.fcc

3D face-centered cubic (fcc) lattice.

The dispersion of the 3D face-centered cubic lattice is given by

$$_{k_x,k_y,k_z} = 4t[\cos(k_x/2)\cos(k_y/2) + \cos(k_x/2)\cos(k_z/2) + \cos(k_y/2)\cos(k_z/2)]$$

which takes values in $_{k_x,k_y,k_z} \in [-4t, +12t] = [-0.5D, +1.5D]$.

half_bandwidth
The half_bandwidth corresponds to a nearest neighbor hopping of *t=D/8*.

API

Functions

dos(eps, half_bandwidth)	DOS of non-interacting 3D face-centered cubic lattice.
dos_moment(m, half_bandwidth)	Calculate the m th moment of the face-centered cubic DOS.
dos_mp(eps[, half_bandwidth])	Multi-precision DOS of non-interacting 3D face-centered cubic lattice.
gf_z(z, half_bandwidth)	Local Green's function of the 3D face-centered cubic (fcc) lattice.
hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the face-centered cubic lattice.

gftool.lattice.fcc.dos

gftool.lattice.fcc.**dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 3D face-centered cubic lattice.

Has a van Hove singularity at *z=-half_bandwidth/2* (divergence) and at *z=0* (continuous but not differentiable).

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(*eps* < -0.5*`half_bandwidth`) = 0, DOS(1.5*`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.fcc.dos_mp
Multi-precision version suitable for integration.

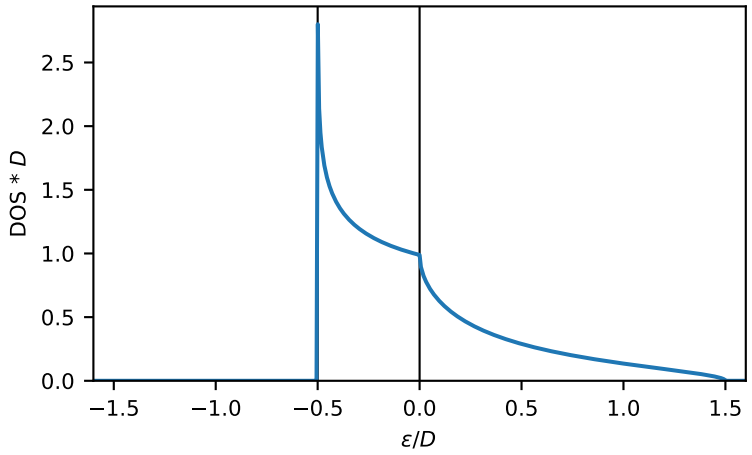
References

[morita1971]

Examples

```
>>> eps = np.linspace(-1.6, 1.6, num=501)
>>> dos = gt.lattice.fcc.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.axvline(-0.5, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.fcc.dos_moment

gftool.lattice.fcc.**dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the face-centered cubic DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D face-centered cubic lattice.

Returns

float
The *m* th moment of the 3D face-centered cubic DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.fcc.dos

gftool.lattice.fcc.dos_mp

gftool.lattice.fcc.**dos_mp**(*eps*, *half_bandwidth=1*)

Multi-precision DOS of non-interacting 3D face-centered cubic lattice.

Has a van Hove singularity at *z=-half_bandwidth/2* (divergence) and at *z=0* (continuous but not differentiable).

This function is particularity suited to calculate integrals of the form $\int dDOS() f()$. If you have problems with the convergence, consider using $\int dDOS()[f() - f(-1/2)] + f(-1/2)$ to avoid the singularity.

Parameters

eps
[mpmath.mpf or mpf_like] DOS is evaluated at points *eps*.

half_bandwidth
[mpmath.mpf or mpf_like] Half-bandwidth of the DOS, DOS(*eps* < -0.5*`half_bandwidth`) = 0, DOS(1.5*`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

mpmath.mpf
The value of the DOS.

See also:

gftool.lattice.fcc.dos
Vectorized version suitable for array evaluations.

References

[morita1971]

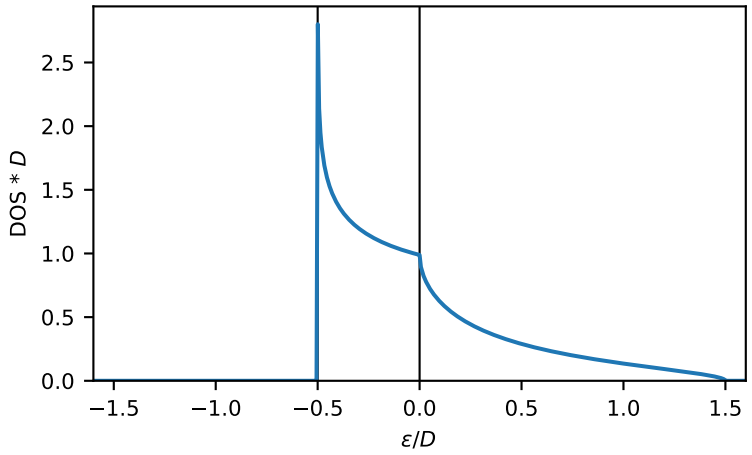
Examples

Calculate integrals:

```
>>> from mpmath import mp
>>> unit = mp.quad(gt.lattice.fcc.dos_mp, [-0.5, 0, 1.5])
>>> mp.identify(unit)
'1'
```

```
>>> eps = np.linspace(-1.6, 1.6, num=501)
>>> dos_mp = [gt.lattice.fcc.dos_mp(ee, half_bandwidth=1) for ee in eps]
>>> dos_mp = np.array(dos_mp, dtype=np.float64)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.axvline(-0.5, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos_mp)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



gftool.lattice.fcc.gf_z

gftool.lattice.fcc.**gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 3D face-centered cubic (fcc) lattice.

Note, that the spectrum is asymmetric and in $[-D/2, 3D/2]$, where D is the half-bandwidth.

Has a van Hove singularity at *z=-half_bandwidth/2* (divergence) and at *z=0* (continuous but not differentiable).

Implements equations (2.16), (2.17) and (2.11) from [morita1971].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency *z*.

half_bandwidth
[float] Half-bandwidth of the DOS of the face-centered cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

complex np.ndarray or complex
Value of the face-centered cubic lattice Green's function.

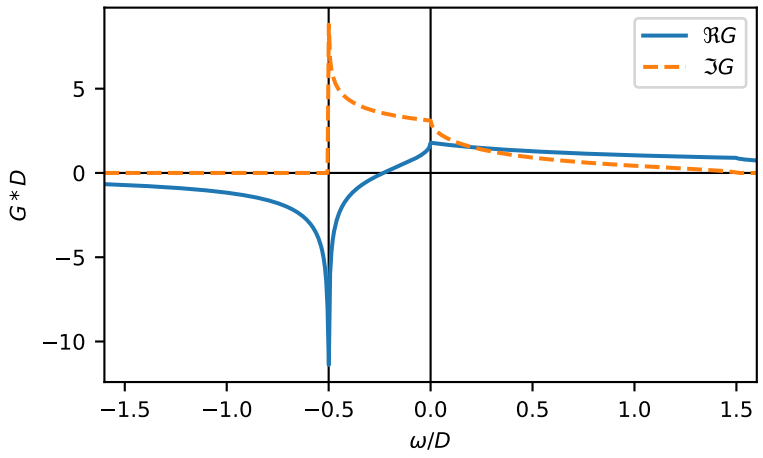
References

[morita1971]

Examples

```
>>> ww = np.linspace(-1.6, 1.6, num=501, dtype=complex)
>>> gf_ww = gt.lattice.fcc.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(-0.5, color='black', linewidth=0.8)
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



gftool.lattice.fcc.hilbert_transform

gftool.lattice.fcc.**hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the face-centered cubic lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
: [complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
: [float] Half-bandwidth of the DOS of the 3D face-centered cubic lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.fcc.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$8t = D$$

API

3.1.7 gftool.linalg

Collection of linear algebra algorithms not contained in `numpy` or `scipy`.

API

Functions

lstsq_ec(a, b, c, d[, rcond])	Least-squares solution with equality constraint for linear matrix eq.
orth_compl(mat)	Calculate the orthogonal complement of a rectangular matrix using QR.

gftool.linalg.lstsq_ec

gftool.linalg.**lstsq_ec**(*a*, *b*, *c*, *d*, *rcond=None*)

Least-squares solution with equality constraint for linear matrix eq.

Solves the equation $ax = b$ with the constraint $cx = d$, where the vector x minimizes the squared Euclidean 2-norm $||ax - b||_2^2$. Internally `numpy.linalg.lstsq` is used to solve the least-squares problem. The algorithm is taken from [golub2013].

Parameters

a
[(M, N) np.ndarray] "Coefficient" matrix.

b
[(M) np.ndarray] Ordinate or "dependent variable" values.

c
[(L, N) np.ndarray] "Coefficient" matrix of the constrains with $L < M$.

d
[(L) np.ndarray] Ordinate of the constrains with $L < M$.

rcond
[float, optional] Cut-off ratio for small singular values of a. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of *a* (default: machine precision times *max(M, N)*).

Returns

(N) np.ndarray Least-squares solution.

References

[golub2013]

gftool.linalg.orth_compl

`gftool.linalg.`**`orth_compl`**`(`*`mat`*`)`

Calculate the orthogonal complement of a rectangular matrix using QR.

For a tall N×M, N>M, matrix *mat* the complete QR-decomposition gives

$$A = QR = (Q_1, Q_2)(R_1, 0)^2.$$

The N×(N-M) matrix *Q_2* gives the orthogonal complement `A_⊥ = Q_2.T.conj()` with the property

$$A_\perp A = 0.$$

Parameters

mat
[(N, M) complex array_like] Tall input matrix.

Returns

(N-M, N) complex np.ndarray
Orthogonal complement of *mat*, such that `mat_perp@mat==0`.

Examples

```
>>> RNG = np.random.default_rng()
>>> mat = RNG.random((10, 5))
>>> mat_perp = gt.linalg.orth_compl(mat)
>>> np.allclose(mat_perp@mat, 0)
True
```

3.1.8 gftool.linearprediction

Linear prediction to extrapolated retarded Green's function.

A nice introductory description of linear prediction can be found in [Vaidyanathan2007]; [Makhoul1975] gives a detailed review.

Linear prediction allows to extend a time series by predicting future points x_l as a linear combination of previous time points x:

$$x_l = -\sum_{k=1}^{K} x_{l-k} a_k$$

where a_k are the prediction coefficients and K is the prediction order. **Recommended** method to obtain the prediction coefficients is *pcoeff_covar*, the results of *pcoeff_burg* seem to be rather **unreliable**.

References

Examples

We consider the retarded-time Bethe Green's function with the known time points

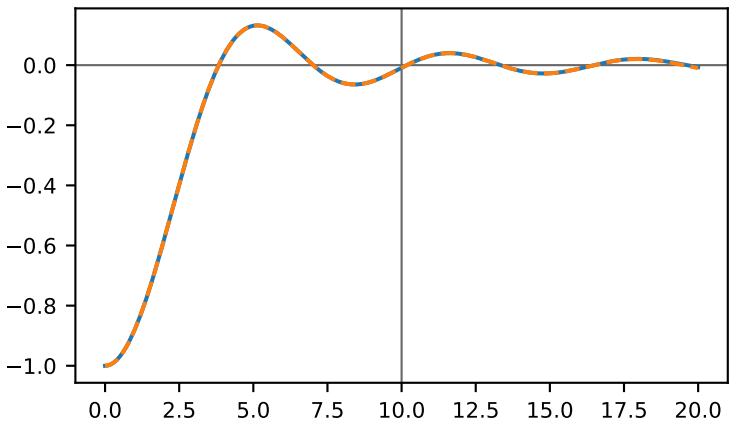
```
>>> tt = np.linspace(0, 10, 101)
>>> gf_t = gt.lattice.bethe.gf_ret_t(tt, half_bandwidth=1)
```

We can predict future time points using linear prediction, let's check the next 100 time points

```
>>> lp = gt.linearprediction
>>> pcoeff, __ = lp.pcoeff_covar(gf_t, order=gf_t.size//2)
>>> gf_pred = lp.predict(gf_t, pcoeff, num=100)
```

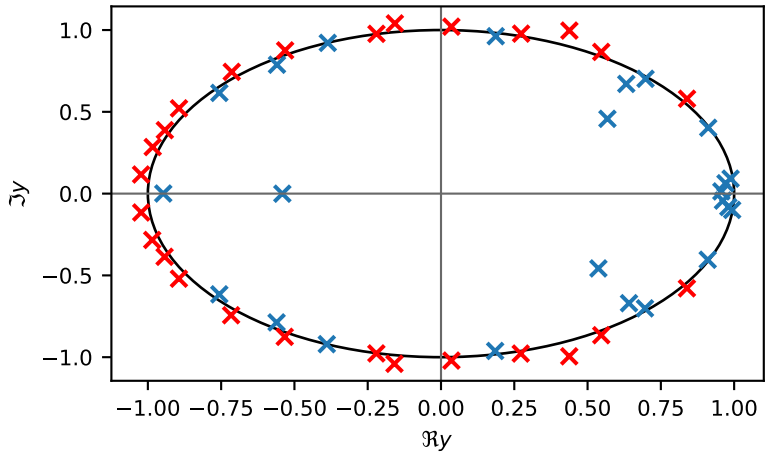
and compare the results

```
>>> import matplotlib.pyplot as plt
>>> tt_pred = np.linspace(0, 20, 201)
>>> gf_full = gt.lattice.bethe.gf_ret_t(tt_pred, half_bandwidth=1)
>>> __ = plt.axhline(0, color='dimgray', linewidth=0.8)
>>> __ = plt.axvline(tt[-1], color='dimgray', linewidth=0.8)
>>> __ = plt.plot(tt_pred, gf_full.imag)
>>> __ = plt.plot(tt_pred, gf_pred.imag, '--')
>>> plt.show()
```



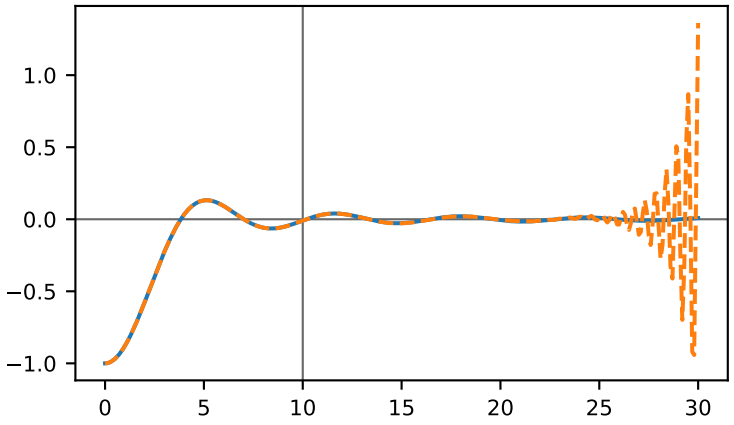
The roots corresponding to the linear prediction polynomial should all lie in the unit circle, numerical inaccuracies can lead to roots outside the unit circle causing exponentially growing contributions. For example, if we add some noise:

```
>>> noise = np.random.default_rng(0).normal(scale=1e-6, size=tt.size)
>>> pcoeff, __ = lp.pcoeff_covar(gf_t + noise, order=gf_t.size//2)
>>> __ = lp.plot_roots(pcoeff)
```



The red crosses correspond to growing contributions. Prediction for long times produces exponentially growing errors:

```
>>> import matplotlib.pyplot as plt
>>> tt_pred = np.linspace(0, 30, 301)
>>> gf_full = gt.lattice.bethe.gf_ret_t(tt_pred, half_bandwidth=1)
>>> gf_pred = lp.predict(gf_t, pcoeff, num=200)
>>> __ = plt.axhline(0, color='dimgray', linewidth=0.8)
>>> __ = plt.axvline(tt[-1], color='dimgray', linewidth=0.8)
>>> __ = plt.plot(tt_pred, gf_full.imag)
>>> __ = plt.plot(tt_pred, gf_pred.imag, '--')
>>> plt.show()
```



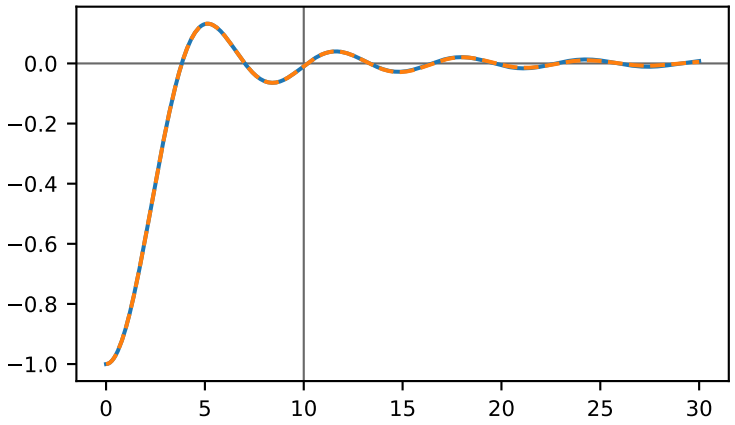
This can be amended by setting *stable=True* in `predict`:

```
>>> import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
>>> tt_pred = np.linspace(0, 30, 301)
>>> gf_full = gt.lattice.bethe.gf_ret_t(tt_pred, half_bandwidth=1)
>>> gf_pred = lp.predict(gf_t, pcoeff, num=200, stable=True)
>>> __ = plt.axhline(0, color='dimgray', linewidth=0.8)
>>> __ = plt.axvline(tt[-1], color='dimgray', linewidth=0.8)
>>> __ = plt.plot(tt_pred, gf_full.imag)
>>> __ = plt.plot(tt_pred, gf_pred.imag, '--')
>>> plt.show()
```



API

Functions

companion(a)	Create a companion matrix.
pcoeff_burg(x, order)	Burg's method for linear prediction (LP) coefficients.
pcoeff_covar(x, order[, rcond])	Calculate linear prediction (LP) coefficients using covariance method.
plot_roots(pcoeff[, axis])	Plot the roots corresponding to *pcoeff*.
predict(x, pcoeff, num[, stable])	Forward-predict a series additional *num* steps.

gftool.linearprediction.companion

gftool.linearprediction.**companion**(*a*)

Create a companion matrix.

Create the companion matrix [1] associated with the polynomial whose coefficients are given in *a*.

Parameters

a

[(…, N) array_like] 1-D array of polynomial coefficients. The length of *a* must be at least two, and `a[0]` must not be zero.

Returns

(…, N-1, N-1) ndarray
The first row of *c* is `-a[1:]/a[0]`, and the first sub-diagonal is all ones. The data-type of the array is the same as the data-type of `1.0*a[0]`.

Raises

ValueError
If any of the following are true: a) `a.size < 2`; b) `a[0] == 0`.

Notes

Modified version of SciPy, contribute it back

References

[1]

gftool.linearprediction.pcoeff_burg

`gftool.linearprediction.`**`pcoeff_burg`**(*x*, *order: int*)

Burg's method for linear prediction (LP) coefficients.

> **Warning:** We found this method to be instable, consider using *`pcoeff_covar`* instead.

Burg's method calculates the coefficients from an estimate of the reflection coefficients using Levinson's method. Burg's method guarantees that poles are inside the unit circle [Kay1988], thus it is stable.

Parameters

x
[(…, N) complex np.ndarray] Data of the (time) series to be predicted.

order
[int] Prediction order, has to be smaller then *N*.

Returns

a
[(…, order) complex np.ndarray] Prediction coefficients.

rho
[(…) float np.ndarray] Variance estimate.

See also:

`pcoeff_covar`

References

[Kay1988]

gftool.linearprediction.pcoeff_covar

gftool.linearprediction.**pcoeff_covar**(*x*, *order: int*, *rcond=None*)

Calculate linear prediction (LP) coefficients using covariance method.

The covariance method gives the equation

$$Ra = X^\dagger Xa = X^\dagger x = -r$$

where R is the covariance matrix and a are the LP coefficients. We solve $Xa = x$ using linear least-squares.

Parameters

x
[(…, N) complex np.ndarray] Data of the (time) series to be predicted.

order
[int] Prediction order, has to be smaller then *N*; for `order>N//2` the system is underdetermined.

rcond
[float, optional] Cut-off ratio for small singular values of a. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of a.

Returns

a
[(…, order) complex np.ndarray] Prediction coefficients.

rho
[(…) float np.ndarray] Error estimate $\|x - Xa\|_2$.

Raises

ValueError
If the prediction *order* is not smaller than the number of data points *N*.

gftool.linearprediction.plot_roots

gftool.linearprediction.**plot_roots**(*pcoeff*, *axis=None*)

Plot the roots corresponding to *pcoeff*.

Roots for the forward prediction should be inside the unit-circle.

Parameters

pcoeff
[(order,) complex np.ndarray] Prediction coefficients.

axis
[matplotlib.axes.Axes , optional] Axis in which the roots are plotted (default: `plt.gca()`).

Returns

matplotlib.axes.Axes
The *axis* in which the roots were plotted.

See also:

pcoeff_covar

gftool.linearprediction.predict

gftool.linearprediction.**predict**(*x*, *pcoeff*, *num: int*, *stable=False*)

Forward-predict a series additional *num* steps.

Parameters

x
[(…, N) complex np.ndarray] Data of the (time) series to be predicted.

pcoeff
[(…, order) complex np.ndarray] Prediction coefficients.

num
[int] Number of additional (time) steps.

stable
[bool, optional] If *stable* exponentially growing terms are suppressed, by setting roots outside the unit-circle to zero (default: False).

Returns

(…, N+num) complex np.ndarray
Data of the (time) series extended by *num* steps, with *px[:x.size] = x*.

Raises

ValueError
If the number of additional steps *num* is negative.

See also:

pcoeff_covar

3.1.9 gftool.matrix

Functions to work with Green's functions in matrix from.

A main use case of this library is the calculation of the Green's function as the resolvent of the Hermitian or from Dyson equation. Instead of calculating the inverse for every frequency/k-point it is oftentimes more efficient, to calculate an eigendecomposition once.

For example, let us calculate the Green's function for 1D tight-binding chain using:

$$G(z) = [1z - H]^{-1} = [1z - UU^\dagger]^{-1} = U[z -]^{-1}U^\dagger$$

```
>>> N = 51  # system size
>>> t = 1   # hopping amplitude
>>> hamilton = np.zeros((N, N))
>>> row, col = np.diag_indices(N)
>>> hamilton[row[:-1], col[1:]] = hamilton[row[1:], col[:-1]] = -t
```

```
>>> ww = np.linspace(-2.5, 2.5, num=201) + 1e-1j
>>> dec = gt.matrix.decompose_her(hamilton)
>>> gf_ww = dec.reconstruct(eig=1.0/(ww[:, np.newaxis] - dec.eig))
```

Let's check that it agrees with the inversion:

```
>>> gf_inv = np.linalg.inv(np.eye(N)*ww[0] - hamilton)
>>> np.allclose(gf_ww[0], gf_inv)
True
```

If we only need the diagonal (local) elements, we can calculate them using:

```
>>> gf_ww = dec.reconstruct(eig=1.0/(ww[:, np.newaxis] - dec.eig), kind='diag')
```

Recommended functions

- *decompose_mat* to create *Decomposition* of general matrices
- *decompose_sym* to create *Decomposition* of complex symmetric matrices
- *decompose_her* to create *UDecomposition* of Hermitian matrices

Rest are mostly legacy functions.

API

Functions

construct_gf(rv, diag_inv, rv_inv)	Construct Green's function from decomposition of its inverse.
decompose_gf(g_inv)	Decompose the inverse Green's function into eigenvalues and eigenvectors.
decompose_hamiltonian(hamilton)	Decompose the Hamiltonian matrix into eigenvalues and eigenvectors.
decompose_her(her_mat[, check])	Decompose Hermitian matrix *her_mat* into eigenvalues and (right) eigenvectors.
decompose_mat(mat)	Decompose matrix *mat* into eigenvalues and (right) eigenvectors.
decompose_sym(sym_mat[, check])	Decompose symmetric matrix *sym_mat* into eigenvalues and (right) eigenvectors.
gf_2x2_z(z, eps0, eps1, hopping[, hilbert_trafo])	Calculate the diagonal Green's function elements for a 2x2 system.

gftool.matrix.construct_gf

gftool.matrix.**construct_gf**(*rv*, *diag_inv*, *rv_inv*)

Construct Green's function from decomposition of its inverse.

$$G^{-1} = PhP^{-1} \Rightarrow G = Ph^{-1}P^{-1}$$

It is recommended to directly use `Decomposition.reconstruct` instead.

Parameters

rv
[(N, N) complex np.ndarray] The matrix of right eigenvectors (P).

diag_inv
[(N) array_like] The eigenvalues (h).

rv_inv
[(N, N) complex np.ndarray] The inverse of the matrix of right eigenvectors (P^{-1}).

Returns

(N, N) complex np.ndarray
The Green's function.

gftool.matrix.decompose_gf

gftool.matrix.**decompose_gf**(*g_inv*) → *Decomposition*

Decompose the inverse Green's function into eigenvalues and eigenvectors.

Deprecated since version 0.10.0: Use the function `decompose_mat` or `decompose_sym` instead.

The similarity transformation:

$$G^{-1} = PgP^{-1}, \quad g = diag(\imath)$$

Parameters

g_inv
[(..., N, N) complex np.ndarray] Matrix to be decomposed.

Returns

Decomposition.rv
[(..., N, N) complex np.ndarray] The right eigenvectors P.

Decomposition.h
[(..., N) complex np.ndarray] The complex eigenvalues of *g_inv*.

Decomposition.rv_inv
[(..., N, N) complex np.ndarray] The *inverse* of the right eigenvectors P.

gftool.matrix.decompose_hamiltonian

gftool.matrix.**decompose_hamiltonian**(*hamilton*) → *UDecomposition*

Decompose the Hamiltonian matrix into eigenvalues and eigenvectors.

Deprecated since version 0.10.0: Use the function `decompose_her`.

The similarity transformation:

$$H = UhU^\dagger, \quad h = diag(\imath)$$

Parameters

hamilton
[(…, N, N) complex np.ndarray] Hermitian matrix to be decomposed.

Returns

Decomposition.rv
[(…, N, N) complex np.ndarray] The right eigenvectors U.

Decomposition.h
[(…, N) float np.ndarray] The eigenvalues of *hamilton*.

Decomposition.rv_inv
[(…, N, N) complex np.ndarray] The *inverse* of the right eigenvectors $U^\dagger$. The Hamiltonian is hermitian, thus the decomposition is unitary $U^\dagger = U^{-1}$.

gftool.matrix.decompose_her

gftool.matrix.**decompose_her**(*her_mat*, *check=True*) → *UDecomposition*

Decompose Hermitian matrix *her_mat* into eigenvalues and (right) eigenvectors.

Decompose the *her_mat* into *rv, eig, rv_inv*, with *her_mat = (rv * eig) @ rv_inv*. This is the unitary similarity transformation:

$$M = UU^\dagger, = diag(_0, _1, \dots)$$

where are the eigenvalues and U the unitary matrix of right eigenvectors returned as *rv* with $U^{-1} = U^\dagger$. Internally, this is just a wrapper for `numpy.linalg.eigh`.

Parameters

her_mat
[(…, N, N) complex np.ndarray] Matrix to be decomposed.

check
[bool, optional] If *check*, raise an error if *her_mat* is not Hermitian (default: True).

Returns

Decomposition.rv
[(…, N, N) complex np.ndarray] The right eigenvectors U.

Decomposition.eig
[(…, N) complex np.ndarray] The complex eigenvalues of *her_mat*.

Decomposition.rv_inv
[(…, N, N) complex np.ndarray] The *inverse* of the right eigenvectors U.

Raises

ValueError
If *check=True* and *her_mat* is not Hermitian.

Examples

Perform the eigendecomposition:

```
>>> matrix = np.random.random((10, 10)) + 1j*np.random.random((10, 10))
>>> her_mat = 0.5*(matrix + matrix.conj().T)
>>> rv, eig, rv_inv = gt.matrix.decompose_her(her_mat)
>>> np.allclose(her_mat, (rv * eig) @ rv_inv)
True
>>> np.allclose(rv @ rv.conj().T, np.eye(*her_mat.shape))
True
```

This can also be simplified using the *Decomposition* class

```
>>> dec = gt.matrix.decompose_her(her_mat)
>>> np.allclose(her_mat, dec.reconstruct())
True
```

gftool.matrix.decompose_mat

gftool.matrix.**decompose_mat**(*mat*) → *Decomposition*

Decompose matrix *mat* into eigenvalues and (right) eigenvectors.

Decompose the *mat* into *rv, eig, rv_inv*, with *mat = (rv * eig) @ rv_inv*. This is the similarity transformation:

$$M = PP^{-1}, \ = diag(_0, _1, \ldots)$$

where are the eigenvalues and P the matrix of right eigenvectors returned as *rv*. Internally, this is just a wrapper for numpy.linalg.eig.

Parameters

mat
[(…, N, N) complex np.ndarray] Matrix to be decomposed.

Returns

Decomposition.rv
[(…, N, N) complex np.ndarray] The right eigenvectors P.

Decomposition.eig
[(…, N) complex np.ndarray] The complex eigenvalues of *mat*.

Decomposition.rv_inv
[(…, N, N) complex np.ndarray] The *inverse* of the right eigenvectors P.

Examples

Perform the eigendecomposition:

```
>>> matrix = np.random.random((10, 10))
>>> rv, eig, rv_inv = gt.matrix.decompose_mat(matrix)
>>> np.allclose(matrix, (rv * eig) @ rv_inv)
True
>>> np.allclose(rv @ rv_inv, np.eye(*matrix.shape))
True
```

This can also be simplified using the *Decomposition* class

```
>>> dec = gt.matrix.decompose_mat(matrix)
>>> np.allclose(matrix, dec.reconstruct())
True
```

gftool.matrix.decompose_sym

gftool.matrix.**decompose_sym**(*sym_mat*, *check=True*) → *Decomposition*

Decompose symmetric matrix *sym_mat* into eigenvalues and (right) eigenvectors.

Decompose the *sym_mat* into *rv, eig, rv_inv*, with *sym_mat = (rv * eig) @ rv_inv*. This is the *almost orthogonal* similarity transformation:

$$M = OO^T, \ = diag(_0, _1, \ldots)$$

where are the eigenvalues and U the unitary matrix of right eigenvectors returned as *rv* with $O^{-1} \approx O^T$. Internally, this is just a wrapper for `numpy.linalg.eig`. As mentioned the transformation is only *almost orthogonal*, so you should not rely on this fact! Still, *decompose_sym* should be better conditioned than *decompose_mat* so it is preferable (so slightly slower).

If you require orthogonality consider using [noble2017], it should also be faster.

Parameters

sym_mat
[(…, N, N) complex np.ndarray] Matrix to be decomposed.

check
[bool, optional] If *check*, raise an error if *sym_mat* is not symmetric (default: True).

Returns

Decomposition.rv
[(…, N, N) complex np.ndarray] The right eigenvectors O.

Decomposition.eig
[(…, N) complex np.ndarray] The complex eigenvalues of *sym_mat*.

Decomposition.rv_inv
[(…, N, N) complex np.ndarray] The *inverse* of the right eigenvectors O.

Raises

ValueError
If *check=True* and *sym_mat* is not symmetric.

References

[noble2017]

Examples

Perform the eigendecomposition:

```
>>> matrix = np.random.random((10, 10)) + 1j*np.random.random((10, 10))
>>> sym_mat = 0.5*(matrix + matrix.T)
>>> rv, eig, rv_inv = gt.matrix.decompose_sym(sym_mat)
>>> np.allclose(sym_mat, (rv * eig) @ rv_inv)
True
```

The result should be almost orthogonal, but *do not* rely on it!

```
>>> np.allclose(np.linalg.inv(rv), rv.T)
True
```

This can also be simplified using the *Decomposition* class

```
>>> dec = gt.matrix.decompose_sym(sym_mat)
>>> np.allclose(sym_mat, dec.reconstruct())
True
```

gftool.matrix.gf_2x2_z

gftool.matrix.**gf_2x2_z**(*z*, *eps0*, *eps1*, *hopping*, *hilbert_trafo=None*)

Calculate the diagonal Green's function elements for a 2x2 system.

Parameters

z
: [(…) complex array_like] Complex frequencies.

eps0, eps1
: [(…) float or complex array_like] On-site energy of element 0 and 1. For interacting systems this can be replaced by on-site energy + self-energy.

hopping
: [(…) float or complex array_like] Hopping element between element 0 and 1.

hilbert_trafo
: [Callable, optional] Hilbert transformation. If given, return the local Green's function. Else the lattice dispersion $_k$ can be given via $z \to z$ - ϵ_k.

Returns

(…, 2) complex array_like
: Diagonal elements of the Green's function of the 2x2 system.

Notes

For the trivial case *eps0==eps1 and hopping==0*, this implementation fails.

Classes

Decomposition(rv, eig, rv_inv)	Decomposition of a matrix into eigenvalues and eigenvectors.
UDecomposition(rv, eig, rv_inv)	Unitary decomposition of a matrix into eigenvalues and eigenvectors.

gftool.matrix.Decomposition

class gftool.matrix.**Decomposition**(*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*)

Decomposition of a matrix into eigenvalues and eigenvectors.

$$M = PP^{-1}, = diag(_0, _1, \ldots)$$

This class holds the eigenvalues and eigenvectors of the decomposition of a matrix and offers methods to reconstruct it. One intended use case is to use the *Decomposition* for the inversion of the Green's function to calculate it from the resolvent.

The order of the attributes is always *rv, eig, rv_inv*, as this gives the reconstruct of the matrix: *mat = (rv * eig) @ rv_inv*

Parameters

rv
[(…, N, N) complex np.ndarray] The matrix of right eigenvectors.

eig
[(…, N) complex np.ndarray] The vector of eigenvalues.

rv_inv
[(…, N, N) complex np.ndarray] The inverse of *rv*.

Examples

Perform the eigendecomposition:

```
>>> matrix = np.random.random((10, 10))
>>> dec = gt.matrix.decompose_mat(matrix)
>>> np.allclose(matrix, dec.reconstruct())
True
```

Inversion of matrix

```
>>> matrix_inv = dec.reconstruct(eig=1.0/dec.eig)
>>> np.allclose(np.linalg.inv(matrix), matrix_inv)
True
```

__init__(*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*) → None

Methods

__init__(rv, eig, rv_inv)	

count(value)	

from_gf(gf)	Decompose the inverse Green's function matrix.
from_hamiltonian(hamilton)	Decompose the Hamiltonian matrix.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
reconstruct([eig, kind])	Get matrix back from *Decomposition*.

gftool.matrix.Decomposition.__init__

Decomposition.**__init__**(*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*) → None

gftool.matrix.Decomposition.count

Decomposition.**count**(*value*) → integer -- return number of occurrences of value

gftool.matrix.Decomposition.from_gf

classmethod Decomposition.**from_gf**(*gf*) → *Decomposition*

Decompose the inverse Green's function matrix.

The similarity transformation:

$$G^{-1} = PgP^{-1}, \quad g = diag(_l)$$

Parameters

gf
[(…, N, N) complex np.ndarray] Matrix to be decomposed.

Returns

Decomposition

gftool.matrix.Decomposition.from_hamiltonian

classmethod Decomposition.**from_hamiltonian**(*hamilton*)

Decompose the Hamiltonian matrix.

The similarity transformation:

$$H = UhU^\dagger, \quad h = diag(_l)$$

Parameters

hamilton
[(…, N, N) complex np.ndarray] Hermitian matrix to be decomposed.

Returns

Decomposition

gftool.matrix.Decomposition.index

Decomposition.**index**(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

gftool.matrix.Decomposition.reconstruct

Decomposition.**reconstruct**(*eig=None*, *kind='full'*)

Get matrix back from *Decomposition*.

If the reciprocal of *self.eig* was taken, this corresponds to the inverse of the original matrix.

Parameters

eig
[(…, N) np.ndarray, optional] Alternative value used for *self.eig*. This argument can be used instead of modifying *self.eig*.

kind
[{'diag', 'full'} or str] Defines how to reconstruct the matrix. If *kind* is 'diag', only the diagonal elements are computed, if it is 'full' the complete matrix is returned. Alternatively a *str* used for subscript of `numpy.einsum` can be given.

Returns

(…, N, N) or (…, N) np.ndarray
The reconstructed matrix. If a subscript string is given as *kind*, the shape of the output might differ.

Attributes

rv	The matrix of right eigenvectors.
eig	The vector of eigenvalues.
rv_inv	The inverse of *rv*.

gftool.matrix.Decomposition.rv

Decomposition.**rv**: **ndarray**

The matrix of right eigenvectors.

gftool.matrix.Decomposition.eig

Decomposition.**eig**: **ndarray**
The vector of eigenvalues.

gftool.matrix.Decomposition.rv_inv

Decomposition.**rv_inv**: **ndarray**
The inverse of *rv*.

gftool.matrix.UDecomposition

class gftool.matrix.**UDecomposition**(*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*)
Unitary decomposition of a matrix into eigenvalues and eigenvectors.

$$H = UU^\dagger, = diag()$$

This class holds the eigenvalues and eigenvectors of the decomposition of a matrix and offers methods to reconstruct it. One intended use case is to use the *UDecomposition* for the inversion of the Green's function to calculate it from the resolvent.

The order of the attributes is always *rv, eig, rv_inv*, as this gives the reconstruct of the matrix: *mat = (rv * eig) @ rv_inv*

Parameters

rv
[(…, N, N) complex np.ndarray] The matrix of right eigenvectors.

eig
[(…, N) float np.ndarray] The vector of real eigenvalues.

rv_inv
[(…, N, N) complex np.ndarray] The inverse of *rv*.

Examples

Perform the eigendecomposition:

```
>>> matrix = np.random.random((10, 10)) + 1j*np.random.random((10, 10))
>>> matrix = 0.5*(matrix + matrix.conj().T)
>>> dec = gt.matrix.decompose_her(matrix)
>>> np.allclose(matrix, dec.reconstruct())
True
```

Inversion of matrix

```
>>> matrix_inv = dec.reconstruct(eig=1.0/dec.eig)
>>> np.allclose(np.linalg.inv(matrix), matrix_inv)
True
```

The similarity transformation is unitary:

```
>>> np.allclose(dec.u.conj().T, dec.uh)
True
>>> np.allclose(dec.u @ dec.u.conj().T, np.eye(*matrix.shape))
True
```

Attributes

u
Unitary matrix of right eigenvectors, same as *rv*.

uh
Hermitian conjugate of unitary matrix *rv*, same as *rv_inv*.

s
Singular values in descending order, different from order of *eig*.

__init__ (*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*) → None

Methods

__init__(rv, eig, rv_inv)

count(value)

from_gf(gf)	Decompose the inverse Green's function matrix.
from_hamiltonian(hamilton)	Decompose the Hamiltonian matrix.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
reconstruct([eig, kind])	Get matrix back from *Decomposition*.

gftool.matrix.UDecomposition.__init__

UDecomposition.**__init__** (*rv: ndarray*, *eig: ndarray*, *rv_inv: ndarray*) → None

gftool.matrix.UDecomposition.count

UDecomposition.**count** (*value*) → integer -- return number of occurrences of value

gftool.matrix.UDecomposition.from_gf

classmethod UDecomposition.**from_gf** (*gf*) → *Decomposition*

Decompose the inverse Green's function matrix.

The similarity transformation:

$$G^{-1} = PgP^{-1}, \quad g = diag(_l)$$

Parameters

gf
[(…, N, N) complex np.ndarray] Matrix to be decomposed.

Returns

Decomposition

gftool.matrix.UDecomposition.from_hamiltonian

classmethod UDecomposition.**from_hamiltonian**(*hamilton*)

Decompose the Hamiltonian matrix.

The similarity transformation:

$$H = UhU^\dagger, \quad h = diag(\iota)$$

Parameters

hamilton
[(…, N, N) complex np.ndarray] Hermitian matrix to be decomposed.

Returns

Decomposition

gftool.matrix.UDecomposition.index

UDecomposition.**index**(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

gftool.matrix.UDecomposition.reconstruct

UDecomposition.**reconstruct**(*eig=None*, *kind='full'*)

Get matrix back from *Decomposition*.

If the reciprocal of *self.eig* was taken, this corresponds to the inverse of the original matrix.

Parameters

eig
[(…, N) np.ndarray, optional] Alternative value used for *self.eig*. This argument can be used instead of modifying *self.eig*.

kind
[{'diag', 'full'} or str] Defines how to reconstruct the matrix. If *kind* is 'diag', only the diagonal elements are computed, if it is 'full' the complete matrix is returned. Alternatively a *str* used for subscript of `numpy.einsum` can be given.

Returns

(…, N, N) or (…, N) np.ndarray
The reconstructed matrix. If a subscript string is given as *kind*, the shape of the output might differ.

Attributes

eig	The vector of eigenvalues.
rv	The matrix of right eigenvectors.
rv_inv	The inverse of *rv*.
s	Singular values in descending order, different from order of *eig*.
u	Unitary matrix of right eigenvectors, same as *rv*.
uh	Hermitian conjugate of unitary matrix *rv*, same as *rv_inv*.

gftool.matrix.UDecomposition.eig

UDecomposition.**eig**: **ndarray**

The vector of eigenvalues.

gftool.matrix.UDecomposition.rv

UDecomposition.**rv**: **ndarray**

The matrix of right eigenvectors.

gftool.matrix.UDecomposition.rv_inv

UDecomposition.**rv_inv**: **ndarray**

The inverse of *rv*.

gftool.matrix.UDecomposition.s

property UDecomposition.**s**

Singular values in descending order, different from order of *eig*.

gftool.matrix.UDecomposition.u

property UDecomposition.**u**

Unitary matrix of right eigenvectors, same as *rv*.

gftool.matrix.UDecomposition.uh

property UDecomposition.**uh**

Hermitian conjugate of unitary matrix *rv*, same as *rv_inv*.

3.1.10 gftool.pade

Padé analytic continuation for Green's functions and self-energies.

The main aim of this module is to provide analytic continuation based on averaging over multiple Padé approximants (similar to [1]).

In most cases the following high level function should be used:

averaged*, *avg_no_neg_imag
Return one-shot analytic continuation evaluated at z.

Averager
Returns a function for repeated evaluation of the continued function.

References

API

Functions

Averager(z_in, coeff, *, valid_pades, kind)	Create function for averaging Padé scheme.
FilterHighVariance([rel_num, abs_num])	Return function to filter continuations with highest variance.
FilterNegImag([threshold])	Return function to check if imaginary part is smaller than *threshold*.
FilterNegImagNum([abs_num, rel_num])	Return function to check how bad the imaginary part gets.
Mod_Averager(z_in, coeff, mod_fct, *, ...[, ...])	Create function for averaging Padé scheme using *mod_fct* before the average.
apply_filter(*filters, validity_iter)	Handle usage of filters for Padé.
averaged(z_out, z_in, *[, valid_z, fct_z, ...])	Return the averaged Padé continuation with its variance.
avg_no_neg_imag(z_out, z_in, *[, valid_z, ...])	Average Padé filtering approximants with non-negative imaginary part.
calc_iterator(z_out, z_in, coeff)	Calculate Padé continuation of function at points *z_out*.
coefficients(z, fct_z)	Calculate the coefficients for the Padé continuation.

gftool.pade.Averager

gftool.pade.**Averager**(*z_in*, *coeff*, *, *valid_pades*, *kind:* KindSelector)
Create function for averaging Padé scheme.

Parameters

z_in
[(N_in,) complex ndarray] Complex mesh used to calculate *coeff*.

coeff
[(…, N_in) complex ndarray] Coefficients for Padé, calculated from *pade.coefficients*.

valid_pades
[list_like of bool] Mask which continuations are correct, all Padés where *valid_pades* evaluates to false will be ignored for the average.

kind
[{KindGf, KindSelf}] Defines the asymptotic of the continued function and the number of minimum and maximum input points used for Padé. For `KindGf` the function goes like $1/z$ for large z, for `KindSelf` the function behaves like a constant for large z.

Returns

function
The continued function *f(z)* (*z*,) -> Result. *f(z).x* contains the function values *f(z).err* the associated variance.

Raises

TypeError
If *valid_pades* not of type *bool*

RuntimeError
If all there are none elements of *valid_pades* that evaluate to True.

gftool.pade.FilterHighVariance

gftool.pade.**FilterHighVariance**(*rel_num: Optional[float] = None*, *abs_num: Optional[int] = None*)

Return function to filter continuations with highest variance.

Parameters

rel_num
[float, optional] The relative number of continuations to keep.

abs_num
[int, optional] The absolute number of continuations to keep.

Returns

callable
The filter function (pade_iter) -> np.ndarray.

gftool.pade.FilterNegImag

gftool.pade.**FilterNegImag**(*threshold=1e-08*)

Return function to check if imaginary part is smaller than *threshold*.

This methods is designed to create *valid_pades* for `Averager`. The imaginary part of retarded Green's functions and self-energies must be negative, this is checked by this filter. A threshold is given as Padé overshoots when the function goes sharply to 0. See for example the semi-circular spectral function of the Bethe lattice with infinite Coordination number as example.

gftool.pade.FilterNegImagNum

gftool.pade.**FilterNegImagNum**(*abs_num=None*, *rel_num=None*)

Return function to check how bad the imaginary part gets.

This methods is designed to create *valid_pades* for `Averager`. The imaginary part of retarded Green's functions and self-energies must be negative, this is checked by this filter. All continuations that are *valid* in this sense are kept, the worst invalid are dropped till only *abs_num* remain.

Warning: Only checked for flat inputs.

gftool.pade.Mod_Averager

gftool.pade.**Mod_Averager**(*z_in*, *coeff*, *mod_fct*, ***, *valid_pades*, *kind:* KindSelector, *vectorized=True*)

Create function for averaging Padé scheme using *mod_fct* before the average.

This function behaves like `Averager` just that *mod_fct* is applied before taking the averages. This should be used, if not the analytic continuation but a mollification thereof is used.

Parameters

z_in
[(N_in,) complex ndarray] Complex mesh used to calculate *coeff*.

coeff
[(…, N_in) complex ndarray] Coefficients for Padé, calculated from *pade.coefficients*.

mod_fct
[callable] Modification of the analytic continuation. The signature of the function should be *mod_fct* (z, pade_z, *args, **kwds), the tow first arguments are the point of evaluation z and the single Padé approximants.

valid_pades
[list_like of bool] Mask which continuations are correct, all Padés where *valid_pades* evaluates to false will be ignored for the average.

kind
[{KindGf, KindSelf}] Defines the asymptotic of the continued function and the number of minimum and maximum input points used for Padé. For `KindGf` the function goes like $1/z$ for large z, for `KindSelf` the function behaves like a constant for large z.

vectorized
[bool, optional] If *vectorized*, all approximants are given to the function simultaneously where the first dimension corresponds to the approximants. If not *vectorized*, *mod_fct* will be called for every approximant separately (default: True).

Returns

function
The continued function *f(z)* (*z*,) -> Result. *f(z).x* contains the function values *f(z).err* the associated variance.

Raises

TypeError
If *valid_pades* not of type *bool*

RuntimeError
If all there are none elements of *valid_pades* that evaluate to True.

gftool.pade.apply_filter

gftool.pade.**apply_filter**(**filters*, *validity_iter*)

Handle usage of filters for Padé.

Parameters

***filters**
[callable] Functions to determine which continuations to keep.

validity_iter
[iterable of (…, N_z) complex np.ndarray] The iterable of analytic continuations as generated by *calc_iterator*.

Returns

(…) bool np.ndarray
Array to index which continuations are good.

gftool.pade.averaged

gftool.pade.**averaged**(*z_out*, *z_in*, ***, *valid_z=None*, *fct_z=None*, *coeff=None*, *filter_valid=None*, *kind:* KindSelector)

Return the averaged Padé continuation with its variance.

The output is checked to have an imaginary part smaller than *threshold*, as retarded Green's functions and self-energies have a negative imaginary part. This is a helper to conveniently get the continuation, it comes however with overhead.

Parameters

z_out
[(N_out,) complex ndarray] Points at with the functions will be evaluated.

z_in
[(N_in,) complex ndarray] Complex mesh used to calculate *coeff*.

valid_z
[(N_out,) complex ndarray, optional] The output range according to which the Padé approximation is validated (compared to the *threshold*).

fct_z
[(N_z,) complex ndarray, optional] Function at points z from which the coefficients will be calculated. Can be omitted if *coeff* is directly given.

coeff
[(N_in,) complex ndarray, optional] Coefficients for Padé, calculated from *pade.coefficients*. Can be given instead of *fct_z*.

filter_valid
[callable or iterable of callable] Function determining which approximants to keep. The signature should be filter_valid(iterable) -> bool ndarray. Currently there are the functions {*FilterNegImag*, *FilterNegImagNum*, *FilterHighVariance*} implemented to generate filter functions. Look into the implemented for details to create new filters.

kind
[{KindGf, KindSelf}] Defines the asymptotic of the continued function and the number of minimum and maximum input points used for Padé. For `KindGf` the function goes like $1/z$ for large z, for `KindSelf` the function behaves like a constant for large z.

Returns

averaged.x
[(N_in, N_out) complex ndarray] Function evaluated at points z.

averaged.err
[(N_in, N_out) complex ndarray] Variance associated with the function values *pade.x* at points z.

gftool.pade.avg_no_neg_imag

gftool.pade.**avg_no_neg_imag**(*z_out*, *z_in*, ***, *valid_z=None*, *fct_z=None*, *coeff=None*, *threshold=1e-08*, *kind:* KindSelector)

Average Padé filtering approximants with non-negative imaginary part.

This function wraps `averaged`, see `averaged` for the parameters.

Returns

averaged.x
[(N_in, N_out) complex ndarray] Function evaluated at points z.

averaged.err
[(N_in, N_out) complex ndarray] Variance associated with the function values *pade.x* at points z.

Other Parameters

threshold
[float, optional] The numerical threshold, how large of an positive imaginary part is tolerated (default: 1e-8). `np.infty` can be given to accept all.

gftool.pade.calc_iterator

gftool.pade.**calc_iterator**(*z_out*, *z_in*, *coeff*)

Calculate Padé continuation of function at points *z_out*.

The continuation is calculated for different numbers of coefficients taken into account, where the number is in [n_min, n_max]. The algorithm is take from [2].

Parameters

z_out
[complex ndarray] Points at with the functions will be evaluated.

z_in
[(N_in,) complex ndarray] Complex mesh used to calculate *coeff*.

coeff
[(…, N_in) complex ndarray] Coefficients for Padé, calculated from *pade.coefficients*.

Yields

pade_calc
[(…, N_in, z_out.shape) complex np.ndarray] Function evaluated at points *z_out*. numbers of Matsubara frequencies between *n_min* and *n_max*. The shape of the elements is the same as *coeff.shape* with the last dimension corresponding to N_in replaced by the shape of *z_out*: (…, N_in, *z_out.shape).

References

[2]

gftool.pade.coefficients

gftool.pade.**coefficients**(*z*, *fct_z*) → ndarray

Calculate the coefficients for the Padé continuation.

Parameters

z
[(N_z,) complex ndarray] Array of complex points.

fct_z
[(…, N_z) complex ndarray] Function at points *z*.

Returns

(…, N_z) complex ndarray
Array of Padé coefficients, needed to perform Padé continuation. Has the same same shape as *fct_z*.

Raises

ValueError
If the size of *z* and the last dimension of *fct_z* do not match.

Notes

The calculation is always done in quad precision (complex256), as it is very sensitive towards rounding errors. Afterwards the type of the result is cast back to double precision (complex128) unless the input data of *fct_z* was already quad precision {float128, complex256}, see `_PRECISE_TYPES`. This avoids giving the illusion that the results are more precise than the input.

Classes

KindGf(n_min, n_max)	Filter approximants such that the high-frequency behavior is $1/$.
KindSelector(n_min, n_max)	Abstract filter class to determine high-frequency behavior of Padé.
KindSelf(n_min, n_max)	Filter approximants such that the high-frequency behavior is a constant.

gftool.pade.KindGf

class gftool.pade.**KindGf**(*n_min*, *n_max*)

Filter approximants such that the high-frequency behavior is $1/$.

We denote approximants with the corresponding high frequency behavior as *valid*. Considers all valid approximants including between *n_min* and *n_max* Matsubara frequencies.

__init__(*n_min*, *n_max*)

Consider approximants including between *n_min* and *n_max* Matsubara frequencies.

Methods

__init__(n_min, n_max)	Consider approximants including between *n_min* and *n_max* Matsubara frequencies.
islice(iterable)	Return an iterator whose next() method returns valid values from *iterable*.

gftool.pade.KindGf.__init__

KindGf.**__init__**(*n_min*, *n_max*)

Consider approximants including between *n_min* and *n_max* Matsubara frequencies.

gftool.pade.KindGf.islice

KindGf.**islice**(*iterable*)

Return an iterator whose next() method returns valid values from *iterable*.

Attributes

slice	Return slice selecting the valid approximants.

gftool.pade.KindGf.slice

property KindGf.**slice**

Return slice selecting the valid approximants.

gftool.pade.KindSelector

class gftool.pade.**KindSelector**(*n_min*, *n_max*)

Abstract filter class to determine high-frequency behavior of Padé.

We denote approximants with the corresponding high frequency behavior as *valid*. Considers all valid approximants including between *n_min* and *n_max* Matsubara frequencies.

abstract __init__(*n_min*, *n_max*)

Consider approximants including between *n_min* and *n_max* Matsubara frequencies.

Methods

__init__(n_min, n_max)	Consider approximants including between *n_min* and *n_max* Matsubara frequencies.
islice(iterable)	Return an iterator whose next() method returns valid values from *iterable*.

gftool.pade.KindSelector.__init__

abstract KindSelector.**__init__**(*n_min*, *n_max*)

Consider approximants including between *n_min* and *n_max* Matsubara frequencies.

gftool.pade.KindSelector.islice

KindSelector.**islice**(*iterable*)

Return an iterator whose next() method returns valid values from *iterable*.

Attributes

slice	Return slice selecting the valid approximants.

gftool.pade.KindSelector.slice

property KindSelector.**slice**

Return slice selecting the valid approximants.

gftool.pade.KindSelf

class gftool.pade.**KindSelf**(*n_min*, *n_max*)

Filter approximants such that the high-frequency behavior is a constant.

We denote approximants with the corresponding high frequency behavior as *valid*. Considers all valid approximants including between *n_min* and *n_max* Matsubara frequencies.

__init__(*n_min*, *n_max*)

Consider approximants including between *n_min* and *n_max* Matsubara frequencies.

Methods

__init__(n_min, n_max)	Consider approximants including between *n_min* and *n_max* Matsubara frequencies.
islice(iterable)	Return an iterator whose next() method returns valid values from *iterable*.

gftool.pade.KindSelf.__init__

KindSelf.**__init__**(*n_min*, *n_max*)

Consider approximants including between *n_min* and *n_max* Matsubara frequencies.

gftool.pade.KindSelf.islice

KindSelf.**islice**(*iterable*)

Return an iterator whose next() method returns valid values from *iterable*.

Attributes

slice	Return slice selecting the valid approximants.

gftool.pade.KindSelf.slice

property KindSelf.**slice**

Return slice selecting the valid approximants.

3.1.11 gftool.polepade

Padé based on robust pole finding.

Instead of fitting a rational polynomial, poles and zeros or poles and corresponding residues are fitted.

The algorithm is based on [ito2018] and adjusted to Green's functions and self- energies. A very short summary can of the algorithm can be found in the appendix of [weh2020]. We assume that we know exactly the high-frequency behavior of the function we want to continue. Here, we will call it *degree* and we define it as the behavior of the function *f(z)* for large *abs(z)*:

$$f(z) \approx z^{degree}.$$

For the diagonal element's of the one-particle Green's function this is *degree=-1* for the self-energy it is *degree=0*.

References

Examples

The function `continuation` provides a high-level interface which can be used for convenience. Let's consider an optimal example: We know the Green's function for complex frequencies on the unit (half-)circle. We consider the Bethe Green's function.

```
>>> z = np.exp(1j*np.linspace(np.pi, 0, num=252)[1:-1])
>>> gf_z = gt.bethe_gf_z(z, half_bandwidth=1)
>>> pade = gt.polepade.continuation(z, gf_z, degree=-1, moments=[1])
>>> print(f"[{pade.zeros.size}/{pade.poles.size}]")
[14/15]
```

We obtain a `[14/15](z)` Padé approximant. Let's compare it on the real axis:

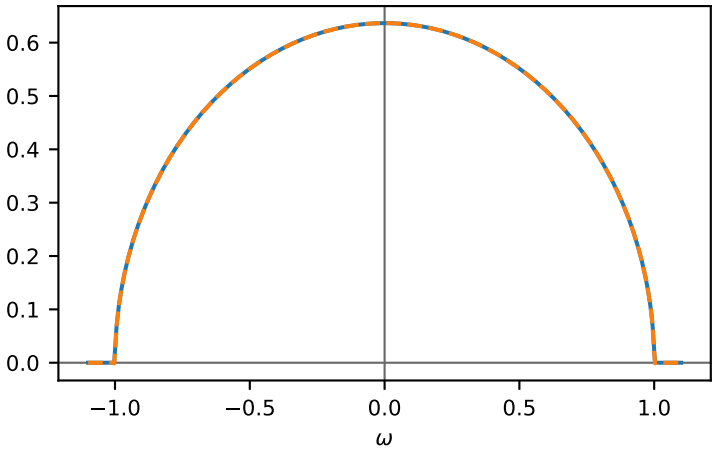
```
>>> import matplotlib.pyplot as plt
>>> ww = np.linspace(-1.1, 1.1, num=500) + 1e-6j
>>> gf_ww = gt.bethe_gf_z(ww, half_bandwidth=1)
>>> pade_ww = pade.eval_polefct(ww)
>>> __ = plt.axhline(0, color='dimgray', linewidth=0.8)
>>> __ = plt.axvline(0, color='dimgray', linewidth=0.8)
>>> __ = plt.plot(ww.real, -gf_ww.imag/np.pi)
>>> __ = plt.plot(ww.real, -pade_ww.imag/np.pi, '--')
>>> __ = plt.xlabel(r"$\omega$")
>>> plt.show()
```

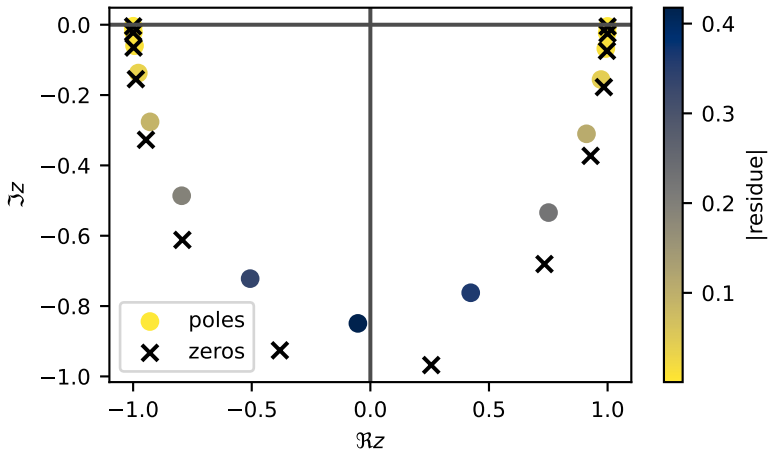
Beside the band-edge, we get a nice fit. We can also investigate the pole structure of the fit:

```
>>> pade.plot()
>>> plt.show()
```

Using a grid on the imaginary axis, the fit is of course worse. Note, that its typically better to continue the self-energy instead of the Green's function, see appendix of [weh2020]. For more control, instead of using `continuation` the elementary functions can be used:

- `number_poles` to determine the degree of the Padé approximant
- `poles`, `zeros` to calculate the poles and zeros of the approximant
- `residues_ols` to calculate the residues


0.6
0.5
0.4
0.3
0.2
0.1
0.0
−1.0
−0.5
0.0
0.5
1.0
ω


0.0
−0.2
−0.4
−0.6
−0.8
−1.0
ℑz
poles
zeros
−1.0
−0.5
0.0
0.5
1.0
ℜz
0.4
0.3
0.2
0.1
|residue|

API

Functions

asymptotic(z, fct_z, zeros, poles[, weight])	Calculate large z asymptotic from *roots* and poles.
continuation(z, fct_z[, degree, weight, ...])	Perform the Padé analytic continuation of *(z, fct_z)*.
number_poles(z, fct_z, *[, degree, weight, ...])	Estimate the optimal number of poles for a rational approximation.
poles(z, fct_z, *[, n, vandermond, weight])	Calculate position of the m poles.
residues_ols(z, fct_z, poles[, weight, moments])	Calculate the residues using ordinary least square.
zeros(z, fct_z, poles, *[, n, vandermond, ...])	Calculate position of n zeros given the poles.

gftool.polepade.asymptotic

gftool.polepade.**asymptotic**(*z*, *fct_z*, *zeros*, *poles*, *weight=None*)

Calculate large z asymptotic from *roots* and poles.

We assume *f(z) = a * np.prod(z - zeros) / np.prod(z - poles)*, therefore The asymptotic for large *abs(z)* is $f(z) \approx a * z^{**}(zeros.size - poles.size)$.

Parameters

z, fct_z
[(N_z) complex np.ndarray] Variable where function is evaluated and function values.

zeros, poles
[(n), (m) complex np.ndarray] Position of the zeros and poles of the function.

weight
[(N_z) float np.ndarray, optional] Weighting of the data points, for a known error σ this should be *weight = 1./σ*.

Returns

asym, std
[float] Large z asymptotic and its standard deviation.

gftool.polepade.continuation

gftool.polepade.**continuation**(*z*, *fct_z*, *degree=-1*, *weight=None*, *moments=()*, *vandermond=<function polyvander>*, *rotate=None*, *real_asymp=True*) → *PadeApprox*

Perform the Padé analytic continuation of *(z, fct_z)*.

Parameters

z, fct_z
[(N_z) complex np.ndarray] Variable where function is evaluated and function values.

degree
[int, optional] The difference of denominator and numerator degree (default: -1). This determines how *fct_z* decays for large *abs(z)*: $fct\_z \to z^{**}degree$. For Green's functions it typically is *-1*, for self-energies it typically is *0*.

weight
[(N_z) float np.ndarray, optional] Weighting of the data points, for a known error σ this should be *weight = 1./σ*.

moments
[(N) float array_like] Moments of the high-frequency expansion, where *f(z) = moments / z**np.arange(1, N+1)* for large *z*. This only affects the calculated *pade.residues*, and constrains them to fulfill the *moments*.

Returns

PadeApprox
Padé analytic continuation parametrized by *pade.zeros*, *pade.poles* and *pade.residues*.

Other Parameters

vandermond
[Callable, optional] Function giving the Vandermond matrix of the chosen polynomial basis. Defaults to simple polynomials.

rotate
[bool or None, optional] Whether to rotate the coordinate to calculated zeros and poles (default: rotate if *z* is purely imaginary).

real_asymp
[bool, optional] Whether to consider only the real part of the asymptote, or treat it as complex number. Physically, to asymptote should typically be real (default: True).

Examples

```
>>> beta = 100
>>> iws = gt.matsubara_frequencies(range(512), beta=beta)
>>> gf_iw = gt.square_gf_z(iws, half_bandwidth=1)
>>> weight = 1./iws.imag  # put emphasis on low frequencies
>>> gf_pade = gt.polepade.continuation(iws, gf_iw, weight=weight, moments=[1.])
```

Compare the result on the real axis:

```
>>> import matplotlib.pyplot as plt
>>> ww = np.linspace(-1.1, 1.1, num=5000)
>>> __ = plt.plot(ww, gt.square_dos(ww, half_bandwidth=1))
>>> __ = plt.plot(ww, -1. / np.pi * gf_pade.eval_zeropole(ww).imag)
>>> __ = plt.plot(ww, -1. / np.pi * gf_pade.eval_polefct(ww).imag)
>>> plt.show()
```

Investigate the pole structure of the continuation:

```
>>> gf_pade.plot()
>>> plt.show()
```

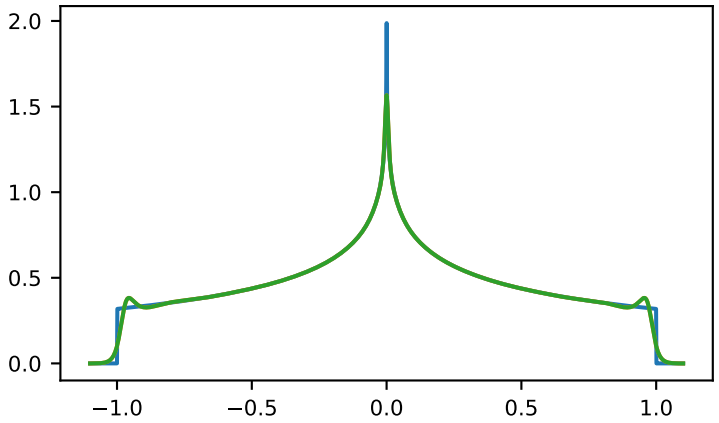
gftool.polepade.number_poles

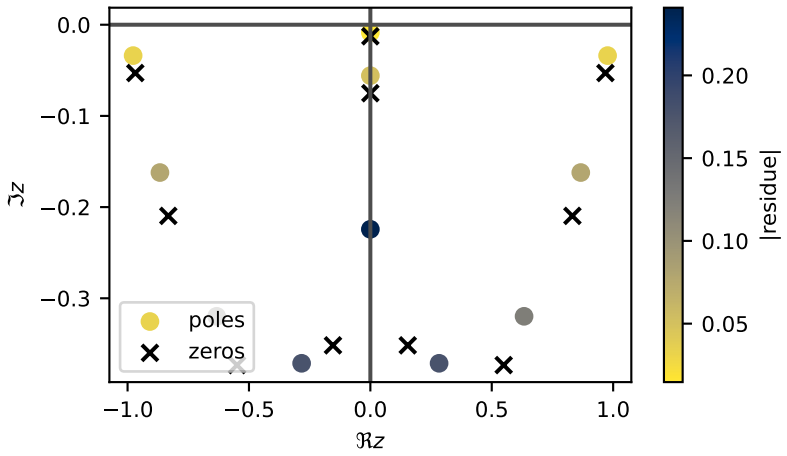
gftool.polepade.**number_poles**(*z*, *fct_z*, ***, *degree=-1*, *weight=None*, *n_poles0: ~typing.Optional[int] = None*, *vandermond=<function polyvander>*) → int

Estimate the optimal number of poles for a rational approximation.

The number of poles is determined, such that up to numerical accuracy the solution is unique, corresponding to a null-dimension equal to 1 [ito2018].

Parameters


2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0


0.0
−0.1
−0.2
−0.3
$\Im z$
poles
zeros
−1.0
−0.5
0.0
0.5
1.0
$\Re z$
0.20
0.15
0.10
0.05
|residue|

z, fct_z
[(N_z) complex np.ndarray] Variable where function is evaluated and function values.

degree
[int, optional] The difference of denominator and numerator degree. (default: -1) This determines how *fct_z* decays for large *abs(z)*: *fct_z → z**degree*. For Green's functions it typically is *-1*, for self-energies it typically is *0*.

weight
[(N_z) float np.ndarray, optional] Weighting of the data points, for a known error σ this should be *weight = 1./σ*.

n_poles0
[int, optional] Starting guess for the number of poles. Can be given to speed up calculation if a good estimate is available.

vandermond
[Callable, optional] Function giving the Vandermond matrix of the chosen polynomial basis. Defaults to simple polynomials.

Returns

int
Best guess for optimal number of poles.

References

[ito2018]

gftool.polepade.poles

gftool.polepade.**poles**(*z*, *fct_z*, ***, *n: ~typing.Optional[int] = None*, *m: int*, *vandermond=<function polyvander>*, *weight=None*)

Calculate position of the *m* poles.

Parameters

z, fct_z
[(N_z) complex np.ndarray] Variable where function is evaluated and function values.

n, m
[int] Number of zeros and poles of the function. For large z the function is proportional to z^{n-m} (*n* defaults to *m-1*).

vandermond
[Callable, optional] Function giving the Vandermond matrix of the chosen polynomial basis.

weight
[(N_z) float np.ndarray, optional] Weighting of the data points, for a known error σ this should be *weight = 1./σ*.

Returns

(m) complex np.ndarray The position of the poles.

Notes

The calculation closely follows [ito2018], we just adjust the scaling.

References

[ito2018]

gftool.polepade.residues_ols

gftool.polepade.**residues_ols**(*z*, *fct_z*, *poles*, *weight=None*, *moments=()*)

Calculate the residues using ordinary least square.

Parameters

z, fct_z
[(N_z) complex np.ndarray] Variable where function is evaluated and function values.

poles
[(M) complex np.ndarray] Position of the poles of the function.

weight
[(N_z) float np.ndarray, optional] Weighting of the data points, for a known error σ this should be *weight = 1./σ*.

moments
[(N) float array_like] Moments of the high-frequency expansion, where *f(z) = moments / z**np.arange(1, N+1)* for large *z*.

Returns

residues
[(M) complex np.ndarray] The residues corresponding to the `poles`.

residual
[(1)] Norm of the residual.

gftool.polepade.zeros

gftool.polepade.**zeros**(*z*, *fct_z*, *poles*, ***, *n: ~typing.Optional[int] = None*, *vandermond=<function polyvander>*, *weight=None*)

Calculate position of *n* zeros given the `poles`.

Parameters

z, fct_z
[(N_z) complex np.ndarray] Variable where function is evaluated and function values.

poles
[(m) complex np.ndarray] Position of the poles of the function.

n
[int] Number of zeros. For large *z* the function is proportional to *z**(n - m)* (default: `m-1`).

vandermond
[Callable, optional] Function giving the Vandermond matrix of the chosen polynomial basis.

weight
[(N_z) float np.ndarray, optional] Weighting of the data points, for a known error σ this should be *weight = 1./σ*.

Returns

(n) complex np.ndarray The position of the zeros.

Notes

The calculation closely follows [ito2018], we just adjust the scaling.

References

[ito2018]

Classes

PadeApprox(zeros, poles, residues, amplitude)	Representation of the Padé approximation based on poles.

gftool.polepade.PadeApprox

class gftool.polepade.**PadeApprox**(*zeros: ndarray*, *poles: ndarray*, *residues: ndarray*, *amplitude: ndarray*)

Representation of the Padé approximation based on poles.

Basically the approximation is obtained as `PoleFct` as well as *ZeroPole*. Note however, that those to approximations will in general not agree. Nevertheless, for a good approximation they should be very similar.

Parameters

zeros
[(…, Nz) complex np.ndarray] Zeros of the represented function.

poles, residues
[(…, Np) complex np.ndarray] Poles and the corresponding residues of the represented function.

amplitude
[(…) complex np.ndarray or complex] The amplitude of the function. This is also the large *abs(z)* limit of the function *ZeroPole.eval(z) = amplitude * z**(Nz-Np)*.

__init__(*zeros: ndarray*, *poles: ndarray*, *residues: ndarray*, *amplitude: ndarray*) → None

Methods

__init__(zeros, poles, residues, amplitude)	

eval_polefct(z)	Evaluate the PoleFct representation.
eval_zeropole(z)	Evaluate the ZeroPole representation.
moments(order)	Calculate high-frequency moments of PoleFct representation.
plot([residue, axis])	Represent the function as scatter plot.

gftool.polepade.PadeApprox.__init__

PadeApprox.**__init__**(*zeros: ndarray*, *poles: ndarray*, *residues: ndarray*, *amplitude: ndarray*) → None

gftool.polepade.PadeApprox.eval_polefct

PadeApprox.**eval_polefct**(*z*)

Evaluate the PoleFct representation.

gftool.polepade.PadeApprox.eval_zeropole

PadeApprox.**eval_zeropole**(*z*)

Evaluate the ZeroPole representation.

gftool.polepade.PadeApprox.moments

PadeApprox.**moments**(*order*)

Calculate high-frequency moments of PoleFct representation.

gftool.polepade.PadeApprox.plot

PadeApprox.**plot**(*residue=False*, *axis=None*)

Represent the function as scatter plot.

Attributes

zeros
poles
residues
amplitude

gftool.polepade.PadeApprox.zeros

PadeApprox.**zeros**: ndarray

gftool.polepade.PadeApprox.poles

PadeApprox.**poles**: ndarray

gftool.polepade.PadeApprox.residues

PadeApprox.**residues**: ndarray

gftool.polepade.PadeApprox.amplitude

PadeApprox.**amplitude**: ndarray

3.1.12 gftool.siam

Basic functions for the (non-interacting) single impurity Anderson model (SIAM).

The Hamiltonian for the SIAM reads

$$H = \sum c^\dagger c + U n_\downarrow n_\uparrow + \sum_k (V_k c^\dagger c_k + H.c.) + \sum_{kk} n_k$$

The first to terms describe the interacting single impurity, the thirds term is the hopping (or hybridization) between impurity and bath-sites, the last term is the on-site energy of the non-interacting bath sites.

In the action formalism, the bath degrees of freedom can be readily integrated out, as the action is quadratic in these degrees. The local action of the impurity reads

$$S_{imp}[c^+, c] = -\sum_n c^+[i_n - \ - (i_n)]c + U\int_0 d n_\uparrow() n_\downarrow()$$

with the hybridization function

$$(z) = \sum_k |V_k|^2/(z - {}_k).$$

API

Functions

gf0_loc_gr_t(tt, e_onsite, e_bath, hopping, beta)	Noninteracting greater local Green's function for the impurity.
gf0_loc_le_t(tt, e_onsite, e_bath, hopping, beta)	Noninteracting lesser local Green's function for the impurity.
gf0_loc_ret_t(tt, e_onsite, e_bath, hopping)	Noninteracting retarded local Green's function for the impurity.
gf0_loc_z(z, e_onsite, e_bath, hopping_sqr)	Noninteracting local Green's function for the impurity.
hamiltonian_matrix(e_onsite, e_bath, hopping)	One-particle Hamiltonian matrix of the SIAM.
hybrid_z(z, e_bath, hopping_sqr)	Hybridization function of the impurity.

gftool.siam.gf0_loc_gr_t

gftool.siam.**gf0_loc_gr_t**(*tt*, *e_onsite*, *e_bath*, *hopping*, *beta*)

Noninteracting greater local Green's function for the impurity.

Parameters

tt
[(…) float np.ndarray] Time variable. Note that the retarded Green's function is *0* for *tt<0*.

e_onsite
[(…) float np.ndarray] On-site energy of the impurity site.

e_bath
[(…, Nb) float np.ndarray] On-site energy of the bath sites.

hopping
[(…, Nb) complex np.ndarray] Hopping matrix element between impurity and the bath sites.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

(…) complex np.ndarray
Greater Green's function of the impurity site.

gftool.siam.gf0_loc_le_t

gftool.siam.**gf0_loc_le_t**(*tt*, *e_onsite*, *e_bath*, *hopping*, *beta*)

Noninteracting lesser local Green's function for the impurity.

Parameters

tt
[(…) float np.ndarray] Time variable. Note that the retarded Green's function is *0* for *tt<0*.

e_onsite
[(…) float np.ndarray] On-site energy of the impurity site.

e_bath
[(…, Nb) float np.ndarray] On-site energy of the bath sites.

hopping
[(…, Nb) complex np.ndarray] Hopping matrix element between impurity and the bath sites.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

(…) complex np.ndarray
Lesser Green's function of the impurity site.

gftool.siam.gf0_loc_ret_t

gftool.siam.**gf0_loc_ret_t**(*tt*, *e_onsite*, *e_bath*, *hopping*)

Noninteracting retarded local Green's function for the impurity.

Parameters

tt
[(…) float np.ndarray] Time variable. Note that the retarded Green's function is *0* for *tt<0*.

e_onsite
[(…) float np.ndarray] On-site energy of the impurity site.

e_bath
[(…, Nb) float np.ndarray] On-site energy of the bath sites.

hopping
[(…, Nb) complex np.ndarray] Hopping matrix element between impurity and the bath sites.

Returns

(…) complex np.ndarray
Retarded Green's function of the impurity site.

gftool.siam.gf0_loc_z

gftool.siam.**gf0_loc_z**(*z*, *e_onsite*, *e_bath*, *hopping_sqr*)

Noninteracting local Green's function for the impurity.

Parameters

z
[(…) complex np.ndarray] Complex frequency variable.

e_onsite
[(…) float np.ndarray] On-site energy of the impurity site.

e_bath
[(…, Nb) float np.ndarray] On-site energy of the bath sites.

hopping_sqr
[(…, Nb) complex np.ndarray] Absolute square of hopping matrix element between impurity and the bath sites.

Returns

(…) complex np.ndarray
Green's function of the impurity site.

gftool.siam.hamiltonian_matrix

gftool.siam.**hamiltonian_matrix**(*e_onsite*, *e_bath*, *hopping*)

One-particle Hamiltonian matrix of the SIAM.

The non-interacting Hamiltonian can be written in the form

$$\hat{H} = \sum_{ij} c_i^\dagger H_{ij} c_j.$$

The Hamiltonian matrix is H_{ij}, where we fixed the spin σ. The element *H_{00}* corresponds to the impurity site.

Parameters

e_onsite
[(…) float np.ndarray] On-site energy of the impurity site.

e_bath
[(…, Nb) float np.ndarray] On-site energy of the bath sites.

hopping
[(…, Nb) complex np.ndarray] Hopping matrix element between impurity and the bath sites.

Returns

(…, Nb+1, Nb+1) complex np.ndarray
Lesser Green's function of the impurity site.

gftool.siam.hybrid_z

gftool.siam.**hybrid_z**(*z*, *e_bath*, *hopping_sqr*)

Hybridization function of the impurity.

Parameters

z
[(…) complex np.ndarray] Complex frequency variable.

e_bath
[(…, Nb) float np.ndarray] On-site energy of the bath sites.

hopping_sqr
[(…, Nb) complex np.ndarray] Absolute square of hopping matrix element between impurity and the bath sites.

Returns

(…) complex np.ndarray
Hybridization function of the impurity site.

3.2 Glossary

DOS
: Density of States

eps
epsilon
ϵ
: (Real) energy variable. Typically used for for the *DOS* where it replaces the k-dependent Dispersion $_k$.

iv
iv_n
: Bosonic Matsubara frequencies

iw
iω_n
: Fermionic Matsubara frequencies

tau
τ
: Imaginary time points

z
: Complex frequency variable

3.3 Green's functions and lattices

1D

onedim_dos(eps, half_bandwidth)	DOS of non-interacting 1D lattice.
onedim_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the 1D DOS.
onedim_gf_z(z, half_bandwidth)	Local Green's function of the 1D lattice.
onedim_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the 1D lattice.

3.3.1 gftool.onedim_dos

gftool.**onedim_dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 1D lattice.

Diverges at the band-edges *abs(eps)* = *half_bandwidth*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.onedim.dos_mp
Multi-precision version suitable for integration.

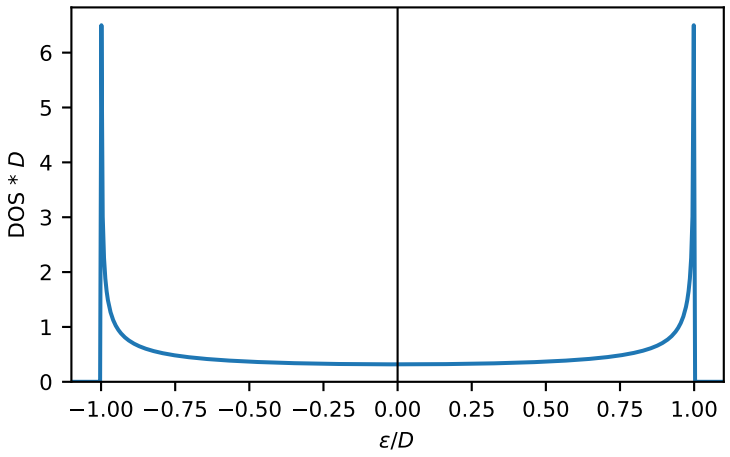
References

[economou2006]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=501)
>>> dos = gt.lattice.onedim.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.2 gftool.onedim_dos_moment

gftool.**onedim_dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the 1D DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 1D lattice.

Returns

float
The *m* th moment of the 1D DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.onedim.dos

3.3.3 gftool.onedim_gf_z

gftool.**onedim_gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 1D lattice.

$$G(z) = \frac{1}{2}\int_{-} \frac{d}{z - D\cos()}$$

where D is the half bandwidth. The integral can be evaluated in the complex plane along the unit circle. See [economou2006].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the 1D lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the square lattice Green's function.

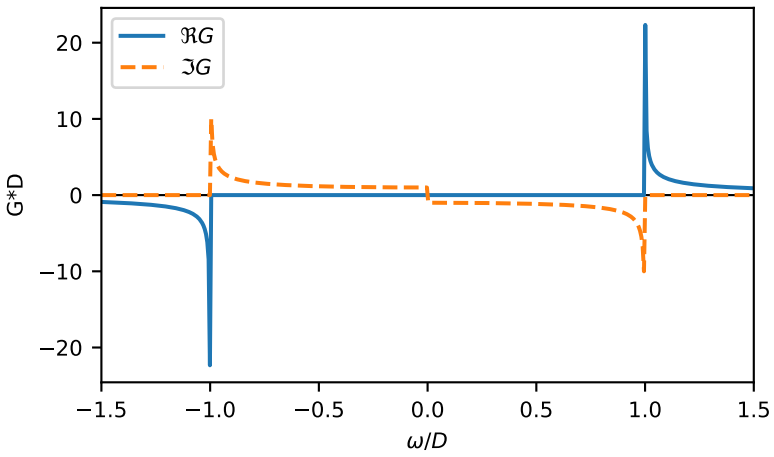
References

[economou2006]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.onedim.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel("G*D")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.4 gftool.onedim_hilbert_transform

`gftool.`**`onedim_hilbert_transform`**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the 1D lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi

[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 1D lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.onedim.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$2t = D$$

2D

square_dos(eps, half_bandwidth)	DOS of non-interacting 2D square lattice.
square_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the square DOS.
square_gf_z(z, half_bandwidth)	Local Green's function of the 2D square lattice.
square_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the square lattice.
triangular_dos(eps, half_bandwidth)	DOS of non-interacting 2D triangular lattice.
triangular_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the triangular DOS.
triangular_gf_z(z, half_bandwidth)	Local Green's function of the 2D triangular lattice.
triangular_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the triangular lattice.
honeycomb_dos(eps, half_bandwidth)	DOS of non-interacting 2D honeycomb lattice.
honeycomb_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the honeycomb DOS.
honeycomb_gf_z(z, half_bandwidth)	Local Green's function of the 2D honeycomb lattice.
honeycomb_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the honeycomb lattice.

3.3.5 gftool.square_dos

gftool.**square_dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D square lattice.

Has a van Hove singularity (logarithmic divergence) at *eps = 0*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/4*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.square.dos_mp
Multi-precision version suitable for integration.

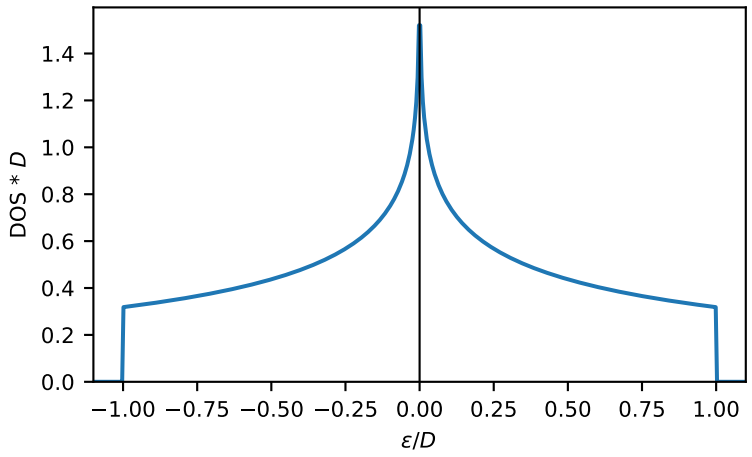
References

[economou2006]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.square.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.6 gftool.square_dos_moment

gftool.**square_dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the square DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D square lattice.

Returns

float
The *m* th moment of the 2D square DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.square.dos

3.3.7 gftool.square_gf_z

gftool.**square_gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D square lattice.

$$G(z) = \frac{2}{z}\int_0^{/2} \frac{d}{\sqrt{1-(D/z)^2\cos^2}}$$

where D is the half bandwidth and the integral is the complete elliptic integral of first kind. See [economou2006].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the square lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/4*.

Returns

complex np.ndarray or complex
Value of the square lattice Green's function.

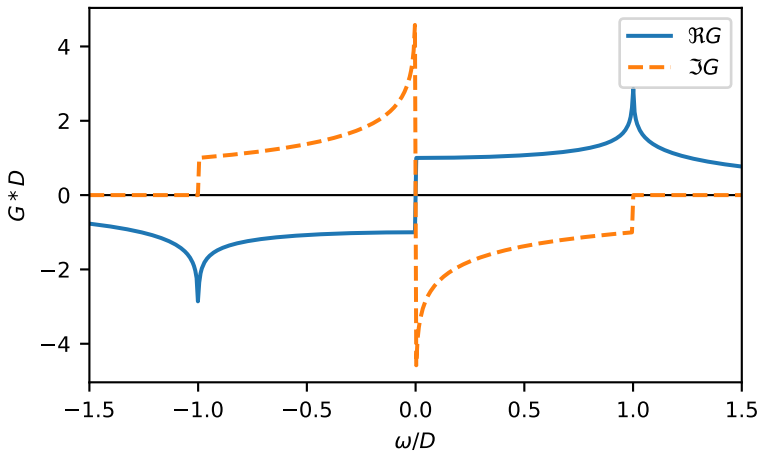
References

[economou2006]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.square.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.8 gftool.square_hilbert_transform

`gftool.`**`square_hilbert_transform`**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the square lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi

[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D square lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.square.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$4t = D$$

3.3.9 gftool.triangular_dos

gftool.**triangular_dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D triangular lattice.

The DOS diverges at *-4/9*half_bandwidth*. The DOS is evaluated as complete elliptic integral of first kind, see [kogan2021].

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(*eps* < -2/3`half_bandwidth`) = 0, DOS(4/3`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 4D/9$.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.triangular.dos_mp
Multi-precision version suitable for integration.

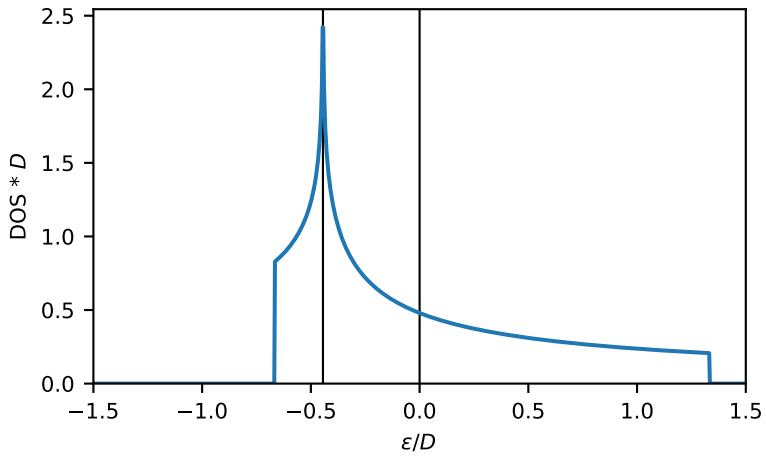
References

[kogan2021]

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=1000)
>>> dos = gt.lattice.triangular.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(-4/9, color='black', linewidth=0.8)
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.10 gftool.triangular_dos_moment

gftool.**triangular_dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the triangular DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D triangular lattice.

Returns

float
The *m* th moment of the 2D triangular DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments m.

See also:

gftool.lattice.triangular.dos

3.3.11 gftool.triangular_gf_z

gftool.**triangular_gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D triangular lattice.

Note, that the spectrum is asymmetric and in $[-2D/3, 4D/3]$, where D is the half-bandwidth. The Green's function is evaluated as complete elliptic integral of first kind, see [horiguchi1972].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the triangular lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 4D/9$.

Returns

complex np.ndarray or complex
Value of the triangular lattice Green's function.

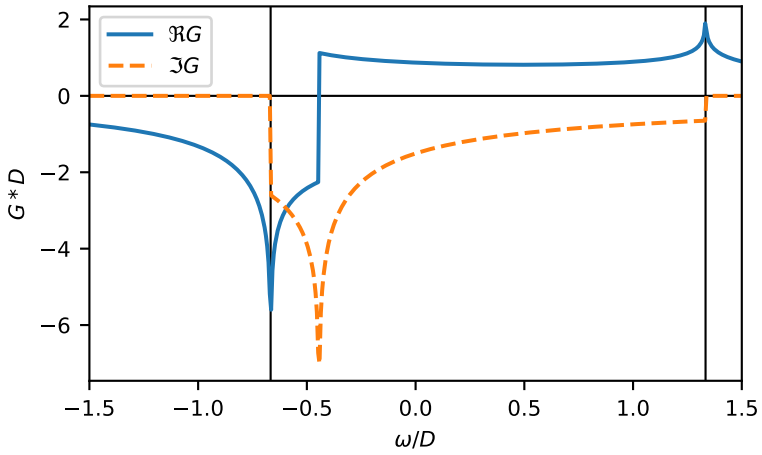
References

[horiguchi1972]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500, dtype=complex) + 1e-64j
>>> gf_ww = gt.lattice.triangular.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.axvline(-2/3, color='black', linewidth=0.8)
>>> _ = plt.axvline(+4/3, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.12 gftool.triangular_hilbert_transform

gftool.**triangular_hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the triangular lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D triangular lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.triangular.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$9t = 4D$$

3.3.13 gftool.honeycomb_dos

gftool.**honeycomb_dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 2D honeycomb lattice.

The DOS diverges at *eps*=±*half_bandwidth/3*. The Green's function and therefore the DOS of the 2D honeycomb lattice can be expressed in terms of the 2D triangular lattice *gftool.lattice.triangular.dos*, see [horiguchi1972].

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.honeycomb.dos_mp
Multi-precision version suitable for integration.

gftool.lattice.triangular.dos

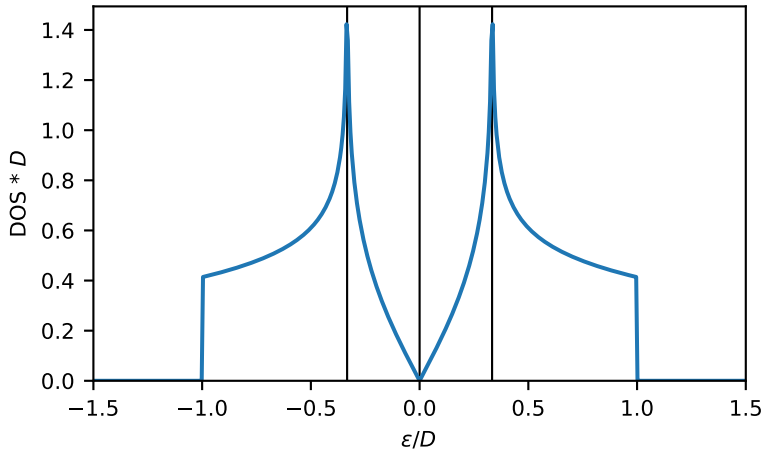
References

[horiguchi1972]

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=501)
>>> dos = gt.lattice.honeycomb.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> for pos in (-1/3, 0, +1/3):
...     _ = plt.axvline(pos, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.14 gftool.honeycomb_dos_moment

`gftool.`**`honeycomb_dos_moment`**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the honeycomb DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D honeycomb lattice.

Returns

float
The *m* th moment of the 2D honeycomb DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.honeycomb.dos

3.3.15 gftool.honeycomb_gf_z

gftool.**honeycomb_gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 2D honeycomb lattice.

The Green's function of the 2D honeycomb lattice can be expressed in terms of the 2D triangular lattice *gftool.lattice.triangular.gf_z*, see [horiguchi1972].

The Green's function has singularities at *z*=±*half_bandwidth/3*.

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the honeycomb lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = 2D/3$.

Returns

complex np.ndarray or complex
Value of the honeycomb lattice Green's function.

See also:

gftool.lattice.triangular.gf_z

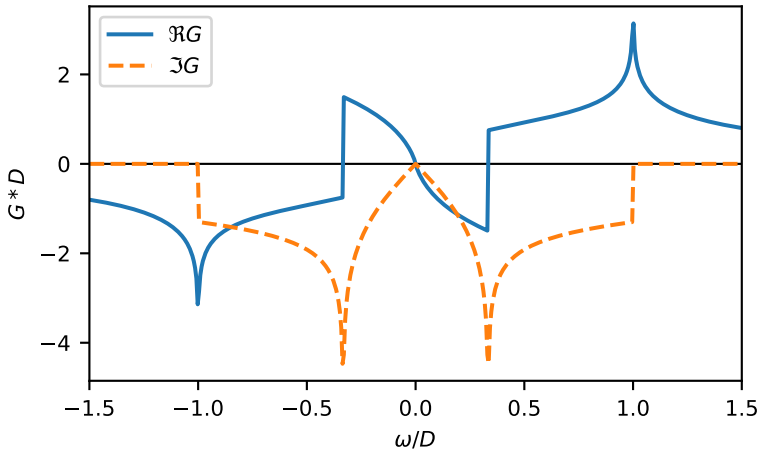
References

[horiguchi1972]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=501, dtype=complex) + 1e-64j
>>> gf_ww = gt.lattice.honeycomb.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.16 gftool.honeycomb_hilbert_transform

gftool.**honeycomb_hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the honeycomb lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 2D honeycomb lattice.

Returns

complex ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.honeycomb.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$3t/2 = D$$

3D

sc_dos(eps[, half_bandwidth])	Local Green's function of 3D simple cubic lattice.
sc_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the simple cubic DOS.
sc_gf_z(z[, half_bandwidth])	Local Green's function of 3D simple cubic lattice.
sc_hilbert_transform(xi[, half_bandwidth])	Hilbert transform of non-interacting DOS of the simple cubic lattice.
bcc_dos(eps, half_bandwidth)	DOS of non-interacting 3D body-centered cubic lattice.
bcc_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the body-centered cubic DOS.
bcc_gf_z(z, half_bandwidth)	Local Green's function of 3D body-centered cubic (bcc) lattice.
bcc_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the body-centered cubic lattice.
fcc_dos(eps, half_bandwidth)	DOS of non-interacting 3D face-centered cubic lattice.
fcc_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the face-centered cubic DOS.
fcc_gf_z(z, half_bandwidth)	Local Green's function of the 3D face-centered cubic (fcc) lattice.
fcc_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the face-centered cubic lattice.

3.3.17 gftool.sc_dos

gftool.**sc_dos**(*eps*, *half_bandwidth=1*)

Local Green's function of 3D simple cubic lattice.

Has a van Hove singularity (continuous but not differentiable) at *abs(eps) = D/3*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS of the simple cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/6$.

Returns

float np.ndarray or float
The value of the DOS.

Notes

Around `eps=0` the expansion Eq. (5.4) using Eq. (7.37) from [joyce1973] is used. Otherwise, it is identical to `-gf_z.imag/np.pi`

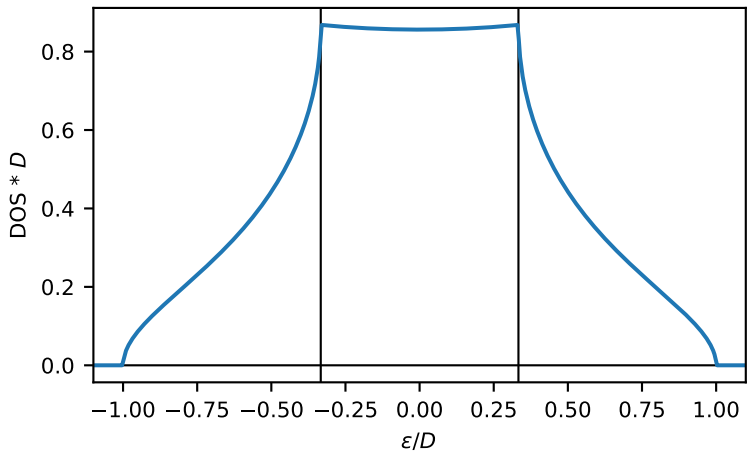
References

[economou2006], [joyce1973], [katsura1971]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=501)
>>> dos = gt.lattice.sc.dos(eps)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color="black", linewidth=0.8)
>>> _ = plt.axvline(-1/3, color="black", linewidth=0.8)
>>> _ = plt.axvline(+1/3, color="black", linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.18 gftool.sc_dos_moment

gftool.**sc_dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the simple cubic DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D simple cubic lattice.

Returns

float
The *m* th moment of the 3D simple cubic DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.sc.dos

3.3.19 gftool.sc_gf_z

gftool.**sc_gf_z**(*z*, *half_bandwidth=1*)

Local Green's function of 3D simple cubic lattice.

Has a van Hove singularity (continuous but not differentiable) at $z = \pm D/3$.

Implements equations (1.24 - 1.26) from [delves2001].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the simple cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping $t = D/6$.

Returns

complex np.ndarray or complex
Value of the simple cubic Green's function at complex energy z.

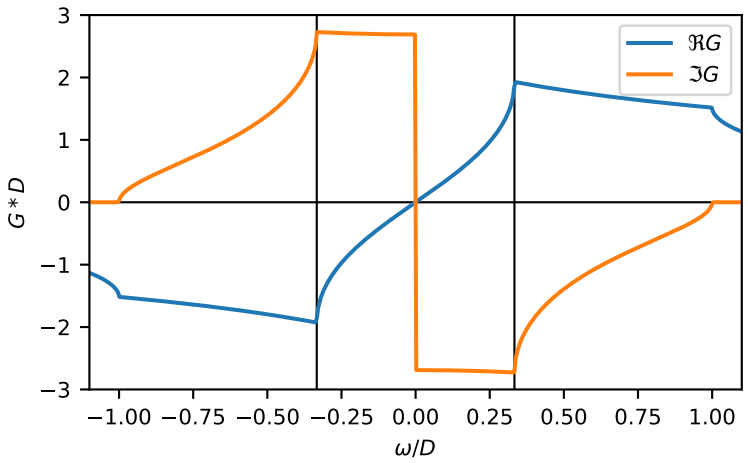
References

[economou2006], [delves2001]

Examples

```
>>> ww = np.linspace(-1.1, 1.1, num=500)
>>> gf_ww = gt.lattice.sc.gf_z(ww)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color="black", linewidth=0.8)
>>> _ = plt.axvline(-1/3, color="black", linewidth=0.8)
>>> _ = plt.axvline(+1/3, color="black", linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.20 gftool.sc_hilbert_transform

`gftool.`**`sc_hilbert_transform`**(*xi*, *half_bandwidth=1*)

Hilbert transform of non-interacting DOS of the simple cubic lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D simple cubic lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.sc.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$6t = D$$

3.3.21 gftool.bcc_dos

gftool.**bcc_dos**(*eps*, *half_bandwidth*)

DOS of non-interacting 3D body-centered cubic lattice.

Has a van Hove singularity (logarithmic divergence) at *eps = 0*.

Parameters

eps
[float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.bcc.dos_mp
Multi-precision version suitable for integration.

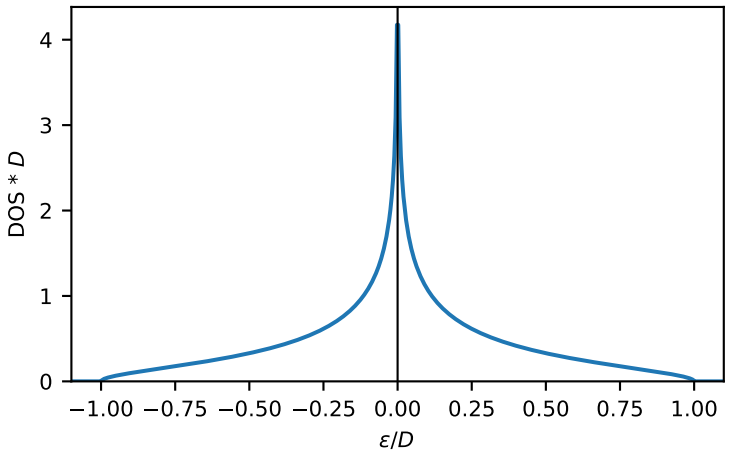
References

[morita1971]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.bcc.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.22 gftool.bcc_dos_moment

gftool.**bcc_dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the body-centered cubic DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D body-centered cubic lattice.

Returns

float
The *m* th moment of the 3D body-centered cubic DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.bcc.dos

3.3.23 gftool.bcc_gf_z

gftool.**bcc_gf_z**(*z*, *half_bandwidth*)

Local Green's function of 3D body-centered cubic (bcc) lattice.

Has a van Hove singularity at *z=0* (divergence).

Implements equations (2.1) and (2.4) from [morita1971]

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency *z*.

half_bandwidth
[float] Half-bandwidth of the DOS of the body-centered cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

complex np.ndarray or complex
Value of the body-centered cubic Green's function at complex energy *z*.

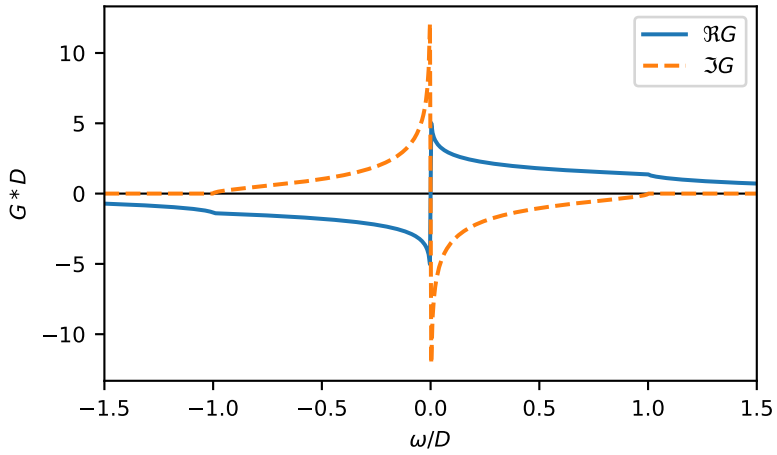
References

[morita1971]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.bcc.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.24 gftool.bcc_hilbert_transform

gftool.**bcc_hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the body-centered cubic lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D body-centered cubic lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.bcc.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$8t = D$$

3.3.25 gftool.fcc_dos

`gftool.`**`fcc_dos`**(*eps*, *half_bandwidth*)

DOS of non-interacting 3D face-centered cubic lattice.

Has a van Hove singularity at *z=-half_bandwidth/2* (divergence) and at *z=0* (continuous but not differentiable).

Parameters

eps
: [float np.ndarray or float] DOS is evaluated at points *eps*.

half_bandwidth
: [float] Half-bandwidth of the DOS, DOS(*eps* < -0.5*`half_bandwidth`) = 0, DOS(1.5*`half_bandwidth` < *eps*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

float np.ndarray or float
: The value of the DOS.

See also:

gftool.lattice.fcc.dos_mp
: Multi-precision version suitable for integration.

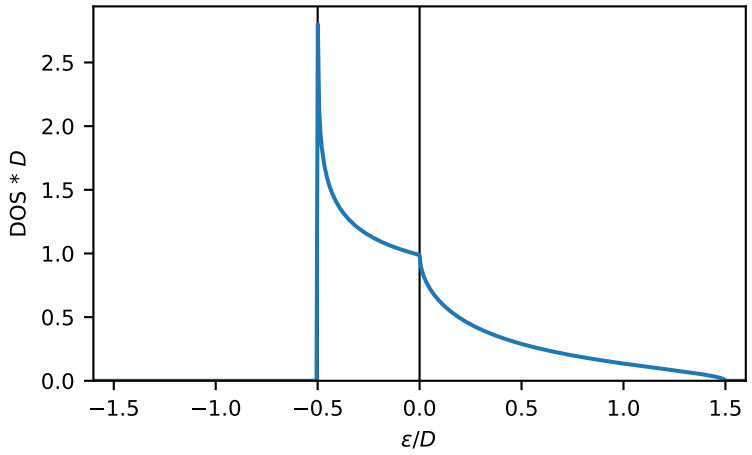
References

[morita1971]

Examples

```
>>> eps = np.linspace(-1.6, 1.6, num=501)
>>> dos = gt.lattice.fcc.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.axvline(-0.5, color='black', linewidth=0.8)
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.26 gftool.fcc_dos_moment

`gftool.`**`fcc_dos_moment`**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the face-centered cubic DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D face-centered cubic lattice.

Returns

float
The *m* th moment of the 3D face-centered cubic DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.fcc.dos

3.3.27 gftool.fcc_gf_z

gftool.**fcc_gf_z**(*z*, *half_bandwidth*)

Local Green's function of the 3D face-centered cubic (fcc) lattice.

Note, that the spectrum is asymmetric and in $[-D/2, 3D/2]$, where D is the half-bandwidth.

Has a van Hove singularity at *z=-half_bandwidth/2* (divergence) and at *z=0* (continuous but not differentiable).

Implements equations (2.16), (2.17) and (2.11) from [morita1971].

Parameters

z
[complex np.ndarray or complex] Green's function is evaluated at complex frequency *z*.

half_bandwidth
[float] Half-bandwidth of the DOS of the face-centered cubic lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/8*.

Returns

complex np.ndarray or complex
Value of the face-centered cubic lattice Green's function.

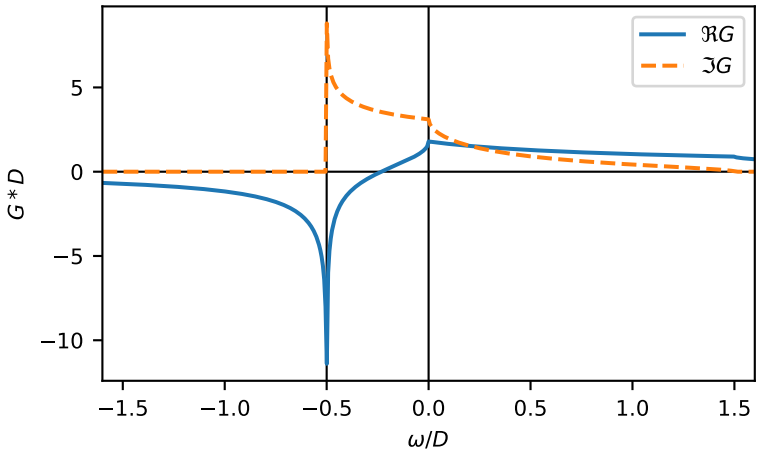
References

[morita1971]

Examples

```
>>> ww = np.linspace(-1.6, 1.6, num=501, dtype=complex)
>>> gf_ww = gt.lattice.fcc.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.axvline(-0.5, color='black', linewidth=0.8)
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.plot(ww.real, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww.real, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.xlim(left=ww.real.min(), right=ww.real.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.28 gftool.fcc_hilbert_transform

gftool.**fcc_hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the face-centered cubic lattice.

The Hilbert transform is defined

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex np.ndarray or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the 3D face-centered cubic lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.fcc.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D

$$8t = D$$

Miscellaneous

bethe_dos(eps, half_bandwidth)	DOS of non-interacting Bethe lattice for infinite coordination number.
bethe_dos_moment(m, half_bandwidth)	Calculate the *m* th moment of the Bethe DOS.
bethe_gf_z(z, half_bandwidth)	Local Green's function of Bethe lattice for infinite coordination number.
bethe_gf_d1_z(z, half_bandwidth)	First derivative of local Green's function of Bethe lattice for infinite coordination number.
bethe_gf_d2_z(z, half_bandwidth)	Second derivative of local Green's function of Bethe lattice for infinite coordination number.
bethe_hilbert_transform(xi, half_bandwidth)	Hilbert transform of non-interacting DOS of the Bethe lattice.
pole_gf_z(z, poles, weights)	Green's function given by a finite number of *poles*.
pole_gf_d1_z(z, poles, weights)	First derivative of Green's function given by a finite number of *poles*.
pole_gf_moments(poles, weights, order)	High-frequency moments of the pole Green's function.
pole_gf_ret_t(tt, poles, weights)	Retarded time Green's function given by a finite number of *poles*.
pole_gf_tau(tau, poles, weights, beta)	Imaginary time Green's function given by a finite number of *poles*.
pole_gf_tau_b(tau, poles, weights, beta)	Bosonic imaginary time Green's function given by a finite number of *poles*.
hubbard_I_self_z(z, U, occ)	Self-energy in Hubbard-I approximation (atomic solution).
hubbard_dimer_gf_z(z, hopping, interaction)	Green's function for the two site Hubbard model on a *dimer*.
surface_gf_zeps(z, eps, hopping_nn)	Surface Green's function for stacked layers.

3.3.29 gftool.bethe_dos

gftool.**bethe_dos**(*eps*, *half_bandwidth*)

DOS of non-interacting Bethe lattice for infinite coordination number.

Parameters

eps
[float array_like or float] DOS is evaluated at points *eps*.

half_bandwidth
[float] Half-bandwidth of the DOS, DOS(| *eps* | > *half_bandwidth*) = 0. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

float np.ndarray or float
The value of the DOS.

See also:

gftool.lattice.bethe.dos_mp
Multi-precision version suitable for integration.

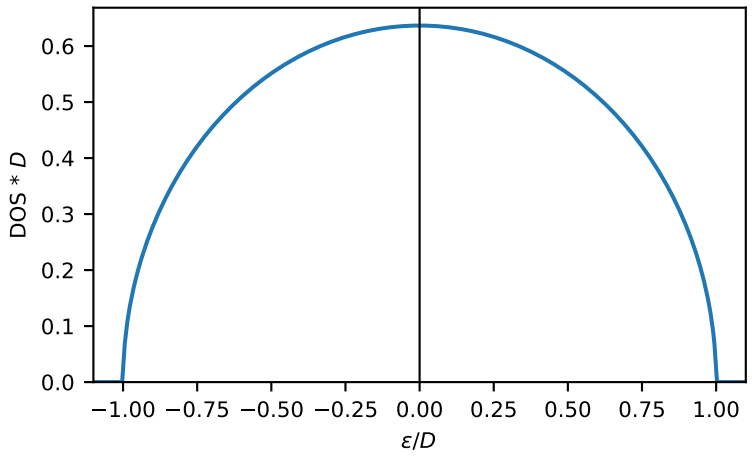
References

[economou2006]

Examples

```
>>> eps = np.linspace(-1.1, 1.1, num=500)
>>> dos = gt.lattice.bethe.dos(eps, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, dos)
>>> _ = plt.xlabel(r"$\epsilon/D$")
>>> _ = plt.ylabel(r"DOS * $D$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.ylim(bottom=0)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```



3.3.30 gftool.bethe_dos_moment

gftool.**bethe_dos_moment**(*m*, *half_bandwidth*)

Calculate the *m* th moment of the Bethe DOS.

The moments are defined as $\int d^m DOS()$.

Parameters

m
[int] The order of the moment.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice.

Returns

float
The *m* th moment of the Bethe DOS.

Raises

NotImplementedError
Currently only implemented for a few specific moments *m*.

See also:

gftool.lattice.bethe.dos

3.3.31 gftool.bethe_gf_z

gftool.**bethe_gf_z**(*z*, *half_bandwidth*)

Local Green's function of Bethe lattice for infinite coordination number.

$$G(z) = 2(z - s\sqrt{z^2 - D^2})/D^2$$

where D is the half bandwidth and $s = sgn[]$. See [georges1996].

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the Bethe Green's function.

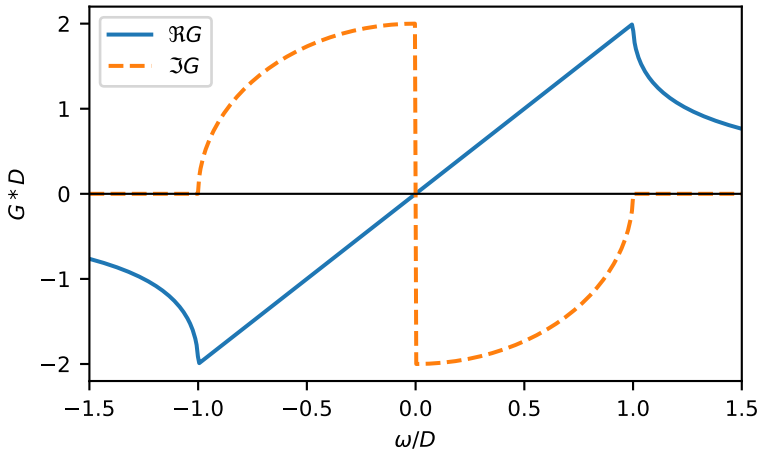
References

[georges1996]

Examples

```
>>> ww = np.linspace(-1.5, 1.5, num=500)
>>> gf_ww = gt.lattice.bethe.gf_z(ww, half_bandwidth=1)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(ww, gf_ww.real, label=r"$\Re G$")
>>> _ = plt.plot(ww, gf_ww.imag, '--', label=r"$\Im G$")
>>> _ = plt.xlabel(r"$\omega/D$")
>>> _ = plt.ylabel(r"$G*D$")
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=ww.min(), right=ww.max())
>>> _ = plt.legend()
>>> plt.show()
```



3.3.32 gftool.bethe_gf_d1_z

`gftool.`**`bethe_gf_d1_z`**(*z*, *half_bandwidth*)

First derivative of local Green's function of Bethe lattice for infinite coordination number.

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency *z*.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the derivative of the Green's function.

See also:

gftool.lattice.bethe.gf_z

3.3.33 gftool.bethe_gf_d2_z

gftool.**bethe_gf_d2_z**(*z*, *half_bandwidth*)

Second derivative of local Green's function of Bethe lattice for infinite coordination number.

Parameters

z
[complex array_like or complex] Green's function is evaluated at complex frequency z.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice. The *half_bandwidth* corresponds to the nearest neighbor hopping *t=D/2*.

Returns

complex np.ndarray or complex
Value of the Green's function.

See also:

gftool.lattice.bethe.gf_z

3.3.34 gftool.bethe_hilbert_transform

gftool.**bethe_hilbert_transform**(*xi*, *half_bandwidth*)

Hilbert transform of non-interacting DOS of the Bethe lattice.

The Hilbert transform is defined as:

$$\tilde{D}() = \int_{-\infty}^{\infty} d \frac{DOS()}{-}$$

The lattice Hilbert transform is the same as the non-interacting Green's function.

Parameters

xi
[complex array_like or complex] Point at which the Hilbert transform is evaluated.

half_bandwidth
[float] Half-bandwidth of the DOS of the Bethe lattice.

Returns

complex np.ndarray or complex
Hilbert transform of *xi*.

See also:

gftool.lattice.bethe.gf_z

Notes

Relation between nearest neighbor hopping t and half-bandwidth D:

$$2t = D$$

3.3.35 gftool.pole_gf_z

gftool.**pole_gf_z**(*z*, *poles*, *weights*)

Green's function given by a finite number of *poles*.

To be a Green's function, *np.sum(weights)* has to be 1 for the *1/z* tail or respectively the normalization.

Parameters

z
[(…) complex array_like] Green's function is evaluated at complex frequency z.

poles, weights
[(…, N) float array_like or float] The position and weight of the poles.

Returns

(…) complex np.ndarray
Green's function.

See also:

gf_d1_z
First derivative of the Green's function.

gf_tau
Corresponding fermionic imaginary time Green's function.

gt.pole_gf_tau_b
Corresponding bosonic imaginary time Green's function.

3.3.36 gftool.pole_gf_d1_z

gftool.**pole_gf_d1_z**(*z*, *poles*, *weights*)

First derivative of Green's function given by a finite number of *poles*.

To be a Green's function, *np.sum(weights)* has to be 1 for the 1/z tail.

Parameters

z
[(…) complex array_like] Green's function is evaluated at complex frequency z.

poles, weights
[(…, N) float array_like or float] The position and weight of the poles.

Returns

(…) complex np.ndarray
Derivative of the Green's function.

See also:

gf_z

3.3.37 gftool.pole_gf_moments

gftool.**pole_gf_moments**(*poles*, *weights*, *order*)

High-frequency moments of the pole Green's function.

Return the moments *mom* of the expansion $g(z) = \sum_m mom_m / z^m$ For the pole Green's function we have the simple relation $1/(z -) = \sum_{m=1}{}^{m-1}/z^m$.

Parameters

poles, weights
[(…, N) float np.ndarray] Position and weight of the poles.

order
[(…, M) int array_like] Order (degree) of the moments. *order* needs to be a positive integer.

Returns

(…, M) float np.ndarray
High-frequency moments.

3.3.38 gftool.pole_gf_ret_t

gftool.**pole_gf_ret_t**(*tt*, *poles*, *weights*)

Retarded time Green's function given by a finite number of *poles*.

Parameters

tt
[(…) float array_like] Green's function is evaluated at times *tt*, for *tt<0* it is 0.

poles, weights
[(…, N) float array_like or float] Position and weight of the poles.

Returns

(…) float np.ndarray
Retarded time Green's function.

See also:

pole_gf_z
Corresponding commutator Green's function.

3.3.39 gftool.pole_gf_tau

gftool.**pole_gf_tau**(*tau*, *poles*, *weights*, *beta*)

Imaginary time Green's function given by a finite number of *poles*.

Parameters

tau
[(…) float array_like] Green's function is evaluated at imaginary times *tau*. Only implemented for $\in [0,]$.

poles, weights
[(…, N) float array_like or float] Position and weight of the poles.

beta
[float] Inverse temperature.

Returns

(…) float np.ndarray
Imaginary time Green's function.

See also:

pole_gf_z
Corresponding commutator Green's function.

3.3.40 gftool.pole_gf_tau_b

gftool.**pole_gf_tau_b**(*tau*, *poles*, *weights*, *beta*)

Bosonic imaginary time Green's function given by a finite number of *poles*.

The bosonic Green's function is given by *G(tau) = -(1 + bose_fct(poles, beta))*exp(-poles*tau)*

Parameters

tau
[(…) float array_like] Green's function is evaluated at imaginary times *tau*. Only implemented for $\in [0,]$.

poles, weights
[(…, N) float array_like] Position and weight of the poles. The real part of the poles needs to be positive *poles.real > 0*.

beta
[float] Inverse temperature.

Returns

(…) float np.ndarray
Imaginary time Green's function.

Raises

ValueError
If any *poles.real <= 0*.

See also:

pole_gf_z
Corresponding commutator Green's function.

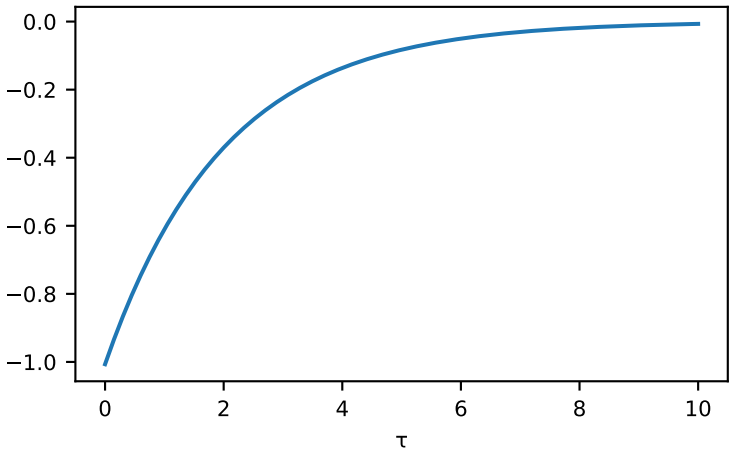
Examples

```
>>> beta = 10
>>> tau = np.linspace(0, beta, num=1000)
>>> gf_tau = gt.pole_gf_tau_b(tau, .5, 1., beta=beta)
```

The integrated imaginary time Green's function gives *-np.sum(weights/poles)*

```
>>> np.trapz(gf_tau, x=tau)
-2.0000041750107735
```

```
>>> import matplotlib.pyplot as plt
>>> __ = plt.plot(tau, gf_tau)
>>> __ = plt.xlabel('τ')
>>> plt.show()
```



3.3.41 gftool.hubbard_I_self_z

gftool.**hubbard_I_self_z**(*z*, *U*, *occ*)

Self-energy in Hubbard-I approximation (atomic solution).

The chemical potential and the onsite energy have to be included in *z*.

Parameters

z
[complex array_like] The complex frequencies at which the self-energy is evaluated. *z* should be shifted by the onsite energy and the chemical potential.

U
[float array_like] The local Hubbard interaction *U*.

occ
[float array_like] The occupation of the opposite spin as the spin of the self-energy.

Returns

complex array_like
The self-energy in Hubbard I approximation.

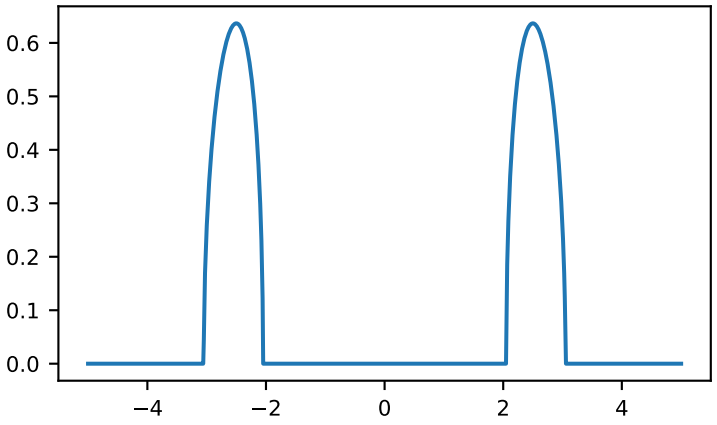
Examples

```
>>> U = 5
>>> mu = U/2  # particle-hole symmetric case -> n=0.5
>>> ww = np.linspace(-5, 5, num=1000) + 1e-6j
```

```
>>> self_ww = gt.hubbard_I_self_z(ww+mu, U, occ=0.5)
```

Show the spectral function for the Bethe lattice, we see the two Hubbard bands centered at ±U/2:

```
>>> import matplotlib.pyplot as plt
>>> gf_iw = gt.bethe_gf_z(ww+mu-self_ww, half_bandwidth=1.)
>>> __ = plt.plot(ww.real, -1./np.pi*gf_iw.imag)
>>> plt.show()
```



3.3.42 gftool.hubbard_dimer_gf_z

`gftool.`**`hubbard_dimer_gf_z`**(*z*, *hopping*, *interaction*, *kind='+'*)

Green's function for the two site Hubbard model on a *dimer*.

The Hamilton is given

$$H = -t\sum(c_1^\dagger c_2 + c_2^\dagger c_1) + U\sum_i n_{i\uparrow} n_{i\downarrow}$$

with the *hopping* t and the *interaction* U. The Green's function is given for the operators $c_\pm = 1/\sqrt{2}(c_1 \pm c_2)$, where ± is given by *kind*

Parameters

z
[complex ndarray or complex] Green's function is evaluated at complex frequency z.

hopping
[float] The hopping parameter between the sites of the dimer.

interaction
[float] The Hubbard interaction strength for the on-site interaction.

kind
[{'+', '-'}] The operator for which the Green's function is calculated.

Returns

complex ndarray
Value of the Hubbard dimer Green's function at frequencies z.

Notes

The solution is obtained by exact digitalization and shown in [eder2017].

References

[eder2017]

3.3.43 gftool.surface_gf_zeps

`gftool.surface_gf_zeps(z, eps, hopping_nn)`

Surface Green's function for stacked layers.

$$\left(1 - \sqrt{1 - 4t^2 g_{00}^2}\right) / (2t^2 g_{00})$$

with $g_{00} = (z -)^{-1}$ [odashima2016]. This is in principle the Green's function for a semi-infinite chain.

Parameters

z
[complex] Green's function is evaluated at complex frequency z.

eps
[float] Eigenenergy (dispersion) for which the Green's function is evaluated.

hopping_nn
[float] Nearest neighbor hopping t between neighboring layers.

Returns

complex
Value of the surface Green's function.

References

[odashima2016]

3.4 Statistics and particle numbers

fermi_fct(eps, beta)	Return the Fermi function *1/(exp(βϵ)+1)*.
fermi_fct_d1(eps, beta)	Return the 1st derivative of the Fermi function.
fermi_fct_inv(fermi, beta)	Inverse of the Fermi function.
bose_fct(eps, beta)	Return the Bose function *1/(exp(βϵ)-1)*.
matsubara_frequencies(n_points, beta)	Return *fermionic* Matsubara frequencies i_n for the points *n_points*.
matsubara_frequencies_b(n_points, beta)	Return *bosonic* Matsubara frequencies i_n for the points *n_points*.
pade_frequencies(num, beta)	Return *num fermionic* Padé frequencies iz_p.
density_iw(iws, gf_iw, beta[, weights, ...])	Calculate the number density of the Green's function *gf_iw* at finite temperature *beta*.
chemical_potential(occ_root[, mu0, step0])	Search chemical potential for a given occupation.
density(gf_iw, potential, beta[, ...])	Calculate the number density of the Green's function *gf_iw* at finite temperature *beta*.
density_error(delta_gf_iw, iw_n[, noisy])	Return an estimate for the upper bound of the error in the density.
density_error2(delta_gf_iw, iw_n)	Return an estimate for the upper bound of the error in the density.
check_convergence(gf_iw, potential, beta[, ...])	Return data for visual inspection of the density error.

3.4.1 gftool.fermi_fct

gftool.**fermi_fct**(*eps*, *beta*)

Return the Fermi function *1/(exp(βϵ)+1)*.

For complex inputs the function is not as accurate as for real inputs.

Parameters

eps
[complex or float or np.ndarray] The energy at which the Fermi function is evaluated.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

complex of float or np.ndarray
The Fermi function, same type as eps.

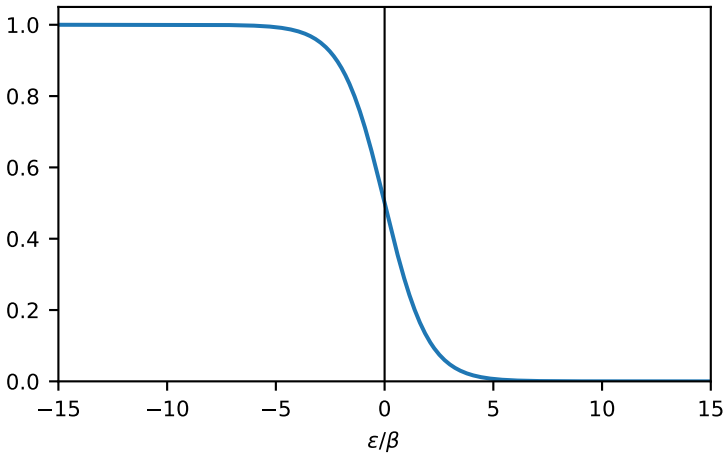
See also:

fermi_fct_inv
The inverse of the Fermi function for real arguments.

Examples

```
>>> eps = np.linspace(-15, 15, num=501)
>>> fermi = gt.fermi_fct(eps, beta=1.0)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, fermi)
>>> _ = plt.xlabel(r"$\epsilon/\beta$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> _ = plt.ylim(bottom=0)
>>> plt.show()
```



3.4.2 gftool.fermi_fct_d1

gftool.**fermi_fct_d1**(*eps*, *beta*)

Return the 1st derivative of the Fermi function.

$$f'() = -\exp()/[\exp() + 1]^2 = -f()[1 - f()]$$

Parameters

eps
[float or float np.ndarray] The energy at which the Fermi function is evaluated.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

float or float np.ndarray
The Fermi function, same type as eps.

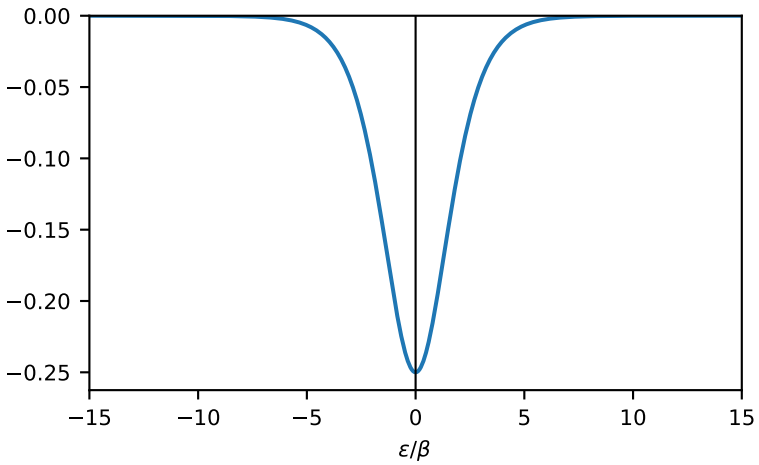
See also:

fermi_fct

Examples

```
>>> eps = np.linspace(-15, 15, num=501)
>>> fermi_d1 = gt.fermi_fct_d1(eps, beta=1.0)
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, fermi_d1)
>>> _ = plt.xlabel(r"$\epsilon/\beta$")
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> _ = plt.ylim(top=0)
>>> plt.show()
```



3.4.3 gftool.fermi_fct_inv

gftool.**fermi_fct_inv**(*fermi*, *beta*)

Inverse of the Fermi function.

This is e.g. useful for integrals over the derivative of the Fermi function.

Parameters

fermi
[float or float np.ndarray] The values of the Fermi function.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

float or float np.ndarray
The inverse of the Fermi function *fermi_fct(fermi_fct_inv, beta)=fermi*.

See also:

fermi_fct

Examples

```
>>> eps = np.linspace(-15, 15, num=500)
>>> fermi = gt.fermi_fct(eps, beta=1)
>>> np.allclose(eps, gt.fermi_fct_inv(fermi, beta=1))
True
```

3.4.4 gftool.bose_fct

gftool.**bose_fct**(*eps*, *beta*)

Return the Bose function *1/(exp(βϵ)-1)*.

Parameters

eps
[complex or float or array_like] The energy at which the Bose function is evaluated.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

complex of float or np.ndarray
The Bose function, same type as eps.

Examples

```
>>> eps = np.linspace(-1.5, 1.5, num=501)
>>> bose = gt.bose_fct(eps, beta=1.0)
```

The Bose function diverges at *eps=0*:

```
>>> bose[eps==0]
array([inf])
```

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(eps, bose)
>>> _ = plt.xlabel(r"$\epsilon/\beta$")
>>> _ = plt.axhline(0, color='black', linewidth=0.8)
>>> _ = plt.axvline(0, color='black', linewidth=0.8)
>>> _ = plt.xlim(left=eps.min(), right=eps.max())
>>> plt.show()
```

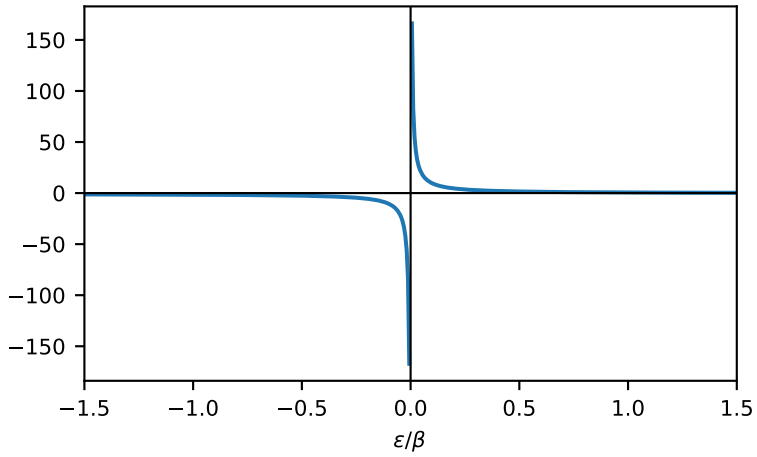
3.4.5 gftool.matsubara_frequencies

gftool.**matsubara_frequencies**(*n_points*, *beta*)

Return *fermionic* Matsubara frequencies i_n for the points *n_points*.

Parameters

n_points
[int array_like] Points for which the Matsubara frequencies i_n are returned.



beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

complex np.ndarray
Array of the imaginary Matsubara frequencies.

Examples

```
>>> gt.matsubara_frequencies(range(1024), beta=1)
array([0.+3.14159265e+00j, 0.+9.42477796e+00j, 0.+1.57079633e+01j, ...,
       0.+6.41827379e+03j, 0.+6.42455698e+03j, 0.+6.43084016e+03j])
```

3.4.6 gftool.matsubara_frequencies_b

gftool.**matsubara_frequencies_b**(*n_points*, *beta*)

Return *bosonic* Matsubara frequencies i_n for the points *n_points*.

Parameters

n_points
[int array_like] Points for which the Matsubara frequencies i_n are returned.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

complex np.ndarray
Array of the imaginary Matsubara frequencies.

Examples

```
>>> gt.matsubara_frequencies_b(range(1024), beta=1)
array([0.+0.00000000e+00j, 0.+6.28318531e+00j, 0.+1.25663706e+01j, ...,
       0.+6.41513220e+03j, 0.+6.42141538e+03j, 0.+6.42769857e+03j])
```

3.4.7 gftool.pade_frequencies

gftool.**pade_frequencies**(*num: int*, *beta*)

Return *num fermionic* Padé frequencies iz_p.

The Padé frequencies are the poles of the approximation of the Fermi function with *2*num* poles [ozaki2007]. This gives an non-equidistant mesh on the imaginary axis.

Parameters

num
[int] Number of positive Padé frequencies.

beta
[float] The inverse temperature $beta = 1/k_B T$.

Returns

izp
[(num) complex np.ndarray] Positive Padé frequencies.

resids
[(num) float np.ndarray] Residue of the Fermi function corresponding to *izp*. The residue is given relative to the true residue of the Fermi function *-1/beta* corresponding to the poles at Matsubara frequencies. This allows to use Padé frequencies as drop-in replacement. The actual residues are *-resids/beta*.

References

[ozaki2007], [hu2010]

Examples

Comparing Padé frequency to Matsubara frequencies:

```
>>> izp, rp = gt.pade_frequencies(5, beta=1)
>>> izp.imag
array([ 3.14159265,  9.42478813, 15.76218003, 24.87650795, 70.52670981])
>>> gt.matsubara_frequencies(range(5), beta=1).imag
array([ 3.14159265,  9.42477796, 15.70796327, 21.99114858, 28.27433388])
```

Relative residue:

```
>>> rp
array([ 1.        ,  1.00002021,  1.04839303,  2.32178225, 22.12980451])
```

3.4.8 gftool.density_iw

gftool.**density_iw**(*iws*, *gf_iw*, *beta*, *weights=1.0*, *moments=(1.0,)*, *n_fit=0*)

Calculate the number density of the Green's function *gf_iw* at finite temperature *beta*.

This function can be used for fermionic Matsubara frequencies *matsubara_frequencies*, as well as fermionic Padé frequencies *pade_frequencies*.

Parameters

iws, gf_iw
[(…, N_iw) complex np.ndarray] Positive Matsubara frequencies i_n (or Padé iz_p) and the Green's function at these frequencies.

beta
[float] The inverse temperature $beta = 1/k_B T$.

weights
[(…, N_iw) float np.ndarray, optional] Residues of the frequencies with respect to the residues of the Matsubara frequencies *1/beta* (default: 1.0). For Padé frequencies this needs to be provided.

moments
[(…, M) float array_like, optional] Moments of the high-frequency expansion, where *G(z) = np.sum(moments / z**np.arange(N))* for large z.

n_fit
[int, optional] Number of additionally to *moments* fitted moments. If Padé frequencies are used, this is typically not necessary (default: 0).

Returns

float
The number density of the given Green's function *gf_iw*.

See also:

matsubara_frequencies
Method generating Matsubara frequencies *iws*.

pade_frequencies
Method generating Padé frequencies *iws* with *weights*.

Examples

```
>>> BETA = 50
>>> iws = gt.matsubara_frequencies(range(1024), beta=BETA)
```

Example Green's function

```
>>> np.random.seed(0)  # to have deterministic results
>>> poles = 2*np.random.random(10) - 1  # partially filled
>>> residues = np.random.random(10); residues = residues / np.sum(residues)
>>> pole_gf = gt.basis.PoleGf(poles=poles, residues=residues)
>>> gf_iw = pole_gf.eval_z(iws)
>>> exact = pole_gf.occ(BETA)
>>> exact
0.17858151698239388
```

Numerical calculation of the occupation number, using Matsubara frequency

```
>>> occ = gt.density_iw(iws, gf_iw, beta=BETA)
>>> m2 = pole_gf.moments([1, 2])  # additional high-frequency moment
>>> m2
array([1.        , 0.30839757])
>>> occ_m2 = gt.density_iw(iws, gf_iw, beta=BETA, moments=m2)
>>> occ_fit2 = gt.density_iw(iws, gf_iw, beta=BETA, n_fit=1)
>>> exact, occ, occ_m2, occ_fit2
(0.17858151..., 0.17934437..., 0.17858150..., 0.17858198...)
>>> abs(occ - exact), abs(occ_m2 - exact), abs(occ_fit2 - exact)
(0.00076286..., 8.18...e-09, 4.72...e-07)
```

using more accurate Padé frequencies

```
>>> izp, rp = gt.pade_frequencies(100, beta=BETA)
>>> gf_izp = pole_gf.eval_z(izp)
>>> occ_izp = gt.density_iw(izp, gf_izp, beta=BETA, weights=rp)
>>> occ_izp
0.17858151...
>>> abs(occ_izp - exact) < 1e-12
True
```

3.4.9 gftool.chemical_potential

gftool.**chemical_potential**(*occ_root: Callable[[float], float]*, *mu0=0.0*, *step0=1.0*, ***kwds*) → float

Search chemical potential for a given occupation.

Parameters

occ_root
[callable] Function *occ_root(mu_i) -> occ_i - occ*, which returns the difference in occupation to the desired occupation *occ* for a chemical potential *mu_i*. Note that the sign is important, *occ_i - occ* has to be returned.

mu0
[float, optional] The starting guess for the chemical potential (default: 0).

step0
[float, optional] Starting step-width for the bracket search. A reasonable guess is of the order of the band-width (default: 1).

****kwds**
Additional keyword arguments passed to `scipy.optimize.root_scalar`. Common arguments might be *xtol* or *rtol* for absolute or relative tolerance.

Returns

float
The chemical potential given the correct charge *occ_root(mu)=0*.

Raises

RuntimeError
If either no bracket can be found (this should only happen for the complete empty or completely filled case), or if the scalar root search in the bracket fails.

Notes

The search for a chemical potential is a two-step procedure: *First*, we search for a bracket *[mua, mub]* with *occ_root(mua) < 0 < occ_root(mub)*. We use that the occupation is a monotonous increasing function of the chemical potential *mu*. *Second*, we perform a standard root-search in *[mua, mub]* which is done using `scipy.optimize.root_scalar`, Brent's method is currently used as default.

Examples

We search for the occupation of a simple 3-level system, where the occupation of each level is simply given by the Fermi function:

```
>>> occ = 1.67  # desired total occupation
>>> BETA = 100  # inverse temperature
>>> eps = np.random.random(3)
>>> def occ_fct(mu):
...     return gt.fermi_fct(eps - mu, beta=BETA).sum()
>>> mu = gt.chemical_potential(lambda mu: occ_fct(mu) - occ)
>>> occ_fct(mu), occ
(1.67000..., 1.67)
```

3.4.10 gftool.density

`gftool.`**`density`**(*gf_iw*, *potential*, *beta*, *return_err=True*, *matrix=False*, *total=False*)

Calculate the number density of the Green's function *gf_iw* at finite temperature *beta*.

Deprecated since version 0.8.0: Mostly superseded by more flexible `density_iw`, thus this function will likely be discontinued. Currently `density` is a little more accurate for *matrix=True*, compared to `density_iw` without using fitting.

As Green's functions decay only as $1/$, the known part of the form $1/(i_n + - - _{\text{static}})$ will be calculated analytically. $_{\text{static}}$ is the ω-independent mean-field part of the self-energy.

Parameters

gf_iw
: [complex ndarray] The Matsubara frequency Green's function for positive frequencies i_n. The last axis corresponds to the Matsubara frequencies.

potential
: [float ndarray or float] The static potential for the large-ω behavior of the Green's function. It is the real constant $- - _{\text{static}}$. The shape must agree with *gf_iw* without the last axis. If `matrix`, then potential needs to be a (N, N) matrix. It is the negative of the Hamiltonian matrix and thus needs to be hermitian.

beta
: [float] The inverse temperature *beta* = 1/T.

return_err
: [bool or float, optional] If *True* (default), the error estimate will be returned along with the density. If *return_err* is a float, a warning will Warning will be issued if the error estimate is larger than *return_err*. If *False*, no error estimate is calculated. See `density_error` for description of the error estimate.

matrix
: [bool, optional] Whether the given *potential* is a matrix (default: False).

total
[bool or tuple] If *total* the total density (summed over all dimensions) is returned. Also a tuple can be given, over which axes the sums is taken.

Returns

x
[float] The number density of the given Green's function *gf_iw*.

err
[float] An estimate for the density error. Only returned if *return_err* is *True*.

Notes

The number density can be obtained from the Matsubara frequency Green's function using

$$\langle n \rangle = \lim_{\nearrow 0} G(\tau = -\tau) = 1/\sum_{n=-\infty}^{\infty} G(i_n)$$

As Green's functions decay only as $O(1/\tau)$, truncation of the summation yields a non-vanishing contribution of the tail. For the analytic structure of the Green's function see [2], [3]. To take this into consideration the known part of the form $1/(i_n + \mu - \tau_{\text{static}})$ will be calculated analytically. This yields [1]

$$\begin{aligned}\langle n \rangle = 1/\sum_{n=-\infty}^{\infty} [G(i_n) - 1/(i_n + \mu - \tau_{\text{static}})] \\ + 1/2 + 1/2 \tanh[1/2(\mu - \tau_{\text{static}})].\end{aligned}$$

We can use the symmetry $G(z*) = G^*(z)$ do reduce the sum only over positive Matsubara frequencies

$$\begin{aligned}\sum_{n=-\infty}^{\infty} G(i_n) &= \sum_{n=-\infty}^{-1} G(i_n) + \sum_{n=0}^{n=\infty} G(i_n) \\ &= \sum_{n=0}^{\infty} [G(-i_n) + G(i_n)] \\ &= 2\sum_{n=0}^{\infty} G(i_n).\end{aligned}$$

Thus we get the final expression

$$\begin{aligned}\langle n \rangle = 2/\sum_{n \geq 0} [G(i_n) - 1/(i_n + \mu - \tau_{\text{static}})] \\ + 1/2 + 1/2 \tanh[1/2(\mu - \tau_{\text{static}})].\end{aligned}$$

References

[1], [2], [3]

3.4.11 gftool.density_error

`gftool.density_error(delta_gf_iw, iw_n, noisy=True)`

Return an estimate for the upper bound of the error in the density.

This estimate is based on the *integral test*. The crucial assumption is, that *ω_N* is large enough, such that $G \sim 1/n^2$ for all larger n. If this criteria is not met, the error estimate is unreasonable and can **not** be trusted. If the error is of the same magnitude as the density itself, the behavior of the variable *factor* should be checked.

Parameters

delta_gf_iw
[(..., N) ndarray] The difference between the Green's function $G(i_n)$ and the non-interacting high-frequency estimate. Only it's real part is needed.

iw_n
[(N) complex ndarray] The Matsubara frequencies corresponding to *delta_gf_iw*.

noisy
[bool, optional] Whether the input *delta_gf_iw* contains noise (default: True). If *noisy*, an average over the highest frequency is taken to estimate the error.

Returns

float
The estimate of the upper bound of the error. Reliable only for large enough Matsubara frequencies.

3.4.12 gftool.density_error2

`gftool.`**`density_error2`**(*delta_gf_iw*, *iw_n*)

Return an estimate for the upper bound of the error in the density.

This estimate is based on the *integral test*. The crucial assumption is, that *ω_N* is large enough, such that $G \sim 1/n^3$ for all larger n. If this criteria is not met, the error estimate is unreasonable and can **not** be trusted. If the error is of the same magnitude as the density itself, the behavior of the variable *factor* should be checked.

Parameters

delta_gf_iw
[(…, N) ndarray] The difference between the Green's function $G(i_n)$ and the non-interacting high-frequency estimate. Only it's real part is needed.

iw_n
[(N) complex ndarray] The Matsubara frequencies corresponding to *delta_gf_iw*.

Returns

float
The estimate of the upper bound of the error. Reliable only for large enough Matsubara frequencies.

3.4.13 gftool.check_convergence

`gftool.`**`check_convergence`**(*gf_iw*, *potential*, *beta*, *order=2*, *matrix=False*, *total=False*)

Return data for visual inspection of the density error.

The calculation of the density error assumes that *sufficient* Matsubara frequencies were used. Sufficient means here, that the reminder G does **not** grow anymore. If the error estimate is small, but `check_convergence` returns rapidly growing data, the number of Matsubara frequencies is not sufficient

See `density` for parameters.

Returns

float ndarray
The last dimension of `check_convergence` corresponds to the Matsubara frequencies.

Other Parameters

order
[int] The assumed order of the first non-vanishing term of the Laurent expansion.

3.5 Utilities

Functions

get_versions()	Get version information or return default if unable to do so.

3.5.1 gftool.get_versions

gftool.**get_versions**() → Dict[str, Any]

Get version information or return default if unable to do so.

Classes

Result(x, err)

3.5.2 gftool.Result

class gftool.**Result**(*x*, *err*)

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

gftool.Result.__init__

Result.**__init__**(**args*, ***kwargs*)

gftool.Result.count

Result.**count**(*value*, */*)

Return number of occurrences of value.

gftool.Result.index

Result.**index**(*value*, *start=0*, *stop=sys.maxsize*, /)

Return first index of value.

Raises ValueError if the value is not present.

Attributes

err	Alias for field number 1
x	Alias for field number 0

gftool.Result.err

property Result.**err**

Alias for field number 1

gftool.Result.x

property Result.**x**

Alias for field number 0

CHAPTER

FOUR

WHAT'S NEW

4.1 unreleased

4.1.1 Bug fixes

- fix vectorization of `linearprediction` functions

4.2 0.11.0 (2022-04-29)

4.2.1 New Features

- Add pole-base Padé analytic continuation `polepade` (41d57537).
  - Allows determining number of poles avoiding overfitting.
  - Least-squares based approach allowing to include uncertainties of input data.
- Add `linearprediction` module (b1c8f636).
  - Linear prediction can be used to extend retarded-time Green's functions.
- Add Padé-Fourier approach to Laplace transform (fe1ac173).
  - Padé-Fourier allows to significantly reduce the truncation error. This allows for contours closer to real-axis for a given maximal time.
  - Linear Padé approximant `tt2z_pade` based on simple poles
  - Quadratic Hermite-Padé approximant `tt2z_herm2` including quadratic branch cuts but introducing ambiguity which branch to choose.
  - Module `hermpade` implements the necessary approximants.
- Add lattice `box` with box-shaped DOS (09974a09).

4.2.2 Internal improvements

- Use `numpy.testing.assert_allclose` for tests, providing more verbose output (dbb8fd7c).

4.2.3 Documentation

- Start to adhere more closely to `numpydoc` (40d57d45).

4.3 0.10.1 (2021-12-01)

4.3.1 New Features

- Officially support Python 3.10
- Add retarded-time Bethe Green's *gf_ret_t* function (6ffc7c91)

4.3.2 Internal improvements

- Switch from Travis to GitHub actions #20 (23ba0a34)
  - This adds test for Mac and Windows

4.3.3 Documentation

- Fix various errors using Velin (03ff6c8e)

4.3.4 Bug fixes

- Accept singular constrains in *lstsq_ec* (167e7886)
- Accurately calculate *gftool.lattice.sc.dos* around `eps=0` (5693184e). Previously, DOS was incorrect for tiny values, e.g. `eps=1e-16`.

4.4 0.10.0 (2021-09-19)

4.4.1 Breaking Changes

- Drop support for Python 3.6, **minimal version** is now **3.7**
- Content of *gftool.matrix* was renamed more appropriately:
  - *xi* of *Decomposition* is now *eig*, as it contains the eigenvalues
  - New functions *decompose_mat* for general matrices, *decompose_sym* for complex symmetric matrices, and *decompose_her* for Hermitian matrices.

4.4.2 Depreciations

- Deprecate the `matrix` functions `decompose_gf`, `decompose_hamiltonian`, `from_gf`, and `from_hamiltonian`.

4.4.3 Documentation

- New index page independent of README, separated *Getting started* page.
- Improve *Tutorial* and `gftool.matrix`
- Generate PDF documentation on ReadTheDocs (3122e1ba)

4.4.4 Internal improvements

- Use eigendecomposition instead of SVD in `gftool.beb` (0475c110)
- Drop slow `asfortranarray` in `gftool.matrix` (4865cc05)
- Use *pre-commit* (6f4028d3)

4.5 0.9.1 (2021-06-01)

4.5.1 Bug fixes

CPA:

- return scalar *mu* in `gftool.cpa.solve_fxdocc_root` (10fae4d)
- find *mu* more reliably

4.5.2 Other New Features

- SIAM: add greater and lesser Green's functions `gf0_loc_gr_t` and `gf0_loc_le_t` (ea541f3)

4.6 0.9.0 (2021-05-09)

4.6.1 New Features

Implement `cpa` and `beb` to treat disorder (c3bad20c)

4.7 0.8.1 (2021-04-25)

4.7.1 New Features

The 3D cubic lattices were added:

- body-centered cubic `gftool.lattice.bcc` (406acef8)
- face-centered cubic `gftool.lattice.fcc` (ddd559cb)

4.8 0.8.0 (2021-04-17)

4.8.1 New Features

The `gftool.lattice` module was extended, especially regarding two-dimensional lattice. There were also some enhancements, given DOS moments are now up to order 20, and they should be accurate to machine precision.

The following lattices where added with full interface:

- Simple cubic: `gftool.lattice.sc` (4e3021) by Andreas Östlin
- Honeycomb: `gftool.lattice.honeycomb` (7aa3133)
- Triangular: `gftool.lattice.triangular` (c56f33e)

Local Green's function and DOS is now also available for the following lattices:

- Lieb: `gftool.lattice.lieb` (c76e948)
- Kagome: `gftool.lattice.kagome` (28a41c0)
- Bethe lattice with general coordination: `gftool.lattice.bethez` (2648cf4)
- Rectangular: `gftool.lattice.rectangular`

4.8.2 Other New Features

- add retarded time Green's function give by its poles `gftool.pole_gf_ret_t`
- added `gftool.siam` module with some basics for the non-interacting siam

4.8.3 Depreciations

- `gftool.density` is deprecated and will likely be discontinued. Consider the more flexible `gftool.density_iw` instead.

4.8.4 Documentation

- Button to toggle the prompt (>>>) was added (46b6f39)

4.8.5 Internal improvements

- Ensure more accurate `numpy.sum` using partial pairwise summation for generalized ufuncs (2d3baef)

4.9 0.7.0 (2020-10-18)

4.9.1 Breaking Changes

- The *gftool.pade* module had a minor rework. The behavior of filters changed. Future breaking changes are to be expected, the module is not well-structured.

4.9.2 New Features

- add *gftool.lattice.onedim* for Green's function of one-dimensional lattice
- add fitting of high-frequency moment to *gftool.fourier.iw2tau* (e2c92e2)

4.9.3 Other New Features

- add *gftool.pade_frequencies* (9f492fc)
- add *gftool.density_iw* function as common interface to calculate occupation number from Matsubara or Padé frequencies
- allow calculation of *gftool.lattice.bethe* for Bethe lattice at complex points (note, that this is probably not a physically meaningful quantity) (ccbac7b)
- add stress tensor transformation *gftool.lattice.square.stress_trafo* for 2D (528fb21)

4.9.4 Bug fixes

- Fix constant in *gftool.fourier.tau2iw_ft_lin* (e2163e3). This error most likely didn't significantly affect any results for a reasonable number of tau-points.
- *gftool.density* should work now with gu-style matrices (4deffdf)

4.9.5 Documentation

- Functions exposed at the top level (*gftool*) should now properly appear in the documentation.

4.10 0.6.1

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[blackman1971] Blackman, J.A., Esterling, D.M., Berk, N.F., 1971. Generalized Locator—Coherent-Potential Approach to Binary Alloys. Phys. Rev. B 4, 2412–2428. https://doi.org/10.1103/PhysRevB.4.2412

[weh2021] Weh, A., Zhang, Y., Östlin, A., Terletska, H., Bauernfeind, D., Tam, K.-M., Evertz, H.G., Byczuk, K., Vollhardt, D., Chioncel, L., 2021. Dynamical mean-field theory of the Anderson–Hubbard model with local and nonlocal disorder in tensor formulation. Phys. Rev. B 104, 045127. https://doi.org/10.1103/PhysRevB.104.045127

[filon1930] Filon, L. N. G. III.—On a Quadrature Formula for Trigonometric Integrals. Proc. Roy. Soc. Edinburgh 49, 38–47 (1930). https://doi.org/10.1017/S0370164600026262

[iserles2006] Iserles, A., Nørsett, S. P. & Olver, S. Highly Oscillatory Quadrature: The Story so Far. in Numerical Mathematics and Advanced Applications (eds. de Castro, A. B., Gómez, D., Quintela, P. & Salgado, P.) 97–118 (Springer, 2006). https://doi.org/10.1007/978-3-540-34288-5_6 http://www.sam.math.ethz.ch/~hiptmair/Seminars/OSCINT/INO06.pdf

[filon1930] Filon, L. N. G. III.—On a Quadrature Formula for Trigonometric Integrals. Proc. Roy. Soc. Edinburgh 49, 38–47 (1930). https://doi.org/10.1017/S0370164600026262

[iserles2006] Iserles, A., Nørsett, S. P. & Olver, S. Highly Oscillatory Quadrature: The Story so Far. in Numerical Mathematics and Advanced Applications (eds. de Castro, A. B., Gómez, D., Quintela, P. & Salgado, P.) 97–118 (Springer, 2006). https://doi.org/10.1007/978-3-540-34288-5_6 http://www.sam.math.ethz.ch/~hiptmair/Seminars/OSCINT/INO06.pdf

[filon1930] Filon, L. N. G. III.—On a Quadrature Formula for Trigonometric Integrals. Proc. Roy. Soc. Edinburgh 49, 38–47 (1930). https://doi.org/10.1017/S0370164600026262

[iserles2006] Iserles, A., Nørsett, S. P. & Olver, S. Highly Oscillatory Quadrature: The Story so Far. in Numerical Mathematics and Advanced Applications (eds. de Castro, A. B., Gómez, D., Quintela, P. & Salgado, P.) 97–118 (Springer, 2006). https://doi.org/10.1007/978-3-540-34288-5_6 http://www.sam.math.ethz.ch/~hiptmair/Seminars/OSCINT/INO06.pdf

[fasondini2019] Fasondini, M., Hale, N., Spoerer, R. & Weideman, J. A. C. Quadratic Padé Approximation: Numerical Aspects and Applications. Computer research and modeling 11, 1017–1031 (2019). https://doi.org/10.20537/2076-7633-2019-11-6-1017-1031

[baker1996] Baker Jr, G. A. & Graves-Morris, Pade Approximants. Second edition. (Cambridge University Press, 1996).

[gonnet2013] 1.Gonnet, P., Güttel, S. & Trefethen, L. N. Robust Padé Approximation via SVD. SIAM Rev. 55, 101–117 (2013). https://doi.org/10.1137/110853236

[fasondini2019] Fasondini, M., Hale, N., Spoerer, R. & Weideman, J. A. C. Quadratic Padé Approximation: Numerical Aspects and Applications. Computer research and modeling 11, 1017–1031 (2019). https://doi.org/10.20537/2076-7633-2019-11-6-1017-1031

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[georges1996] Georges et al., Rev. Mod. Phys. 68, 13 (1996) https://doi.org/10.1103/RevModPhys.68.13

[georges1996] Georges et al., Rev. Mod. Phys. 68, 13 (1996) https://doi.org/10.1103/RevModPhys.68.13

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[arsenault2013] Arsenault, L.-F., Tremblay, A.-M.S., 2013. Transport functions for hypercubic and Bethe lattices. Phys. Rev. B 88, 205109. https://doi.org/10.1103/PhysRevB.88.205109

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[kogan2021] Kogan, E., Gumbs, G., 2020. Green's Functions and DOS for Some 2D Lattices. Graphene 10, 1–12. https://doi.org/10.4236/graphene.2021.101001

[kogan2021] Kogan, E., Gumbs, G., 2020. Green's Functions and DOS for Some 2D Lattices. Graphene 10, 1–12. https://doi.org/10.4236/graphene.2021.101001

[kogan2021] Kogan, E., Gumbs, G., 2020. Green's Functions and DOS for Some 2D Lattices. Graphene 10, 1–12. https://doi.org/10.4236/graphene.2021.101001

[kogan2021] Kogan, E. and Gumbs, G. (2021) Green's Functions and DOS for Some 2D Lattices. Graphene, 10, 1-12. https://doi.org/10.4236/graphene.2021.101001.

[kogan2021] Kogan, E. and Gumbs, G. (2021) Green's Functions and DOS for Some 2D Lattices. Graphene, 10, 1-12. https://doi.org/10.4236/graphene.2021.101001.

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[varm2013] Varma, V.K., Monien, H., 2013. Lattice Green's functions for kagome, diced, and hyperkagome lattices. Phys. Rev. E 87, 032109. https://doi.org/10.1103/PhysRevE.87.032109

[kogan2021] Kogan, E., Gumbs, G., 2020. Green's Functions and DOS for Some 2D Lattices. Graphene 10, 1–12. https://doi.org/10.4236/graphene.2021.101001

[varm2013] Varma, V.K., Monien, H., 2013. Lattice Green's functions for kagome, diced, and hyperkagome lattices. Phys. Rev. E 87, 032109. https://doi.org/10.1103/PhysRevE.87.032109

[kogan2021] Kogan, E., Gumbs, G., 2020. Green's Functions and DOS for Some 2D Lattices. Graphene 10, 1–12. https://doi.org/10.4236/graphene.2021.101001

[varm2013] Varma, V.K., Monien, H., 2013. Lattice Green's functions for kagome, diced, and hyperkagome lattices. Phys. Rev. E 87, 032109. https://doi.org/10.1103/PhysRevE.87.032109

[kogan2021] Kogan, E., Gumbs, G., 2020. Green's Functions and DOS for Some 2D Lattices. Graphene 10, 1–12. https://doi.org/10.4236/graphene.2021.101001

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[joyce1973] G. S. Joyce, Phil. Trans. of the Royal Society of London A, 273, 583 (1973). https://www.jstor.org/stable/74037

[katsura1971] S. Katsura et al., J. Math. Phys., 12, 895 (1971). https://doi.org/10.1063/1.1665663

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[joyce1973] G. S. Joyce, Phil. Trans. of the Royal Society of London A, 273, 583 (1973). https://www.jstor.org/stable/74037

[katsura1971] S. Katsura et al., J. Math. Phys., 12, 895 (1971). https://doi.org/10.1063/1.1665663

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[delves2001] Delves, R. T. and Joyce, G. S., Ann. Phys. 291, 71 (2001). https://doi.org/10.1006/aphy.2001.6148

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[delves2001] Delves, R. T. and Joyce, G. S., Ann. Phys. 291, 71 (2001). https://doi.org/10.1006/aphy.2001.6148

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[golub2013] Golub, Gene H., und Charles F. Van Loan. Matrix Computations. JHU Press, 2013.

[Vaidyanathan2007] Vaidyanathan, P.P., 2007. The Theory of Linear Prediction. Synthesis Lectures on Signal Processing 2, 1–184. https://doi.org/10.2200/S00086ED1V01Y200712SPR003

[Makhoul1975] Makhoul, J. Linear prediction: A tutorial review. Proceedings of the IEEE 63, 561–580 (1975). https://doi.org/10.1109/PROC.1975.9792

[1] R. A. Horn & C. R. Johnson, *Matrix Analysis*. Cambridge, UK: Cambridge University Press, 1999, pp. 146-7.

[Kay1988] Kay, S.M., 1988. Modern spectral estimation: theory and application. Pearson Education India.

[noble2017] Noble, J.H., Lubasch, M., Stevens, J., Jentschura, U.D., 2017. Diagonalization of complex symmetric matrices: Generalized Householder reflections, iterative deflation and implicit shifts. Computer Physics Communications 221, 304–316. https://doi.org/10.1016/j.cpc.2017.06.014

[1] Schött et al. "Analytic Continuation by Averaging Padé Approximants". Phys Rev B 93, no. 7 (2016): 075104. https://doi.org/10.1103/PhysRevB.93.075104.

[2] Vidberg, H. J., and J. W. Serene. "Solving the Eliashberg Equations by Means of N-Point Padé Approximants." Journal of Low Temperature Physics 29, no. 3-4 (November 1, 1977): 179-92. https://doi.org/10.1007/BF00655090.

[ito2018] Ito, S., Nakatsukasa, Y., 2018. Stable polefinding and rational least-squares fitting via eigenvalues. Numer. Math. 139, 633–682. https://doi.org/10.1007/s00211-018-0948-4

[weh2020] Weh, A. et al. Spectral properties of heterostructures containing half-metallic ferromagnets in the presence of local many-body correlations. Phys. Rev. Research 2, 043263 (2020). https://doi.org/10.1103/PhysRevResearch.2.043263

[ito2018] Ito, S., Nakatsukasa, Y., 2018. Stable polefinding and rational least-squares fitting via eigenvalues. Numer. Math. 139, 633–682. https://doi.org/10.1007/s00211-018-0948-4

[ito2018] Ito, S., Nakatsukasa, Y., 2018. Stable polefinding and rational least-squares fitting via eigenvalues. Numer. Math. 139, 633–682. https://doi.org/10.1007/s00211-018-0948-4

[ito2018] Ito, S., Nakatsukasa, Y., 2018. Stable polefinding and rational least-squares fitting via eigenvalues. Numer. Math. 139, 633–682. https://doi.org/10.1007/s00211-018-0948-4

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[kogan2021] Kogan, E. and Gumbs, G. (2021) Green's Functions and DOS for Some 2D Lattices. Graphene, 10, 1-12. https://doi.org/10.4236/graphene.2021.101001.

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[horiguchi1972] Horiguchi, T., 1972. Lattice Green's Functions for the Triangular and Honeycomb Lattices. Journal of Mathematical Physics 13, 1411–1419. https://doi.org/10.1063/1.1666155

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[joyce1973] G. S. Joyce, Phil. Trans. of the Royal Society of London A, 273, 583 (1973). https://www.jstor.org/stable/74037

[katsura1971] S. Katsura et al., J. Math. Phys., 12, 895 (1971). https://doi.org/10.1063/1.1665663

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[delves2001] Delves, R. T. and Joyce, G. S., Ann. Phys. 291, 71 (2001). https://doi.org/10.1006/aphy.2001.6148

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[morita1971] Morita, T., Horiguchi, T., 1971. Calculation of the Lattice Green's Function for the bcc, fcc, and Rectangular Lattices. Journal of Mathematical Physics 12, 986–992. https://doi.org/10.1063/1.1665693

[economou2006] Economou, E. N. Green's Functions in Quantum Physics. Springer, 2006.

[georges1996] Georges et al., Rev. Mod. Phys. 68, 13 (1996) https://doi.org/10.1103/RevModPhys.68.13

[eder2017] Eder, Robert. “Introduction to the Hubbard Mode.” In The Physics of Correlated Insulators, Metals and Superconductors, edited by Eva Pavarini, Erik Koch, Richard Scalettar, and Richard Martin. Schriften Des Forschungszentrums Jülich Reihe Modeling and Simulation 7. Jülich: Forschungszentrum Jülich, 2017. https://www.cond-mat.de/events/correl17/manuscripts/eder.pdf.

[odashima2016] Odashima, Mariana M., Beatriz G. Prado, and E. Vernek. Pedagogical Introduction to Equilibrium Green’s Functions: Condensed-Matter Examples with Numerical Implementations. Revista Brasileira de Ensino de Fisica 39, no. 1 (September 22, 2016). https://doi.org/10.1590/1806-9126-rbef-2016-0087.

[ozaki2007] Ozaki, Taisuke. Continued Fraction Representation of the Fermi-Dirac Function for Large-Scale Electronic Structure Calculations. Physical Review B 75, no. 3 (January 23, 2007): 035123. https://doi.org/10.1103/PhysRevB.75.035123.

[hu2010] J. Hu, R.-X. Xu, and Y. Yan, “Communication: Padé spectrum decomposition of Fermi function and Bose function,” J. Chem. Phys., vol. 133, no. 10, p. 101106, Sep. 2010, https://doi.org/10.1063/1.3484491

[1] Hale, S. T. F., and J. K. Freericks. “Many-Body Effects on the Capacitance of Multilayers Made from Strongly Correlated Materials.” Physical Review B 85, no. 20 (May 24, 2012). https://doi.org/10.1103/PhysRevB.85.205444.

[2] Eder, Robert. “Introduction to the Hubbard Mode.” In The Physics of Correlated Insulators, Metals and Superconductors, edited by Eva Pavarini, Erik Koch, Richard Scalettar, and Richard Martin. Schriften Des Forschungszentrums Jülich Reihe Modeling and Simulation 7. Jülich: Forschungszentrum Jülich, 2017. https://www.cond-mat.de/events/correl17/manuscripts/eder.pdf.

[3] Luttinger, J. M. “Analytic Properties of Single-Particle Propagators for Many-Fermion Systems.” Physical Review 121, no. 4 (February 15, 1961): 942–49. https://doi.org/10.1103/PhysRev.121.942.

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

X

Z

