NRGMapping Documentation

Release 1.0.0

Jinguo Leo

December 29, 2015

CONTENTS

CHAPTER

ONE

AUTHORS

- JinGuo Leo (NJU)

Source code: https://github.com/GiggleLiu/nrg_mapping

CHAPTER

TWO

CONTENTS

2.1 Using the code

The requirements are:

- Python 2.6 or higher
- numpy and scipy
- matplotlib for plotting
- gmpy2 the arbituary precision package
- mpi4py (optional)

These packages can be installed using a single command:

```
$pip install -r requirements.txt
```

requirements.txt is contained in the source code.

Before installing these packages, make sure you have lapack or mkl library on your host. However, it is recommended to use Anaconda to install these packages.

Download the code using the Download ZIP button on github, or run the following command from a terminal:

```
$ wget -O nrg_mapping-master.zip https://github.com/GiggleLiu/nrg_mapping/archive/master.zip
```

Within a terminal, execute the following to unpack the code:

```
$ unzip nrg_mapping-master.zip
$ cd nrg_mapping-master/source/
```

Once the relevant software is installed, each program is contained entirely in a single file. The first program, for instance, can be run by issuing:

```
$ python sample_simple.py
```

2.2 Tutorial

2.2.1 Example

First, I will show a rather simple example, it looks like:

```
from numpy import *
from discretization import quick_map,get_wlist,check_disc
from chainmapper import map2chain,check_spec
from matplotlib import pyplot as plt

sx=array([[0,1],[1,0]])
sy=array([[0,-1j],[1j,0]])
sz=array([[1,0],[0,-1]])

#defined the hybridization function
wlist=get_wlist(Nw=5000,mesh_type='log',w0=1e-9,D=1.)
rhofunc=lambda w:identity(2)+0.4*sx+0.5*(0.1+w**2)*sy

#map it to a sun model
discmodel=quick_map(wlist=wlist,rhofunc=rhofunc,N=35,z=linspace(0.05,0.95,10),\
        tick_params={'tick_type':'adaptive','Lambda':2.})[1]

#map it to a Wilson chain
chain=map2chain(discmodel,prec=3000)

#do some checking
#check the sun model
plt.subplot(211)
check_disc(rhofunc=rhofunc,wlist=wlist[20::40],discmodel=discmodel,\
        smearing=0.7,mode='pauli')
plt.ylim(-0.1,1.1)
#check the chain
plt.subplot(212)
check_spec(rhofunc=rhofunc,chain=chain,wlist=wlist[20::40],mode='pauli',smearing=0.7)
plt.ylim(-0.1,1.1)
plt.show()
```

We discretized a hybridization function $\Delta(\omega) = \mathbf{1} + 0.4\omega\sigma_x + 0.5(0.1+\omega^2)\sigma_y$ using the function *discretization.quick_map* for 10 twisting parameters z ranging from 0.05 to 0.95. Then the discrete model is transform to a Wilson chain using *chaimmapper.map2chain*. Finally, we checked the validity of the mapping scheme by recovering the hybridization function for both discrete model and Wilson chain.

In the following subsections, we will show how this example works.

2.2.2 Construct hybridization function

The first step is to construct the hybridization function. A hybridization function should be a function(any callable instance) with only one input variable ω and ouput variable $\Delta(\omega)$ which could be either a number of a square matrix.

The choice of ω space is important, if a linear ω-mesh is choosen naively, the precision near $\omega = 0$ is not guranteened. To generate a logarithmic ω-list:

```
>>> from discretization import get_wlist
>>> wlist = get_wlist(w0=1e-8,Nw=5000,mesh_type='log',D=1,Gap=0)
```

Here, parameter *Gap* is the gapped interval, and *w0* is the onset(the minimum energy scale) of logarithmic mesh.

2.2.3 Using the right discretization mesh points.

The discretization mesh points(tick_params) are important for aquiring optimal scaling behavior. Most of times, you need an 'adaptive' mesh in NRG mapping scheme. The hybridization strength needed by 'adaptive' mesh points is

then defined as the mean square of the eigenvalues of $\Delta(\omega)$.

Other types of discretization meshes like *log*, *linear* et. al. which can be used in some special cases are also predefined.

2.2.4 Mapping procedure

The simplist approach to discretize the continuous hybridization function is to use the *discretization.quick_map* function. Tuple of (<Ticker>s, <DiscModel>) is returned by this function.

<Ticker>s is a list with <Ticker> instances for negative and positive branches. The discrete ticks can be generated using:

```
>>> indices = arange(1,10)     #the decrete indices.
>>> z = 1.0         #twisting parameter
>>> tick_position = ticker(indices+z)    #tick position is always positive.
```

<DiscModel> is nicknamed "sun model" or "star model" with bath replaced by a set of sites directly coupled to the impurity.

The interface of *quick_map* looks like .. autofunction:: discretization.quick_map

In this function, the discretization mesh used is specified by a dict *tick_params*, parameters like *tick_type*, *Lambda* (scaling factor) and some other parameters are passed to the program.

Twisting parameter z could be either a float or an 1D array, $0 < z \leq 1$ is required.

If you're planning to use this model in other programs like Frotran/C++, use <DiscModel>.save('xxx') method to save it to plain texts, see API of <DiscModel> for details of the data format.

2.2.5 Tridiagonalization towards a Wilson chain.

The method *chainmapper.map2chain* is used to cope with this problem, parameters of this function can be easily understood. A <Chain> instance is returned by this function, to use it in other programs, call <Chain>.save('xxx') method to store chain datas.

This procedure is based on the Lanczos(*tridiagonalize.tridiagonalize*) Block Lanczos(*tridiagonalize.tridiagonalize_qr*) algorithm.

For 2 x 2 block size, the block lanczos has an optional implementation *tridiagonalize.tridiagonalize2*, which maximally retains the channel symmetry(general multi-channel version is not implemented due to the difficulty of analitical eigenvalue decomposition).

The interface of function *map2chain* looks like .. autofunction:: chainmapper.map2chain

2.2.6 Check for validity of mapping

Finally, we need a method to check the validity of results, we provided 2 functions to check the quality of the "sun model" and Wilson chain, they are *discretization.check_disc_pauli/discretization.check_disc_eval* and *chainmapper.check_spec*

Due to the difficulty of choosing the smearing factor in Green's function(essentially lorenzian smearing) for logarithmic disctributed energies, the checking procedure itself brings lost of precision. The smearing parameter should be choosen carefully to get a suitible result.

2.3 API

2.3.1 discretization module

Discretize the continuous hybridization function into a discretized model(DiscModel). checking method is also provided.

class discretization.**DiscHandler**
Bases: object

Base class of handlers for discretization of hybridization function.

get_TEfunc(*sgn*, *xmax*, *Nx=500000*, ***kwargs*)
Get representative hopping T(x) and onsite-energy function E(x) for specific branch.

class discretization.**SingleBandDiscHandler**(*wlist*, *rholist*, *tickers*)
Bases: *discretization.DiscHandler*

Handler for discretization of hybridization function.

Construct: SingleBandDiscHandler(wlist,rholist,tickers), with the wlist, rholist containing both positive and negative branches.

Attributes:

nband integer, the number of bands with respect to hybridization function, readonly.

wlists len-2 tuple (wlist_neg,wlist_pos) defining the frequency space.

rholists len-2 tuple (rholist_neg,rholist_pos) of the hybridization function defined on *wlists*.

int_rholists len-2 tuple (rholist_pos_int,rholist_neg_int), integration of rholists over *wlist*.

tickers len-2 tuple with elements function/<Ticker> of discretization points $epsilon(x)$.

wlist,rholist 1D array,ndarray, Whole frequency space and rho(w) defined on it(readonly).

D len-2 list with integer elements, the band range(readonly).

get_TEfunc(*sgn*, *xmax*, *Nx=500000*)
Get hopping function T(x) and on-site energy function E(x).

Parameters:

sgn 0/1 to specify the negative/positive branch.

xmax,Nx The upper limit of x, and the number of samples in x-space for integration and interpolation.

Return: A tuple of functions of representative hopping T(x) and on-site energy E(x) for specific band and branch.

D
The band width.

rholist
The rho(w) defined on whole frequency space.

wlist
The whole frequency space.

class discretization.**MultiBandDiscHandler**(*rhofunc*, *wlist*, *tickers*, *autofix=1e-05*)
Bases: *discretization.DiscHandler*

Handler for discretization of hybridization function.

Construct: DiscHandler(rhofunc,wlist,tickers,autofix=1e-5)

autofix is the the negative tolerence for eigen values of rho(w), above which it will be automatically fixed to 0.

Attributes:

tickers len-2 list with elements <Ticker>/function, The tick generator, a function or a callable instance.

wlist 1D array, The frequency space.

rholist ndarray, The hybridization function defined on *wlist*.

single_band_handlers <SingleBandDiscHandler>, A list of <SingleBandDiscHandler> instances to handler mapping at each eigen-value channel.

nband integer, The number of bands with respect to hybridization function

get_TEfunc(*sgn*, *xmax*, *Nx*)

Get the representative hopping and on-site energy Tfunc/Efunc for multi-band system.

Parameters:

sgn integer, 0/1 to specify the negative/positive branch.

xmax,Nx interger, The upper limit of x, and the number of samples in x-space for integration and interpolation.

Return: A tuple of functions of representative hopping and energy (T(x),E(x)).

nband

The number of bands

discretization.**get_wlist**(*w0*, *Nw*, *mesh_type*, *D=1*, *Gap=0*)

A well defined mesh can make the rho(w) more accurate.

Parameters:

w0 float, The starting w for wlist for *log* and *sclog* type wlist..

Nw integer, The number of samples in each branch.

mesh_type string, The type of wlist.

- *linear* -> linear mesh.
- *log* -> log mesh.
- *sclog* -> log mesh suited for superconductors.

D Interger/len-2 tuple, the band interval.

Gap Interger/len-2 tuple, the gap interval.

Return: 1D array, the frequency space.

discretization.**quick_map**(*rhofunc*, *wlist*, *N*, *z=1.0*, *Nx=500000*, *tick_params=None*, *autofix=1e-05*)

Perform quick mapping(All in one suit!) for nband x nband hybridization matrix.

Parameters:

rhofunc function, The hybridization function.

wlist 1D array, the frequency space holding this hybridization function.

N integer, The number of discrete sites for each z number.

z float/1D array, twisting parameters.

Nx integer, The number of samples for integration over rho(epsilon(x)).

tick_params dict, the parameters for ticks, the following keys could be available,

- ***tick_type* -> The type of ticks** 'log': logarithmic tick, 'sclog': logarithmic ticks suited for superconductor. 'adaptive': adaptive ticks(default). 'linear': linear ticks. 'adaptive_linear': adaptive linear ticks.
- *r* -> Adaptive ratio for *adaptive*/*adaptive_linear* ticks(default: 1.0).
- *wn* -> The typical frequency to achieve the best fit for *ed* ticks(default: pi/200).
- *Lambda* -> The scaling factor for *log*, *adaptive*, *sclog* ticks(default: 2.0).
- *Gap* -> The gap region of the hybridization function, it is used in discretization schemes

that do not allow zeros in hybridization function: *log*, *sclog*, 'linear'(default: 0).

autofix float, Automatically fix the occational small negative eigenvalue(>=autofix) of rho(w) to 0.

Return: Tuple of (<Ticker>,<DiscModel>).

`discretization.`**`check_disc`**(*rhofunc*, *discmodel*, *wlist*, *smearing=0.02*, *mode='eval'*)

check the discretization quality by eigenvalues - the multiple-band Green's function version.

Parameters:

rhofunc function, The original hybridization function.

discmodel <DiscModel>, The discretized model.

wlist 1D array, the frenquncy space.

smearing float, smearing constant.

2.3.2 ticklib module

The library for deciding discretization mesh points.

class `ticklib.`**`Ticker`**(*tp*)

Bases: `object`

Base class for discretization tick generator. Use Ticker(x) to get the tick position of x.

Construct: Ticker(tp)

Attributes:

tp The type string of this tick class.

class `ticklib.`**`AdaptiveTickerBase`**(*tp*, *xf*, *wlist*, *rlist*)

Bases: *`ticklib.Ticker`*

Base class of adaptive tick(Ref: Comp. Phys. Comm. 180.1271).

Construct: AdaptiveTickerBase(tp,xf,wlist,rlist), *wlist* is the base space $\rho(\omega)$, and *rlist* is the weight function of rho(w).

Attributes:

xf A function defined on index-space(to decide logarithmic or linear...).

RD The integration over rho(w) from 0 to D.

iRfunc Inverse function of $\int \rho(\omega)$, $iRfunc(x) = \int_0^{\omega} \rho(\omega)$

class ticklib.**AdaptiveLinearTicker**(*N*, *wlist*, *rholist*, *r=1.0*)
Bases: *ticklib.AdaptiveTickerBase*

Adaptive linear scale tick, with hopping terms constant.

Construct: AdaptiveLinearTicker(Lambda,wlist,rholist,r=1.), *rholist* is the weight of rho defined frequency space *wlist*.

Attributes:

N The number of intervals.

r Adaptive ratio, 1.0 for traditional adaptive and 0 for linear.

class ticklib.**AdaptiveLogTicker**(*Lambda*, *wlist*, *rholist*, *r=1.0*)
Bases: *ticklib.AdaptiveTickerBase*

Zitko's adaptive log scale tick(Ref: Comp. Phys. Comm. 180.1271).

Construct: AdaptiveLogTicker(Lambda,wlist,rholist,r=1.), *rholist* is the weight of rho defined frequency space *wlist*.

Attributes:

Lambda Scaleing factor.

r Adaptive ratio, 1.0 for typical adaptive-log and 0 for traditional log.

class ticklib.**LogTicker**(*Lambda*, *D*, *Gap=0.0*)
Bases: *ticklib.Ticker*

Logarithmic ticks. $\epsilon(x) = \Lambda^{(2-x)}$

Construct: LogTicker(Lambda,D,Gap=0.,N=Inf)

Attributes:

Lambda Scaling factor, a number between 1 and infinity(optimally choose as 1.5-10).

D/Gap The bandwidth/Gap range, then the hybridization function is defined in the interval Gap <= abs(w) <= bandwidth.

class ticklib.**ScLogTicker**(*Lambda*, *D*, *Gap*)
Bases: *ticklib.Ticker*

Logarithmic tick suited for superconductor(logarithmic for normal state).

Construct: ScLogTicker(Lambda,D,Gap)

Attributes:

Lambda Scaling factor.

D/Gap The bandwidth/Gap range.

Note: It is in fact, a mimic for the discretization scheme given by JPSJ 61.3239

class ticklib.**LinearTicker**(*N*, *D*, *Gap=0.0*)
Bases: *ticklib.Ticker*

Linear scale tick.

Construct: LinearTicker(N,D,Gap=0.)

Attributes:

N The number of intervals.

D/Gap The bandwidth/Gap range.

class ticklib.**EDTicker**(*N*, *wlist*, *rholist*, *wn*)
Bases: *ticklib.Ticker*

Scale tick optimized for Exact-diagonalization cost function(PRL 72.1545).

Construct EDTicker(N,wlist,rholist,wn), *rholist* is the weight of rho defined frequency space *wlist*.

Attributes:

N The number of intervals.

D The bandwidth.

wn The typical frequency to achieve the best fit.

Lambda The complex scaling factor.

RD The integration over rho(w) from 0 to D.

iRfunc Inverse function of $\int \rho(\omega)$, $iRfunc(x) = \int_0^\omega \rho(\omega)$

xf A auxiliary function defined on index-space.

Note: The derivation of this tick distribution function involves many approximations, and is tested only for limited cases(like Bethe latttice, constant DOS).

The main theme is to lower down the cost function while keeping the number of intervals fixed, which is different from NRG's infinite intervals.

If it fails to get satisfactory mapping, please contact the author: dg1422033@smail.nju.edu.cn

ticklib.**get_ticker**(*tick_type*, *D*, ***kwargs*)
Get specific <Ticker>.

Parameters:

tick_type The type of discretization ticks,

- *log* -> logarithmic tick, kwargs: Lambda, Gap(optional)
- *sclog* -> logarithmic ticks suited for superconductor, kwargs: Lambda, Gap(optional)
- *adaptive* -> adaptive ticks, kwargs: Lambda, wlist, rholist, r(optional)
- *linear* -> linear ticks, kwargs: N, Gap(optional)
- *adaptive_linear* -> adaptive linear ticks, kwargs: N, wlist, rholist, r(optional)
- *ed* -> ticks suited for fix number of intervals, kwargs: N, wlist, rholist, wn(optional)

D The bandwidth.

kwargs see tick_type.

Return: A <Ticker> instance.

2.3.3 discmodel module

The Discretized model, or the sun model.

class discmodel.**DiscModel**(*Elists*, *Tlists*, *z=1.0*)
Bases: object

Discrete model class.

Construct: DiscModel((Elist_neg,Elist_pos),(Tlist_neg,Tlist_pos),z=1.)

z could be one or an array of float >0 but <=1.

Attributes:

Elist_neg/Elist_pos/Elist(readonly)/Tlist_neg/Tlist_pos/Tlist(readonly) An array of on-site energies and hopping terms.

- The shape of Elist_neg/Tlist_neg is (N_neg,nz,nband,nband)
- The shape of Elist_pos/Tlist_pos is (N_pos,nz,nband,nband)
- The shape of Elist/Tlist is (N_pos+N_neg,nz,nband,nband)

z The twisting parameters.

nz The number of z-numbers(readonly)

nband The number of bands(readonly)

N_neg/N_pos/N The number of intervals for negative/positive/total band(readonly)

is_scalar Is a single band scalar model if True(readonly)

save(*token*)

Save data.

Parameters:

token The target filename token.

Note: For the scalar model mapped from SingleBandDiscHandler, with z=[0.3,0.7] and 2 sites for each-branch. The data file *negfile''posfile*) looks like:

E1(z=0.7)
E1(z=0.7)
E2(z=0.7)
E2(z=0.7)
T1(z=0.7)
T1(z=0.7)
T2(z=0.7)
T2(z=0.7)

However, for the multi-band model, the parameters are allowed to take imaginary parts, Now, the data file for a two band model looks like:

E1[0,0].real, E1[0,0].imag, E1[0,1].real, E1[0,1].imag, E1[1,0].real, E1[1,0].imag, E1[1,1].real, E1[1,1].imag #z=0.3
...

It will take 8 columns to store each matrix element.

Elist

The list of on-site energies.

N

The total number of particles

N_neg

The total number of particles in negative branch

N_pos

The total number of particles in positive branch

Tlist

The list of on-site energies.

is_scalar
is a scalar model(no matrix representation of on-site energies and hopping) or not.

nband
number of bands.

nz
number of twisting parameters.

discmodel.**load_discmodel**(*token*)
Load specific data.

Parameters:

token The target filename token.

Return: A <DiscModel> instance.

2.3.4 chainmapper module

Map a discretized model into a Chain by the method of (block-)lanczos tridiagonalization. checking method is also provided.

chainmapper.**map2chain**(*model*, *prec=5000*)
Map discretized model to a chain model using lanczos method.

Parameters:

model The discretized model(<DiscModel> instance).

prec The precision(by bit instead of digit),

:the higher, the slower, 2000 to 6000 is recommended.

Return: A <Chain> instance.

chainmapper.**check_spec**(*chain*, *rhofunc*, *wlist*, *mode='eval'*, *smearing=1.0*)
Check mapping quality for wilson chain.

Parameters:

chain The chain after mapping.

rhofunc Hybridization function.

wlist The frequency space.

mode Choose the checking method, * *eval* -> check eigenvalues. * *pauli* -> check pauli components, it is only valid for 2 band system.

smearing The smearing factor.

2.3.5 chain module

The Wilson chain class.

class chain.**Chain**(*t0*, *elist*, *tlist*)
Bases: object

Wilson chain class.

Attributes:

t0 The coupling term of the first site and the impurity.

elist/tlist A list of on-site energies and coupling terms.

is_scalar True is it is a single band scalar chain.(readonly)

nsite The number of bath sites.(readonly)

nsite The number of bands.(readonly)

save(*token*)
save a Chain instance to files.

Parameters:

token A string as a prefix to store datas of a chain.

Note: The data is stored in 3 files,

1.<token>.info.dat, len-4 array -> [chain length, number of z, nband, nband], i.g. the shape of elist.

2.**<token>.el.dat, an array of on-site energies, in the format(2-band,2-site,z=[0.3,0.7] as an example)**

E0(z=0.3)[0,0].real E0(z=0.3)[0,0].imag
E0(z=0.3)[0,1].real E0(z=0.3)[0,1].imag
E0(z=0.3)[1,0].real E0(z=0.3)[1,0].imag
E0(z=0.3)[0,1].real E0(z=0.3)[0,1].imag
E0(z=0.7)[0,0].real E0(z=0.7)[0,0].imag
E0(z=0.7)[0,1].real E0(z=0.7)[0,1].imag
E0(z=0.7)[1,0].real E0(z=0.7)[1,0].imag
E0(z=0.7)[0,1].real E0(z=0.7)[0,1].imag
E1(z=0.3)[0,0].real E1(z=0.3)[0,0].imag
... ...

3. <token>.tl.dat, an array of hopping terms with the first element by t0(coupling with the impurity). The data format is similar to above.

to_scalar()
transform 1-band non-scalar model to scalar one by remove redundant dimensions.

is_scalar
True if this is a model mapped from scalar hybridization function.

nband
number of bands.

nsite
The number of sites.

chain.**load_chain**(*token*)
load a Chain instance from files.

Parameters:

token A string as a prefix to store datas of a chain.

Return: A <Chain> instance.

2.3.6 tridiagonalize module

tridiagonalize.**construct_tridmat**(*data*, *offset*)
Construct tridiagonal matrix.

Parameters:

data The datas of lower, middle, upper tridiagonal part.

offset The offsets indicating the position of datas.

Return: 2D sparse matrix, use res.toarray() to get a dense array.

`tridiagonalize.`**`tridiagonalize`**(*A*, *q*, *m=None*, *prec=5000*, *getbasis=False*)

Use m steps of the lanczos algorithm starting with q to generate the tridiagonal form of this matrix(The traditional scalar version).

Parameters:

A A sparse hermitian matrix.

q The starting vector.

m The steps to run.

prec The precision in bit, *None* for double precision.

getbasis Return basis vectors if True.

Return: Tridiagonal part elements (data,offset), | data -> (lower part, middle part, upper part) | offset -> (-1, 0, 1) to indicate the value of (j-i) of specific data with i,j the matrix element indices.

To construct the matrix, set the block-matrix elements with block indices j-i == offset[k] to data[k]. This is exactly what *construct_tridmat* function do.

Note: The initial vector q will be renormalized to guarant the correctness of result,

`tridiagonalize.`**`tridiagonalize2`**(*A*, *q*, *m=None*, *prec=5000*, *getbasis=False*)

Use block lanczos algorithm to generate the tridiagonal part of matrix. This is the symmetric version of block-tridiagonalization in contrast to *qr* version. However, only matrices with blocksize p = 2 are currently supported.

Parameters:

A A sparse Hermitian matrix.

q The starting columnwise orthogonal vector q with shape (n*p,p) with p the block size and n the number of blocks.

m the steps to run.

prec The precision in bit, *None* for double precision.

getbasis Return basis vectors if True.

Return: Tridiagonal part elements (data,offset), | data -> (lower part, middle part, upper part) | offset -> (-1, 0, 1) to indicate the value of (j-i) of specific data with i,j the matrix element indices.

To construct the matrix, set the block-matrix elements with block indices j-i == offset[k] to data[k]. This is exactly what *construct_tridmat* function do.

Note: The orthogonality of initial vector q will be re-inforced to guarant the convergent result, meanwhile, the orthogonality of starting vector is also checked.

`tridiagonalize.`**`tridiagonalize_qr`**(*A*, *q*, *m=None*, *prec=5000*)

Use m steps of the lanczos algorithm starting with q - the block QR decomposition version.

Parameters:

A A sparse Hermitian matrix.

q The starting columnwise orthogonal vector q with shape (n*p,p) with p the block size and n the number of blocks.

m The number of iteractions.

prec The precision in bit, *None* for double precision.

Return: Tridiagonal part elements (data,offset), | data -> (lower part, middle part, upper part) | offset -> (-1, 0, 1) to indicate the value of (j-i) of specific data with i,j the matrix element indices.

To construct the matrix, set the block-matrix elements with block indices j-i == offset[k] to data[k]. This is exactly what *construct_tridmat* function do.

Note: The orthogonality of initial vector q will be re-inforced to guarant the convergent result, meanwhile, the orthogonality of starting vector is also checked.

2.3.7 utils module

Physics utility library

`utils.`**`eigh_pauliv_npy`**(*a0*, *a1*, *a2*, *a3*)
eigen values for pauli vectors - numpy version.

Parameters:

a0/a1/a2/a3 Pauli components.

Return: Tuple of (eval,evecs), the eigenvalue decomposition of A.

`utils.`**`plot_pauli_components`**(*x*, *y*, *method='plot'*, *ax=None*, *label='\\sigma'*, ***kwargs*)
Plot data by pauli components.

Parameters

x,y Datas.

ax The axis to plot, will use gca() to get one if None.

label The legend of plots.

method *plot* or *scatter*

kwargs The key word arguments for plot/scatter.

Return: A list of plot instances.

`utils.`**`mpqr`**(*A*)
Analytically, get the QR decomposition of a matrix.

Parameters:

A The matrix.

Return: (Q,R), where QR=A, Q is orthogonal by columns, and R is upper triangular.

`utils.`**`H2G`**(*h*, *w*, *tp='r'*, *geta=0.01*, *sigma=None*)
Get Green's function g from Hamiltonian h.

Parameters:

h An array of hamiltonian.

w The energy(frequency).

tp The type of Green's function.

- 'r': retarded Green's function.(default)
- 'a': advanced Green's function.

- 'matsu': finite temperature Green's function.

geta Smearing factor. default is 1e-2.

sigma Additional self energy.

Return: A(Array of) Green's function.

utils.**s2vec**(*s*)

Transform a 2 x 2 matrix to a 4 dimensional vector, corresponding to s0,sx,sy,sz component.

Parameters:

s The matrix.

Return: len-4 vector indicating the pauli components.

utils.**vec2s**(*n*)

Transform a vector of length 3 or 4 to a pauli matrix.

Parameters:

n 1D array of length 3 or 4 to specify the *direction* of spin.

Return: 2 x 2 matrix.

utils.**sqrth2**(*A*)

analytically, get square root of a hermion matrix.

Parameters:

A The matrix.

Return: 2D array, The matrix squareroot of A.

c

d

t

u

U

V

W