Python Codes Documentation

Release 0.1

Daniel Cocks

May 07, 2012

CONTENTS

#.. only:: html

Contents:

CHAPTER

ONE

RDMFT FOR GOLDMAN ETAL TOPOLOGICAL INSULATOR PROPOSAL CODES

This module implements real-space DMFT code for the spinful Hofstadter model, as proposed in Goldman et al, PRL 105, 255302 (2010). The non-interacting Hamiltonian is:

$$H = -\sum_j \left\{ t_x c^\dagger_{j+\hat{\mathbf{x}}} e^{-i2\pi\gamma\sigma^x} c_j + t_y c^\dagger_{j+\hat{\mathbf{y}}} e^{i2\pi\alpha x\sigma^z} c_j + \text{h.c.} \right\} + \lambda_x \sum_j (-1)^x c^\dagger_j c_j$$

where the creation/annihiliation operators are $c_j = (c_{j,\uparrow}, c_{j,\downarrow})^T$ and the Peierl's phases result from the Landau gauge $\boldsymbol{A} = (0, Bx, 0)$.

The main calculation is controlled by `TI_CTAUX()` and this should be the entry point for any code. The parameters for the self-consistency will be set up here, and the appropriate symmetries and potential will be created by the `TI_Symmetry()` function. `TI_CTAUX()` then calls `ConvergenceSpin()` which glues the individual self-consistency loops together. It manages the dampening and convergence to a target filling. The individual self-consistency loops themselves are implemented by `SelfConsistencySpin()`, which generates the Green's functions and calls the impurity solver. Further detail about the process is given below.

A lot of the information that is output from the solver is saved into files with three extension types:

- `*.dat`: These files are text files, and are meant for human readability.
- `*.npy`: These files are numpy saved files which contain one array.
- `*.npz`: These files are numpy groups which contain many arrays.

In order to look at the data in the `*.np?` files, please see the section *Reading .npy and .npz files*.

There are many analysis functions, however an important convenience function exists to access general information about an individual run: `ReadInfo()`. It will parse the `info.dat` file, which summarizes a run and return a dictionary of useful information.

This module depends on `ImagConv`, `CT_AUX_SM`, `ED_new`, `BlockMatrixInv` and `Continuation`.

Summary of functions:

RDMFT Calculations: (with more high-level functions first)

`TI_CTAUX()` Sets up parameters for the calculation.

`ConvergenceSpin()` Manages the self-consistency loops.

SelfConsistencySpin() Performs one single self-consistency loop with parallelization.

GreensMatSinglePython() Calculates the Green's matrix for a single omega.

TI_Symmetry() Generates appropriate symmetry and potential arrays.

Continuation and Fourier transforming:

TI_FourierCTAUX() Interface for `FourierTransformSpin()` for Matsubara frequencies.

TI_FourierCTAUXRealFreq() Interface for `FourierTransformSpin()` for real frequencies.

FourierTransformSpin() Calculates the various types of Fourier transformed Green's matrices.

FourierTransformSpinSingle() Calculates the Fourier transformed Green's matrix for a single omega.

TI_ContinueCTAUX_SE() Continues the Matsubara self-energies to real-space.

TI_ContinueCTAUX_Gk() Continues the Fourier transformed Green's matrix to real-space.

ContinueSelfEnergy() Handles the continuation of a single function to real-space.

Analysis:

ReadInfo() Parse the `info.dat` file into a convenient dictionary.

ReadGkData() A convenience function to read the data output from `TI_FourierCTAUX()` and `TI_FourierCTAUXRealFreq()`.

AverageData() A convenience function to average the last few self-consistency iterations.

PlotGkData() Create some basic plots from the G(x,y) Green's function data.

PlotPeaksMap() Create some basic plots from the G(x,k_y) Green's function data.

PlotOmega0Data() Show detailed plots of the spectra about omega=0.

SavePeaksTI() Collects all the peaks from each spectra.

FindPeaksWithWeights() Identifies peaks within one spectra.

PlotPeaksWithWeights() Plots the peaks calculated with `SavePeaksTI()`

FitMagExp() Extrapolates the magnetisation to infinite self-consistency loops.

CalcNZInvariant() Calculate the integer QH invariant for S_z conserving systems.

GenerateAllDataCTAUX() Run various other functions to generate the common data from each run.

GenerateAllDataForU0() Setup a fake U=0 run and generate data like `GenerateAllDataCTAUX()`.

DisplayFillingFromNup() Display the filling and spin dependence over the grid.

JudgeValidity() Give a rough estimate of whether the run has converged and if it has encountered any issues.

Additional:

These functions are used only for specific purposes and have not been documented.

TI_AnotherIter() Performs an additional self-consistency loop at larger CT sweeps.

- FitMagExp()
- LookForGaps()
- PlotLastMags()
- PlotGkLargestAtZero()
- DoNiceMagPlot()

1.1 Typical Run of the Program

A typical calculation will perform something similar to the code beneath. The steps are basically

1. Setup the parameters.
2. Determine if a previous run has occurred, and if so from which iteration to resume.
3. Run the calculation with TI_CTAUX().
4. Run the continuation with TI_ContinueCTAUX_SE().
5. Create all data with GenerateAllDataCTAUX().
6. Try to catch obvious errors with JudgeValidity().

This code currently is designed to run on the Loewe cluster which has nodes with 24 cores. If you would like to run this on a local machine, then one should indicate how many cores are available with `export NSLOTS=x` in a shell prompt, followed by `bash script.sh`. The code is:

```
#! /bin/bash
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=24
#SBATCH --job-name=JOBNAME
#SBATCH --mem-per-cpu=2000
#SBATCH --time=4-00:00:00
#SBATCH --partition=parallel
#SBATCH --mail-type=FAIL
#SBATCH --tmp=2000

. ~/.bashrc

export PYTHONPATH="$PYTHONPATH:$HOME/TISM_Michael/python/"
export CTAUX_CODE_PATH="$HOME/TISM_Michael/ccode/Michael_TISM_CT/Spin-mixin/"

if [ -n "$SLURM_JOB_ID" ] ; then
    TMPDIR="/local/$SLURM_JOB_ID"
fi

if [ -z "$NSLOTS" ] ; then
export NSLOTS=24
fi

python - <<EOF
```

```
from __future__ import print_function
import sys,os
from std_imports import *
import matplotlib
matplotlib.use('Agg')

max_iters = 30
U = 0.01
mu = -3+U/2.
beta = 20.
N = 24
M = 60
boundaries = "PP"
p = 1
q = 6
lambda_x = 0.
gamma = 0.
t = 1.
num_sweeps = 10**7
row_sym = "grid6"
force_filling = 1.0/6.0
solver = "CT-AUX"
covar_factor = 0.01

import os
cwd = os.getcwd()

import time
start = time.time()

import RDMFT_TISM

import glob
filelist = glob.glob('SE_iter*.npy')
last_iter = -1
for filename in filelist:
        num = int(filename.split('_iter')[1].split('.')[0])
        if num > last_iter:
                last_iter = num

if last_iter >= max_iters-1:
        pass
else:
        if last_iter == -1:
                last_iter = False
                valid = False

        RDMFT_TISM.TI_CTAUX(U,mu,beta,(N,M),boundaries,(p,q),max_iters,num_sweeps,row_sym,l

RDMFT_TISM.TI_ContinueCTAUX_SE(covar_factor=covar_factor,model=4)
RDMFT_TISM.GenerateAllDataCTAUX()
RDMFT_TISM.JudgeValidity()

if True:
    # Clean up files we don't want to keep for space saving reasons.

    def unlinkdan(x):
        try:
```

```
            os.unlink(x)
        except:
            pass

    for i in range(max_iters-6):
        if i % 30 != 0:
            unlinkdan('SE_iter{0}.npy'.format(i))
            unlinkdan('everything_iter{0}.npz'.format(i))
    unlinkdan('data.pickle.gz')
    unlinkdan('SE.pickle.gz')

    # Also remove the huge *.npz files
    if True:
        import glob
        filelist = glob.glob('G[kx]*.npz')
        for filename in filelist:
            unlinkdan(filename)

os.chdir(cwd)
with open('runtime.txt','wb') as file:
    file.write(str(time.time() - start))

EOF

exit 0
```

1.2 Reading .npy and .npz files

To open `.npy` and `.npz` files, one must use the python function `load` from the module `numpy`. `.npy` files contain a single array of data, whereas `.npz` files contain many sets of data. The following code snippet illustrates how to load these:

```
from numpy import load
SE = load('SE_iter20.npy')
SE[0,:,0] # <-- This is \Sigma_{i=0,upup}(omega)

data = load('everything_iter20.npz')
data['SE'][0,:,0] # As above
data['N_dbl'][0] # Double occupancy at site i=0
```

With `.npz` files one can view all of the data contained within, with the following method:

```
data = load('everything_iter20.npz')
print(data.files)
['GT', 'old_SE', 'G', 'weiss', 'WT', 'SE', 'test_mat']
```

It is also possible to convert these to Matlab files using the following type of code:

```
from numpy import load
from scipy.io import savemat
data = load('file.npz')
savemat('newfile.mat',data)
```

Be aware that the `None` type cannot be interpreted into Matlab, so one may have to manually delete these before saving.

1.3 Member functions

RDMFT_TISM.**TI_CTAUX**(*U*, *mu*, *beta*, *(N, M)*, *boundaries*, *(p, q)*, *max_iters=100*, *num_sweeps=1000000.0*, *row_sym='AF'*, *num_omega=400*, *SE=None*, *lambda_x=0.0*, *gamma=0.0*, *force_filling=None*, *fit_large_SE=False*, *continue_from_iter=False*, *solver=None*)

Here is where the main part of the calculation begins. This function sets up all the parameters to be passed to the worker function `ConvergenceSpin()`.

Arguments `U,mu,beta,(N,M),boundaries,(p,q),max_iters,num_sweeps,num_omega,SE,gamma,fo` are passed directly to `ConvergenceSpin()` and are documented there.

Arguments `row_sym,lambda_x` are passed directly to `TI_Symmetry()` and are documented there. The resultant arrays for `symmetry` and `potential`, which `ConvergenceSpin()` requires, are determined there.

The function will generate a `info.dat` file, with a header that describes the parameters specified. See `ReadInfo()` for a useful way to parse this file.

If `continue_from_iter` is an integer, an existing calculation will be resumed from that iteration. The self-energy for the iteration specified is loaded from `SE_iter<iter>.npy` and if `force_filling` is active then the latest value of `mu` is found with the help of `ReadInfo()`.

If `SE==None` then the initial self-energy will be set to zero. However, if a `SE_init.npy` file exists in the directory, then this will be read and used as the initial self-energy. This can be useful for using the result of a run with lower `CT_sweeps`, or starting from a magnetic solution. When this occurs, randomisation is disable in `ConvergenceSpin()`.

Note: although it is possible to use this function to calculate non-interacting properties, it is recommended to use `GenerateAllDataForU0()` to directly generate the output.

RDMFT_TISM.**ConvergenceSpin**(*U*, *mu*, *beta*, *gamma*, *potential*, *num_omega*, *(N, M)*, *boundaries*, *t*, *(p, q)*, *SE=None*, *max_iters=100*, *symmetry=None*, *use_tmpdir=False*, *force_para=False*, *CT_sweeps=None*, *row_sym=None*, *fit_large_SE=False*, *force_filling=False*, *continue_from_iter=False*, *solver=None*, *randomize=True*)

This function is the glue between the self-consistency loops. It calculates any quantities that are required between loops and performs dampening, etc...

The parameters `U,mu,gamma,potential,(N,M),boundaries,t,(p,q),SE,symmetry,CT_sweeps,fit_` are the same as for `SelfConsistencySpin()` and are mostly passed on directly.

The array of omega frequencies is calculated from `num_omega` and `beta` using the `ImagConv` module. Note that `num_omega` specifies the number of *positive* frequencies only, which will be doubled in the output to include the negative Matsubara frequencies.

The file `info.dat`, which should already exist from a higher-level function, will be appended to, in order to record the progress. At the current time, the following information is saved in a tabulated format:

- `iter`: The iteration number.
- **`SE_err`: The error in the self-energy, calculated by integrating** over `omega` the difference of `SE` from one iteration to the next, for each site and then averaging over all sites. Each component $\Sigma_{\sigma\sigma'}$ is calculated separately and averaged.
- **`G_err`: The estimated error in the Green's functions, calculated** as for `SE_err`. Only the imaginary part is shown.
- **`N_up_err`: The largest difference of N_up on all sites, from the** last iteration.

- `N_up_err2`: As for `N_up_err` but from two iterations previous.
- `N_tot`: The average total filling.
- `Sx`: The average (absolute) spin projected along x-axis.
- `Sy`: The average (absolute) spin projected along y-axis.
- `Sz`: The average (absolute) spin projected along z-axis.
- **`cur_mu`: Current value of chemical potential. Only relevant for** finite `force_filling`.

An example `info.dat` file follows:

```
Version 10
Solver CT-AUX
N,M 23 60
Boundaries OP
U 2.0
mu 1.0
num_sweeps 10000000.0
num_omega 400
row_sym AFwholerow
p,q 1 6
beta 20.0
t 1.0
lambda_x 0.75
Force_filling None
gamma 0.25
fit_large_SE False
-----
iter | SE_err | G_err | N_up_err | N_up_err2 | N_tot | Sx | Sy | Sz | cur_mu
 0  -1.00000  -1.00000  -1.00000  -1.00000   0.99342   0.00000   0.00000   0.00054
 1   0.19884   0.00958   0.00161  -1.00000   0.99305   0.00071   0.00014   0.00054
 2   3.44063   0.00526   0.00307   0.00293   0.99269   0.00071   0.00008   0.00066
 3   0.14029   0.04087   0.00585   0.00733   0.99367   0.00014   0.00001   0.00092
<... snip ...>
25   0.12286   0.00163   0.00153   0.00224   0.99399   0.00000   0.00000   0.00122
26   0.12649   0.00188   0.00239   0.00223   0.99406   0.00000   0.00000   0.00106
27   0.12293   0.00189   0.00290   0.00317   0.99413   0.00000   0.00000   0.00117
28   0.11791   0.00160   0.00216   0.00192   0.99404   0.00000   0.00000   0.00140
29   0.11835   0.00175   0.00204   0.00180   0.99413   0.00000   0.00000   0.00112
```

In addition, several other files are created. The files `Nup_Ndown_<iter>.dat` list `Nup,Ndown,Sx,Sy,Sz` in text format for each site of the lattice, as follows:

```
Nup Ndown Sx Sy Sz
  0.49523   0.40201   0.00031   0.01286   0.04661
  0.59268   0.51195  -0.00020   0.00002   0.04036
  0.55328   0.35085   0.00206   0.03385   0.10121
<...snip...>
  0.60168   0.50198  -0.00019  -0.00057   0.04985
  0.49988   0.39751   0.00028  -0.01449   0.05118
```

The files `SE_iter<iter>.npy` store the self-energy for all sites which are evaluated in the impurity solver in the numpy `.npy` format. The files `everything_iter<iter>.npz` store much more information in a numpy `.npz` format, including self-energy, Green's functions, weiss functions, imaginary time functions, and double occupancy. For information on how to read the numpy file formats, see *Reading .npy and .npz files*.

The function loops over the self-consistency for up to `max_iters`, unless convergence is reached in this time. At the moment there is no method with which to adjust the convergence, and it is hard-coded to be that the error in `N_up_err` and `Im_G_err` is `< 1e-4`. If force_filling is also enabled, then the filling difference to the desired filling must be `< 1e-4`.

> Note that these tight criteria will only ever be reached for the ED solver. In the CT-AUX case, one should not (and really cannot) rely on such criteria, due to the jitter of the solver. One should instead look at how the results behave over many iterations.

The argument `continue_from_iter` has no numerical effect here, and only affects the formatting of the output files. See `TI_CTAUX()` for more information.

To allow for different magnetic order, the self-energies of the first two iterations are randomized if the parameter `randomize==True`. That is, all of the self-energy functions are averaged for each omega

$$A(i\omega_n) = \sum_{\sigma,\sigma',i} \Sigma_{\sigma,\sigma',i}(i\omega_n)$$

and then this averaged quantity is spread over the self-energy with a random coefficient. That is:

$$\Sigma'_{\sigma,\sigma',i}(i\omega_n) = C_{\sigma,\sigma',i,n} A(i\omega_n)$$

where C is a random variable that is evenly distributed over 0-1.

If the parameter `force_filling` is False, then there the chemical potential is never adjusted. Otherwise `force_filling` must be a filling value, which is the target filling, as measured by `N_tot`, described above. The procedure for adjusting the filling is:

1. Run for a few (five) iterations without any adjustments.
2. Wait for a global filling error of < 0.01 to activate the search of filling.
3. When the search is activated, if `N_up_err` is > 0.05 then do nothing special this iteration (this is to prevent false bounding below). Otherwise:
   (a) If the filling is above the target, set `max_mu` as a bound.
   (b) If the filling is below the target, set `min_mu` as a bound.
   (c) **If the target filling is bounded by `max_mu` and `min_mu` then bisect this region** to choose the new `mu`.
   (d) **If the target filling is not bounded, adjust `mu` in the right direction by a change** `dmu = max(0.5,U/10)`.

Notes:

The code contains some very weak dampening using only the previous iteration,

$$\Sigma'_n = \alpha\Sigma_n + (1-\alpha)\Sigma_{n-1}$$

which can be adjusted in the function itself. Currently `alpha=0.9`.

It is possible to force paramagnetic behaviour by setting the argument `force_para=True`. However this does not take into account the cross terms and so only works correctly without spin-mixing.

An option `use_tmpdir` is provided, which allows one to run this code is a randomly generated temporary directory. This is not recommended, as all useful output is actually saved to files in the directory.

Returns: `SE,N_up,N_down,N_dbl,G`

`SE` (complex double array) The self-energy of the sites where the impurity problem is calculated.

`N_up` (double array) The filling of the up spin at each site.

`N_down` (double array) As above for down spin.

`N_dbl` (double array) Double occupancy at each site.

`G` (complex double array) Green's function in Matsubara frequency at calculated sites.

Note that the return values of this function are not necessary, as all information is also saved to various files.

RDMFT_TISM.**SelfConsistencySpin**(*U*, *mu*, *beta*, *gamma*, *SE*, *potential*, *omega*, *(N, M)*, *boundaries*, *t*, *(p, q)*, *symmetry=None*, *CT_sweeps=None*, *parallelize_CT=True*, *fit_large_SE=False*, *solver='CT-AUX'*, *ED_params=None*)

Perform one self-consistency loop of the RDMFT.

This function takes a set of self-energies `SE` and performs the self-consistency steps to generate a new set of self-energies. These steps are:

1. Determine which sites need to be calculated in the impurity solver.
2. Calculate Green's functions for the old self-energy (using `GreensMatSinglePython()`).
3. Determine the local Weiss Green's function to give to the impurity solver.
4. Run the impurity solver.
5. Take the resultant Green's functions and calculate a new self-energy.
6. Return this new self-energy, along with other various local quantities.

The parameters for the system are given with `U,mu,beta,gamma,N,M,boundaries,t,p,q`. See `GreensMatSinglePython()` for more details about these.

The array `omega` specifies the Matsubara frequencies to use in the calculations. Note that this function currently assumes that this grid is mirrored about zero such that `omega[i] == -omega[len(omega)-1-i]`. `omega` should be purely imaginary.

The self-energy is given as either `SE==None`, for which the self-energy is assumed to be zero, or of dimensions `(3,num_calc_sites,len(omega))`, where the first dimension corresponds to the individual self-energy funtcions, $\Sigma_{\uparrow\uparrow}, \Sigma_{\downarrow\downarrow}, \Sigma_{\uparrow\downarrow}$, and `num_calc_sites` is described below.

Similarly, `potential` is an 1D array of size `num_calc_sites` that gives the local potential at each site. If `potential==None` then it is assumed to be zero at each site.

Symmetries:

If `symmetry==None` then all sites will be calculated in the impurity solver step and the `SE` array will be of dimensions `(3,NxM,len(omega))`. To specify a symmetry, such that only `num_calc_sites` need be calculated, one can pass a list via `symmetry`. This list must contain `num_calc_sites` `None`'s, which designate which sites must be calculated in the impurity solver. Call this list `calc_sites`. The remaining elements of the list must be integers, which indicate that the site is identical to one within `calc_sites`. I.e. the integer is an index to `calc_sites`. An example better describes this. If we have:

```
symmetry = [None,None,0,1,None,None,2,3]
```

then only four sites of the eight will be solved in the impurity step (those with `None`'s). The Green's functions will then be copied into the other sites, such that:

```
[0,1,0,1,2,3,2,3]
```

indicates the symmetry, where entries with the same integer have the same Green's function.

Note that the `SE` array is expanded from the input to the complete grid by copying these entries as described above, and so the `symmetry` array must be in row-major ordering as expected by `GreensMatSinglePython()`. The above example, for `(N,M)=(2,2)` would then be a calculation on the left half of a grid.

In addition, one can specify a negative integer for the `symmetry` elements. This then indicates an anti-ferromagnetic symmetry, which is given by

$$G_{A,\sigma\sigma}(i\omega_n) = G_{B,\bar{\sigma}\bar{\sigma}}(i\omega_n)$$
$$G_{A,\sigma\bar{\sigma}}(i\omega_n) = -G_{B,\sigma\bar{\sigma}}(i\omega_n)$$

The negative integer is also an index directly to the array `calc_list` in python. I.e. an index `-1` will access the last element of the calculation list and an index `-num_calc_sites` is the same as `0`, but with AF symmetry instead. For example, one can specify:

```
symmetry = [None,None,0,1,0,1,0,1,-2,-1,-2,-1,-2,-1,-2,-1]
```

which indicates AF order on a `(N,M)=(8,2)` grid, with a two-site symmetry that is repeated in the lower row, but flipped for AF order in the upper row.

The solver is chosen from the `solver` argument. For `solver=='CT-AUX'` there are some additional options: `CT_sweeps` sets the number of iterations in the solver for a single impurity problem, and `parallelize_CT` sets whether the solver itself is parallelized (i.e. sweeps are split over different cores) or the solver is parallelized over many sites. Note that `mu` should be provided without the CT correction. I.e. $\mu = U/2$ corresponds to half-filling for CT and ED. For more information about the CT solver, see `CT_AUX_SM`.

For `solver=='EDx'` the ED solver is chosen with x orbitals. To speed the solver's minimization of Anderson parameters, one can also provide a list in `ED_params` where each element of the list corresponds to the impurity calculation on one site and has an array of Anderson parameters. See `ED_new` for further information.

This function is parallelized with the `multiprocessing` package. The Green's function calculations will be split onto different cores for different omega values. For the ED solver, the calculations will be parallelized over each site. For the CT-AUX solver, either the solver itself is parallelized (as given in the `parallelize_CT` input argument) without parallelizing over the sites, or the code is parallelized over the sites without parallelizing the solver itself. If you would like to disable multiprocessing entirely, you can do so by:

```
import dancommon
dancommon.disable_mp = True
```

Additional notes:

- The diagonal spin self-energies are enforced to be real in imaginary time.
- The cross-spin terms fit the relation $f_{\sigma\bar{\sigma}}(i\omega) = f^*_{\bar{\sigma}\sigma}(-i\omega_n)$ where f stands for either the Green's functions or the self-energy.
- There exists a `fit_large_SE` option that is only valid for the case without spin-mixing. It turned out that this only hides problems, rather than help with the self-consistency loops.

RDMFT_TISM.**GreensMatSinglePython**(*SE*, *potential*, *(N, M)*, *boundaries*, *omega*, *(p, q)*, *mu*, *gamma*, *no_invert*, *complete_mat*, *t=1.0*)

This function calculates the inverted Green's matrix at frequency `omega` (real or Matsubara) from a given self-energy `SE` and system parameters:

- Flux alpha = `p/q` `gamma`
- Global chemical potential `mu`
- Spin-mixing `gamma`
- Lattice dimensions `N``x``M`
- Hopping `t`.

A site-dependent potential can be specified as an array in `potential` and the boundary conditions are specified as a string of two characters ('P' or 'O' for periodic and open respectively) in `boundaries`. (E.g. `boundaries='OP'` for open boundary conditions in x and periodic in y)

The matrix is defined from its inverse as:

$$\begin{aligned}(G^{-1})_{ij} &= t\exp(\pm 2\pi i x p/q)\delta_{i_x,i_x\pm 1}\\ &+ t\exp(\mp 2\pi i\gamma)\delta_{i_y,i_y\pm 1}\\ &+ (\omega+\mu-V_i)\delta_{ij} - \Sigma_{i,j_\sigma}\delta_{i_x,j_x}\delta_{i_y,j_y}\end{aligned}$$

where $i \equiv i_x, i_j, i_\sigma$. Note the reversal of sign in the gamma term.

Normally, the matrix returned is the matrix of Green's functions, but when `no_invert == True` then the uninverted matrix is returned (i.e. the single-particle Hamiltonian with omega subtracted).

As well, the function normally returns the diagonal $G_{ii}\delta_{i,j}$ terms, however one can return the complete matrix by setting `complete_mat == True`.

Note that the self-energy must be provided in the variable `SE` as a tuple of four vectors, i.e. `SE = (SE_up,SE_down,SE_updown,SE_downup)`. This is necessary for the Matsubara frequency case, where this function cannot calculate `SE_downup(i omega) = SE_updown(-i omega)`. Each vector must contain `NxM` elements which are stored in row-major ordering (i.e. the first `N` elements are one row of the system).

This function uses the `BlockMatrixInv` module for quick inversion.

Returns:

`mat` (complex double array)

- With `complete_mat==False`, this is only the diagonal. The array dimensions are: x-coord,y-coord,func, where func is 0,1,2,3 for G_upup, G_downdown, G_updown, G_downup. I.e. mat[1,2,0] is G_upup(x=1,y=2)
- With `complete_mat==True`, the array dimensions are: i_x,i_y,i_spin,j_x,j_y,j_spin such that mat[1,2,0,5,6,1] = G_updown(x=1,x'=5,y=2,y'=6)

RDMFT_TISM.**TI_Symmetry**(*(N, M)*, *row_sym*, *lambda_x*)

This function defines the lattice symmetries. From the arguments `row_sym`, the arrays `symmetry` and `potential` which are used in `SelfConsistencySpin()` are generated for a grid of size `NxM` and staggering `lambda_x`.

Details of the structure of `symmetry` are specified in `SelfConsistencySpin()`, but will be briefly repeated here.

`symmetry` is an array, whose elements are in row-major ordering, indicating sites of the lattice. If an element is `None` then it will be fully calculated in the impurity solver. These sites make up a list `calc_sites`. If an element is a positive integer then this indicates

that the site has an identical Green's function with `calc_sites[i]`. If the element is a negative integer then the site has antiferromagnetic symmetry with `calc_sites[i]`.

There are several different types of symmetry possible:

`row_sym=='AF'` Calculate a half row of the lattice, and mirror this to the rest of the row, and then enforce AF order in the y-direction. Only valid for open boundaries in x with an odd number of sites in the x-direction and gamma = 0 or 0.25.

`row_sym=='AFwholerow'` Calculate an entire row of the lattice, and enforce AF order in the y-direction.

`row_sym=='AFwholerow_quartic<val>'` As above, but with an additional trapping in the x direction. <val> is a number that defines the height of the trap at the system edges.

`row_sym=='gridAF'` Assume only the simplest AF order in periodic case. If `lambda_x=0` then only calculate one site and enforce AF order in x and y directions. If `lambda_x!=0` then calculate two sites in the x-direction and repeat these in x.

`row_sym=='gridAF<val>'` Calculate a line of sites <val> long in x, and repeat these along x. Enforce AF order in y. Assumes periodic boundaries.

`row_sym=='grid<val>'` Calculate a square of sites <val>x<val> (i.e. <val>^2 sites in total). Repeat these along x and y. Assumes periodic boundaries.

`row_sym=='rect<x>x<y>'` Calculate a recentable of sites <x>x<y> (i.e. <x>*<y> sites in total). Repeat these along x and y. Assumes periodic boundaries.

`row_sym==<val>` Calculate <val> number of rows and repeat these in y.

`row_sym==None` Calculate all sites.

Note that this is the only point in the code at which the parameter `lambda_x` is explicitly considered. Afterwards, `potential` is all that needed.

Returns: `symmetry,potential`

`symmetry` (list of `None` or integers) See description above.

`potential` (double array) The local potential at sites that will be calculated in the impurity solver.

RDMFT_TISM.**TI_FourierCTAUX**(*use_U0=False*, *y_only=True*, *no_transform=False*, *complete_mat=False*, *no_invert=False*, *use_additional=None*, *only_omega0=False*, *num_average=5*)

This function provides `FourierTransformSpin()` with the appropriate parameters to calculate various Fourier transformed Green's function matrices of Matsubara frequencies.

The self-energy is given using `AverageData()` or using `additional.npz` if `use_additional==True`.

The parameters `y_only,no_transform,complete_mat,no_invert,only_omega0` are described in `FourierTransformSpin()`. However, their values determine the filename of which to save. The filename is:

```
<base><comp><onlyomega0><noinvert>.npz
```

where `<base>` is 'Gx' when `no_transform == True`, 'Gk' if `y_only == True` or 'Gkk' otherwise. `<comp>,<onlyomega0>,<noinvert>` are empty if `complete_mat,only_omega0,no_invert` are False and appropriately named otherwise. For example, a filename:

```
Gk_comp_o0.npz
```

indicates the quantity G(x,x',ky,ky') at omega~0 has been saved.

The option `use_U0` exists to allow a zero self-energy to be chosen. However, it is still required that a `info.dat` file already exists. See `GenerateAllDataForU0()`.

RDMFT_TISM.**TI_FourierCTAUXRealFreq**(*redo_peaks=False*, *redo_all=False*, *min_weight=None*, *max_weight=None*, *use_U0=False*, *y_only=True*, *no_transform=False*, *show_figures=True*, *only_omega0=False*, *complete_mat=False*, *r_deriv='N'*, *no_invert=False*, *broadening=None*, *omega=None*, *interpolate_omega=True*)

This function provides `FourierTransformSpin()` with the appropriate parameters to calculate various Fourier transformed Green's function matrices of real frequencies.

The self-energy is taken from the continued self-energy in `SE_real.npz` and interpolated onto a grid of frequencies, chosen to be finer around omega=0, so as to generate smoother figures. If `interpolate_omega==False` then the frequencies are taken from that provided in `omega`. If `omega==None` then the grid from the analytical continuation is used instead.

Some broadening is required in real space, and this is calculated from the number of sites in the y-direction. This is so that an edge mode will be smoothly continued as it crosses a bulk gap, rather than consisting of individual peaks. It is possible to override this by provided a value in `broadening`.

The parameters `y_only`,`no_transform`,`complete_mat`,`no_invert`,`only_omega0` are described in `FourierTransformSpin()`. However, their values determine the filename of which to save. The filename is:

```
<base><comp><onlyomega0><noinvert>_real.npz
```

see `TI_FourierCTAUX()` for more information.

The option `use_U0` exists to allow a zero self-energy to be chosen. However, it is still required that a `info.dat` file already exists. See `GenerateAllDataForU0()`.

This functions also determines the peaks using `SavePeaksTI()`. The arguments `redo_peaks`, `redo_all`, `min_weight`, `max_weight`.

RDMFT_TISM.**FourierTransformSpin**(*U*, *mu*, *gamma*, *SE*, *potential*, *omega*, *(N, M)*, *boundaries*, *t*, *(p, q)*, *symmetry=None*, *y_only=False*, *no_transform=False*, *only_omega0=False*, *complete_mat=False*, *no_invert=False*, *output_filename=None*, *matsubara=None*)

Calculate the transformed Green's functions with a given self-energy. The resultant functions are not returned and are saved to a `.npz` file with filename given by `output_filename`. See *Reading .npy and .npz files* for more information.

The system parameters `U`,`mu`,`gamma`,`(N,M)`,`boundaries`,`t`,`(p,q)` are described in `GreensMatSinglePython()`.

The frequencies on which to calculate are given in `omega`. These may be either Matsubara or real frequencies. If Matsubara frequencies are used, then one should also set `matsubara=True` as described below.

The lattice symmetries, as given by `symmetry` and used by `SE` and `potential` is described in `SelfConsistencySpin()` and `TI_Symmetry()`. On top of this, it is necessary to know the symmetry of $G^*_{\sigma,\bar{\sigma}}(\omega)$ which is different for real and Matsubara frequencies. Hence, one must specify `matsubara` appropriately.

If `y_only==True` then the transformed functions are G(x,k_y). If `y_only==False` then the transformed functions are G(k_x,k_y). If `no_transform==True` then no Fourier transform is performed at all.

If `only_omega0==True` then the omega value closest to zero in the specified list `omega` is used. This is useful for looking at behaviour near the Fermi edge.

Normally, only the diagonal part of the total Green's matrix is returned, however one can access the entire matrix with `complete_mat=True`. Note that this is slower to calculate.

One can also access the single-particle Hamiltonian by specifying `no_invert=True`.

Notes:

The input `mu` should be adjusted appropriately, if the self-energies have been calculated with the CT-AUX difference of `mu-U/2` (i.e. this function does not do any adjustment, unlike `SelfConsistencySpin()`. See `TI_FourierCTAUX()` for an example.

RDMFT_TISM.**FourierTransformSpinSingle**(*SE*, *potential*, *(N, M)*, *boundaries*, *omega*, *(p, q)*, *mu*, *gamma*, *no_invert=False*, *y_only=True*, *complete_mat=False*, *t=1.0*)

Calculate the Green's function in real-space and then Fourier transform it to k-space.

See `GreensMatSinglePython()` for details about the arguments `SE,potential,(N,M),boundaries,omega,(p,q),mu,gamma,no_invert`.

With `y_only = True` (the default) this will only transform y->ky and leave x untouched.

The Fourier transform is $\int dxdx' G_{x,x',\ldots} exp^{-ikx+ik'x'}$ of which only the diagonal part, `k=k'` is returned.

Returns:

mat (complex double array)

- With `complete_mat==False`, this is only the diagonal elements

RDMFT_TISM.**TI_ContinueCTAUX_SE**(*use_U0=False*, *num_omegaA=400*, *covar_factor=1.0*, *use_additional=None*, *num_average=5*, ***keys*)

Continue all of the calculated Matsubara frequency self-energies into real-frequency spectra. These will then be saved into `SE_real.npz`. If that file already exists, then this function is not run. Additional information is stored in the text file `SE_real_info.dat` and some plots with details are saved to `SE_real_imag.png` and `SE_real_real.png`.

`covar_factor` is the artificial variance to give to the self-energy data. `num_omegaA` is the number of grid points in the real frequency grid to use.

The -U/2 factor that is added in the CT-AUX solver is adjusted at this point.

The continuation itself is performed in `ContinueSelfEnergy()`, and this function parallelizes the execution over the sites.

Any additional information for the MaxEnt can be passed via `**keys`.

An example `SE_real_info.dat` file is given below:

```
Info file for continued self-energy
num_T 400
num_omegaA 400
omegaA_limits -10.0 10.0
covar_factor 0.01
beta 20.0
------------------------------
Additional keys for MaxEnt:
model 4

------------------------------
Info dict read from info.dat:
mu 1.0
num_iters 29
final_mu 1.0
solver 'CT-AUX'
num_omega 400
Nup_diff 0.0020400000000000001
row_sym 'AFwholerow'
M 60
beta 20.0
N 23
boundaries 'OP'
p 1
num_sweeps 10000000.0
U 2.0
q 6
lambda_x 0.75
force_filling None
gamma 0.25
t 1.0
```

RDMFT_TISM.**TI_ContinueCTAUX_Gk**(*num_omegaA=400*, *covar_factor=1.0*, ***keys*)

Continue the Fourier transformed Matsubara Green's functions to real-frequency. The `covar_factor` is the artificial variance to give to the data.

Any additional information for the MaxEnt can be passed via `**keys`.

After testing, it was found that the continuation from this form does not work as well as continuing the self-energies directly. This is because there are far fewer self-energies to continue and the results are therefore more consistent.

This function is not suitable for casual use. All details are hard-coded and may not function as expected. Please use `TI_ContinueCTAUX_SE()` instead.

RDMFT_TISM.**ContinueSelfEnergy**(*SE_iw*, *real_omega*, *num_T*, *beta*, *covar_factor*, *bias_covar=False*, *show_figures=False*, ***keys*)

This function takes a single self-energy `SE_iw` in Matsubara frequency and then analytically continues it to real frequencies, on the grid specified in `real_omega`.

Because the self-energy is not sampled in the CT solver, it is necessary to impose an artificial variance in order to perform the continuation. This is `covar_factor`, and should be varied to determine the best fit to the data. Values that are too small will generate spikey spectra, whereas values that are too large will lose information and blur out the spectra. I have been working with values in the range `0.01 < covar_factor < 1.0`.

One can also specify `bias_covar==True`, in which case the covariance will be biased towards small values of `T`. Use this only for testing purposes.

To perform the continuation, the input is Fourier transformed into imaginary time, of which a grid

of num_T may be specified (it is recommended that this is set to the same size as SE_iw although smaller values can speed up the continuation). The inverse temperature beta is also required here.

Notes:

- If the input is zero, the continuation will not be performed and zero will be returned.
- As the model functions in the continuation are normalised to unity, the input is first renormalised, and after continuation the result is again corrected. This normalisation comes from $G(\tau = 0^+) - G(\tau = 0^-)$.
- If the normalisation of the input is found to be less than 10^-8, it is assumed to be zero.
- The function should be passed with both positive and negative frequency components. I.e. SE_iw[0] is $G(-i\omega_N)$ where N is the largest frequency.
- The $1/\omega$ term is factored out of the input with a polynomial fit in inverse powers of $1/\omega$ to the last 200 points. This is necessary for a smooth Fourier transformation.
- From the polynomial fit, the constant mean-field term is also extracted and then added later onto the result, as this is arbitrary in the Kramers-Kronig relations.
- The final 'spectra' of the continuation is transformed into the full real-space self-energy with the Kramers-Kronig relations

The actual continuation is done by the function Continuation.ClassicalMaxEnt(). To pass any additional arguments to this function, supply them in **keys.

Returns:

full_SE (complex double array)

RDMFT_TISM.**ReadInfo**()

This is a convenience function to access common information about the output of a job. It reads the info.dat file and performs some simple calculations that can then be easily accessed in a dictionary.

The information returned currently includes the system parameters:

N,''M'',''U'',''mu'',''num_sweeps'',''num_omega'',''row_sym'',''p'',''q'',''beta'',''t'',''lambda_x'',''force_filling'

as well as the following:

num_iters The number of iterations performed. Note that this is not the requested number of iterations.

final_mu The final chemical potential at the last iteration. If force_filling==None then final_mu==mu.

Nup_diff The largest difference of a single site's up-spin filling from the last iteration. See ConvergenceSpin().

Returns: info (dictionary)

RDMFT_TISM.**ReadGkData**(*spin='up'*, *xdata=False*, *kkdata=False*, *complete_mat=False*, *r_deriv='N'*, *no_invert=False*, *real_freq=False*, *directfft=False*)

This is a convenience function to read the data for spin spin from the appropriate file of the form:

```
<base><comp><onlyomega0><noinvert>.npz
```

See TI_FourierCTAUX() for details about the file name and arguments when real_freq==False and see TI_FourierCTAUXRealFreq() for details when real_freq==True.

`spin` can be either one of 'up','down','cross' or 0,1,2 respectively.

The argument `directfft` is designed for continuation of the Green's function directly (see `TI_ContinueCTAUX_Gk()`). It should not be used.

Returns: `omega,trans`

omega (complex double array) The frequencies (Matsubara or real) of the Green's function.

trans (complex double array) The Green's function itself as an array, which will be many-dimensional.

RDMFT_TISM.**AverageData** (*first_iter=-5*, *last_iter=-1*)

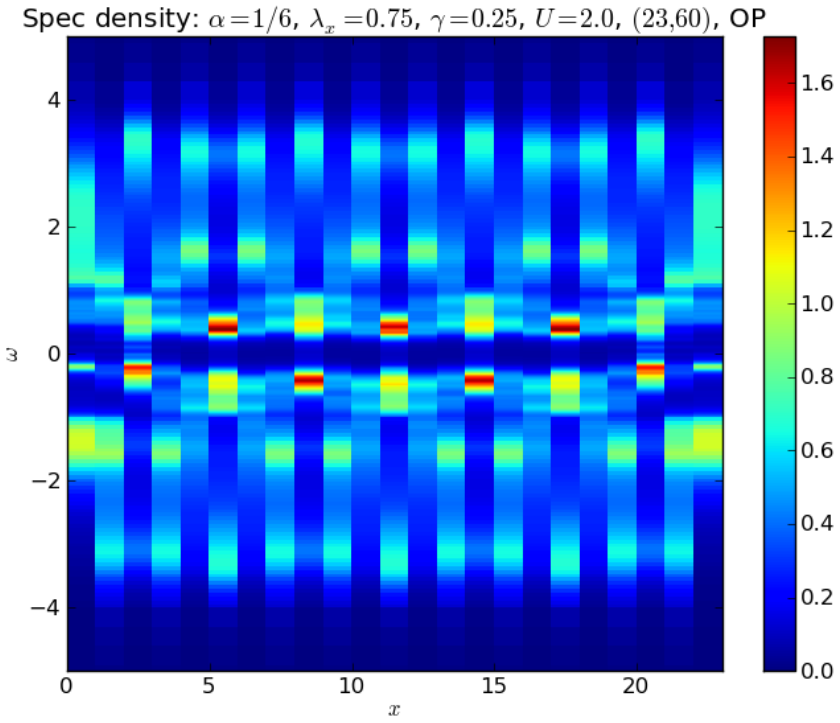
A convenience function to average some iterations of the self-consistency loops. Normally this will be the last 5 iterations, unless overridden with `first_iter` and `last_iter`. These parameters should be given as if python indices to a list of iterations.

RDMFT_TISM.**PlotGkData** (*spin*, *x=None*, *map=False*, *directfft=False*)

Plot some basic graphs from the G(x,y) Green's function data saved with `TI_FourierCTAUXRealFreq()`.

The `spin` parameter is passed directly to `ReadGkData()`.

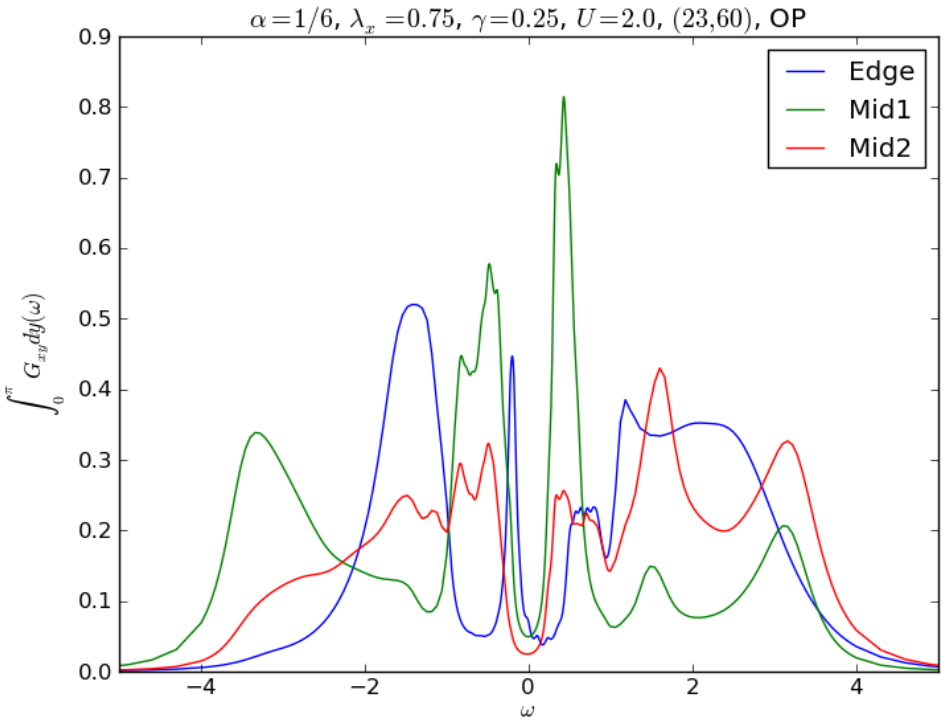
If `map==True` then `x` is ignored and a colormap of the spectra, integrated over `x` is plotted. An example is shown below:



If `map==False` then $ImG(x)$ is plotted for either a specific x==''x'' or for all values of x when `x==None`. An example that results from the following code:

```
PlotGkData('up',x=0)
PlotGkData('up',x=int(info['N']/2))
PlotGkData('up',x=int(info['N']/2 + 1))
```

is shown below:



Note that these figures are not saved. To save these figures, after calling this function, one can use the follow code (as an example):

```
savefig('spec_dens_ky.png')
```

RDMFT_TISM.**PlotPeaksMap**(*spin='up'*, *x=None*, *directfft=False*)

Plot some graphs from the G(x,k_y) Green's function data saved with `TI_FourierCTAUXRealFreq()`.

The `spin` parameter is passed directly to `ReadGkData()`.

The graphs are a color map of the spectra of the Green's functions integrated over x: $\int G(x, k_y)dx$. Depending on the value of `x`, different regions are integrated over with respect to x.

- `x==None` Integrate the entire region.
- `x=='mid'` Integrate the middle third of the lattice.
- `x=='lefthalf'` Integrate the left half of the lattice.
- `x==<int>` Do not integrate, but plot only $G(x = x, k_y)$.

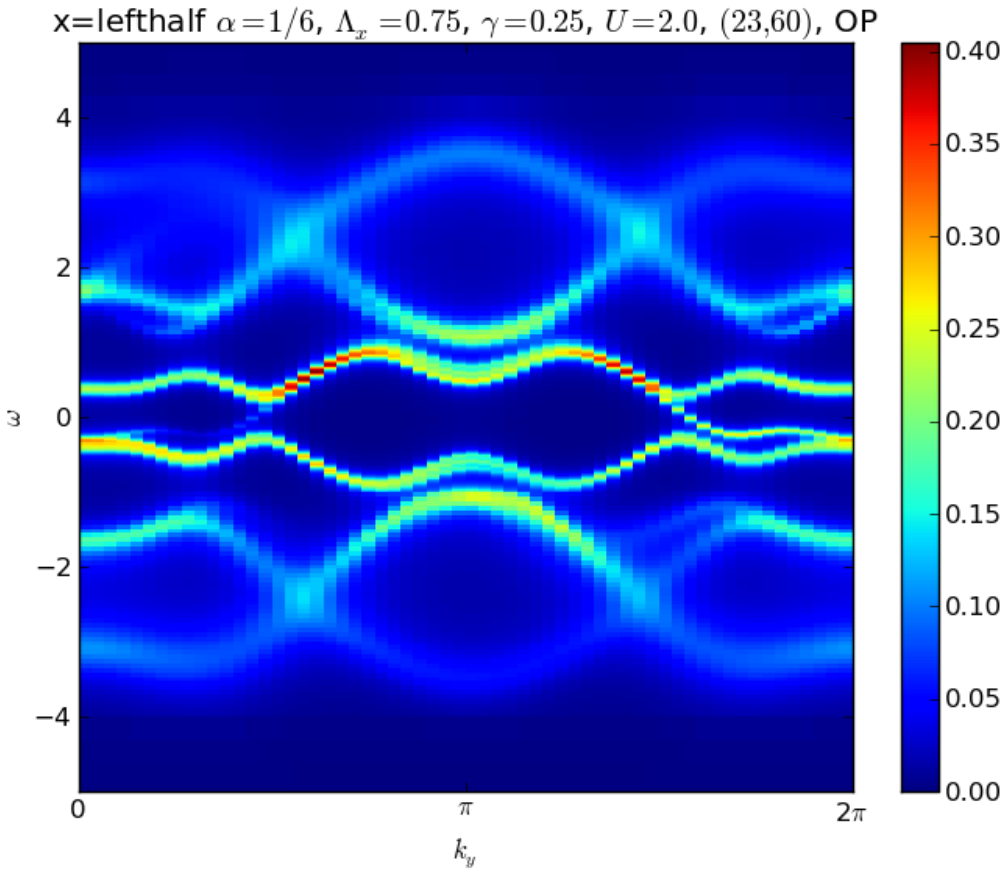
An example of `x=='lefthalf'` is shown below:

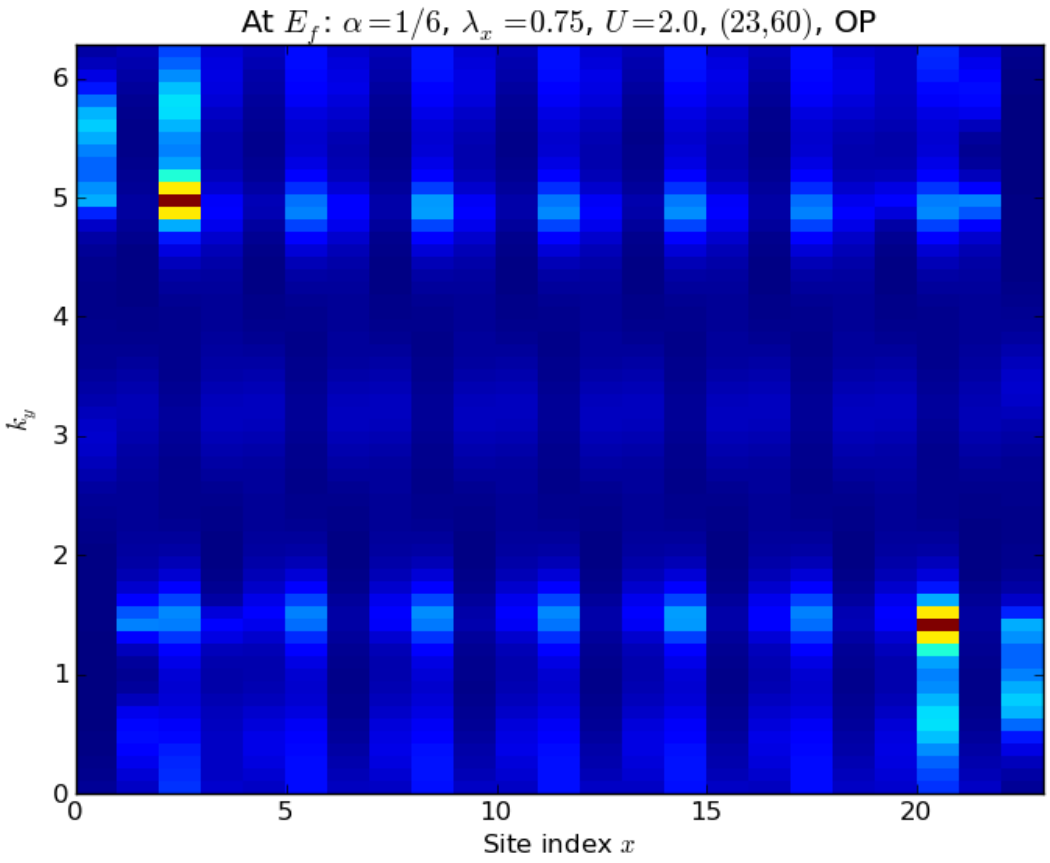
Note that the name of this function is misleading and does not actually have anything to do with the Peaks functions.

RDMFT_TISM.**PlotOmega0Data**(*spin='up'*, *omega_index=None*, *map=True*, *filename=None*)

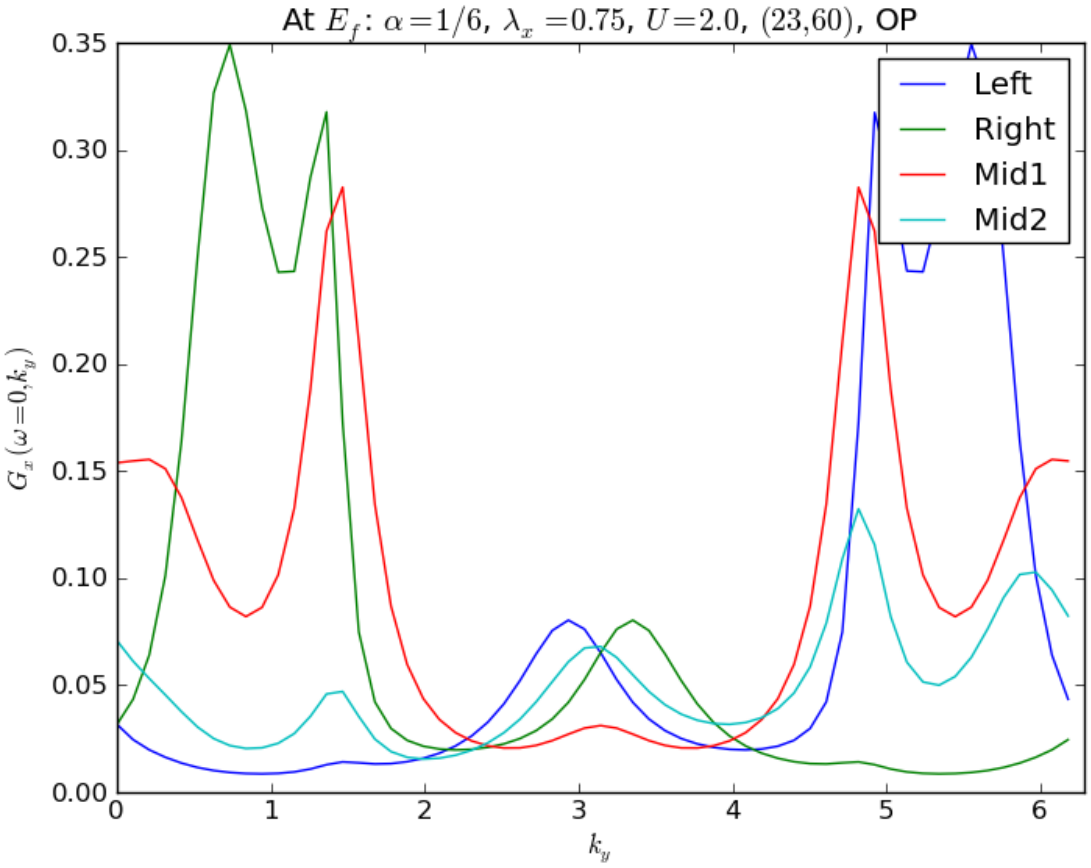
This function will plot the spectra at omega=0 with both x and k_y resolution.

If `map==True` then this will be shown as a color map with all details. An example is shown below:


x=lefthalf $\alpha=1/6$, $\Lambda_x=0.75$, $\gamma=0.25$, $U=2.0$, $(23,60)$, OP
ω
4
2
0
−2
−4
0
π
2π
k_y
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00


At E_f: α=1/6, λ_x =0.75, U=2.0, (23,60), OP
k_y
0
1
2
3
4
5
6
0
5
10
15
20
Site index x

If `map==False` then four separate values of x will be chosen, and the corresponding $G(\omega = 0, k_y)$ functions plotted. These values are the left side, right side, and two central points. An example follows:



The idea of this function was to attempt to identify the transition points to metallic/insulating phases with a more quantitative approach. Unfortunately it cannot be done so easily due to the inherent broadening in the calculations.

`RDMFT_TISM.`**`SavePeaksTI`**(*spin*)

This function iterates through all of the spectra in x,ky-space and saves the peaks and weights found to the file 'peaks<spin>.pickle.gz' where <spin> is the given `spin`.

These files are saved in pickle format, with the entries: `file_version`: `x,ky,peaks` with

`x`: (double array) list of x points. `ky`: (double array) list of ky points. `peaks`: (list of lists of [double array,double array]) `peaks[x][ky][0][peak_i]` is the peak with weight `peaks[x][ky][1][peak_i]`

Returns: `x,ky,peaks`: These values are as is saved above.

`RDMFT_TISM.`**`FindPeaksWithWeights`**(*omega*, *f*)

This function takes a spectra `f` with frequencies `omega` and looks for peaks. Peaks are defined to be at turning points in the spectra, calcualted from the sign change of the first derivative.

Also returned is a value indicating the 'strength' of the peak. This weight is larger for taller and sharper peaks. The function also returns the locations of minima in the spectra for which the weight

is negative.

Returns: `points,weights`

`points` (double array) Frequencies of peaks.

`weights` (double array) Weights of peaks.

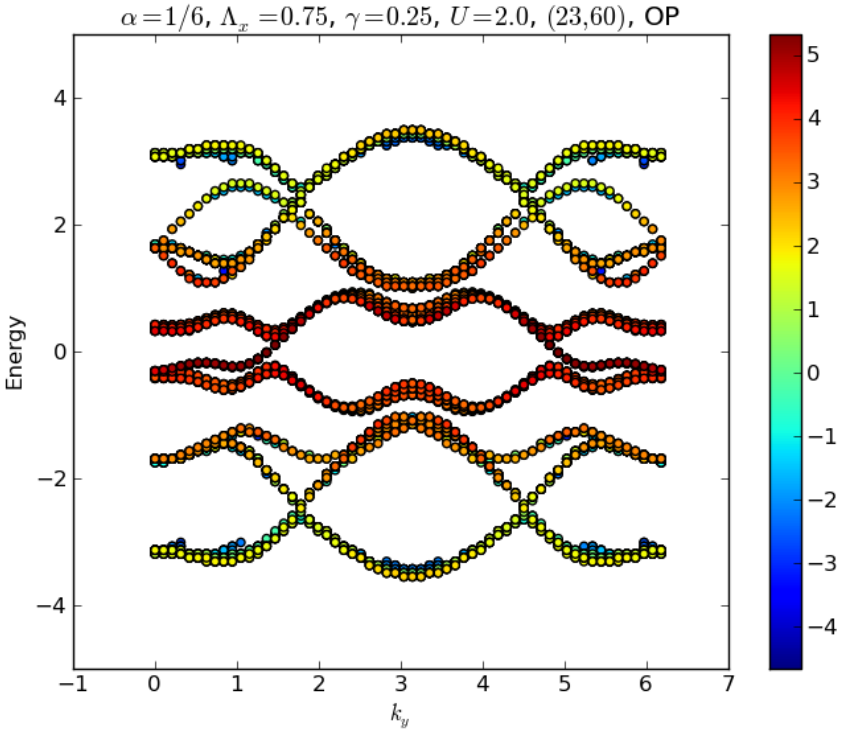
`RDMFT_TISM.`**`PlotPeaksWithWeights`**(*input=None*, *color=None*, *min_weight=None*, *max_weight=None*, *spin='up'*, *x=None*)

This function takes the peaks calculated by function `FindPeaksWithWeights()` and saved by `SavePeaksTI()` and display them as a scatter plot using colours to indicate the weights.

One can safely call this function with no arguments, if `SavePeaksTI()` has already been called.

For additional configuration, one can pass a set of `kx,ky,peaks` calculated from `FindPeaksWithWeights()` as `input`. The colour of each peak can be set via an array `color`. Peaks can be omitted based on `min_weight` and `max_weight`. And the peaks corresponding to only one value of `x` can also be specified.

An example of a plot from this function is below:



`RDMFT_TISM.`**`FitMagExp`**(*num_iters=70*, *show_figures=True*, *also_last=False*)

This is a convenience function to extrapolate the magnetisation calculated in the DMFT iterations to infinite iteration. It assumes the magnetisation can be found in the `info.dat` file in the second last column.

`RDMFT_TISM.`**`CalcNZInvariant`**(*spin='up'*)

Calculate the integer topological invariant of the effective spin-less system. This is the Chern number which is proportional to the quantum Hall current and is applicable to the QSH effect only when S_z is a good quantum number.

The formula is:

$$Z = (2\pi)^2 \int k_x k_y z G^{-1} \partial_{k_x} G G^{-1} \partial_{k_y} G G^{-1} \partial_z G$$

where z are Matsubara frequencies and $G = G(k_x, k_y, z)$.

Warning: This code is currently not working.

Note that this is only relevant for periodic boundaries.

RDMFT_TISM.**GenerateAllDataCTAUX**(*use_U0=False*, *directfft=False*, *num_average=5*)

This is a convenience function to generate all the common graphs that are saved for one run of the code.

If `use_U0==True` then a zero self-energy will be given to all the respective functions. Otherwise `num_average` is the parameter that will designate how many of the previous iterations will be averaged. See `AverageData()`.

The following files will be saved:

- `SE_matsu_imag.png`
- `SE_matsu_real.png`
- `peaks_up.png`
- `spec_dens_ky_left.png`
- `spec_dens_ky_lefthalf.png`
- `spec_dens_ky_bulk.png`
- `spec_dens_ky.png`
- `edge_mid_comp.pdf`
- `edge_mid_comp.png`
- `spec_dens.pdf`
- `spec_dens.png`
- `Ef_ky_edge_bulk_comp.png`
- `Ef_map.png`

as well as the following if `directfft==True`:

- `spec_dens_ky_directfft.png`
- `spec_dens_ky_directfft_left.png`
- `spec_dens_ky_directfft_bulk.png`

RDMFT_TISM.**GenerateAllDataForU0**(*N*, *M*, *boundaries*, *mu*, *p*, *q*, *lambda_x*, *gamma*, *beta*, *only_info=False*)

In order to generate output from the analysis functions, one requires a `info.dat` file. However, for U=0 it is not necessary to run `ConvergenceSpin()` and so one can create the necessary files with this function.

The necessary arguments are `N,M,boundaries,mu,p,q,lambda_x,gamma,beta` as described by `GreensMatSinglePython()`. There are some default parameters, which are:

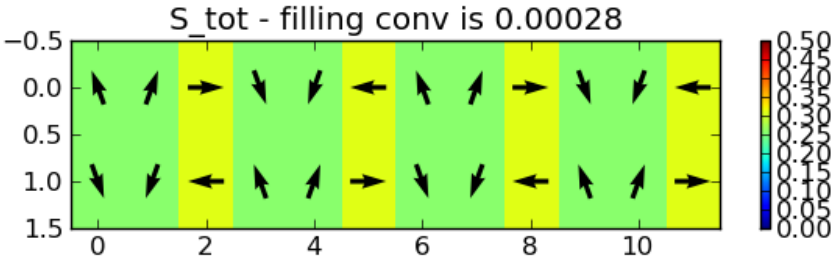
- `num_omega = 400`

- t = 1

This function calls GenerateAllDataCTAUX() to do the actual work. If you would like to only create the info.dat file, then specify only_info==True.

RDMFT_TISM.**DisplayFillingFromNup**(*filename*, *rowsym_override=None*, *vmin=0.0*, *vmax=0.5*)

Given a particular file with filename, saved in the Nup_Ndown_<iter>.dat format that ConvergenceSpin() does, this function will plot the filling, and various spins as an array of squares. An example is below:



The range of the colorbar can be specified with vmin and vmax. As well, one can override the number of blocks displayed, with the rowsym_override parameter, by passing something that is expected by the TI_Symmetry() function.

Note that the function doesn't display the entire grid, but only the small region specified by rowsym in the info.dat files or rowsym_override.

The following files are saved:

- Nftot_map.png
- Sz_map.png
- Sx_map.png
- Sy_map.png
- Stot_map.png

RDMFT_TISM.**JudgeValidity**(*num_average=5*)

This function tests the sanity and validity of the results from a TISM run.

It will distinguish between being in the RDMFT runs, or the continuation part, by looking at the directory name ('covar*'). If it is in a continuation directory then it will (in future) run further tests.

The current tests are:

- Existing iterations.
- Force filling converged.
- For ED: filling convergence.
- Negative imaginary diagonal self-energy (and consequently real spectra).

Returns: validity,mesg

validity (boolean) mesg_list (list of strings): Reasons when invalid.

RDMFT_TISM.**TI_AnotherIter**(*high_steps*, *overwrite_old=False*)

This function exists, so that one may perform an additional iteration at the conclusion of the calculation, with a greater number of sweeps in the CT solver.

The function will determine the original parameters from the `info.dat` file using `ReadInfo()` and apply `high_steps` instead of the normal `CT_sweeps` parameter.

The output is then saved into the file `additional.npz` along with some information in the file `additional_info.dat`. See *Reading .npy and .npz files*.

The argument `overwrite_old` is made available, so that one may request not to run this function if some `additional.npz` file already exists.

RDMFT_TISM.**FitMagExp**(*num_iters=70*, *show_figures=True*, *also_last=False*)

This is a convenience function to extrapolate the magnetisation calculated in the DMFT iterations to infinite iteration. It assumes the magnetisation can be found in the `info.dat` file in the second last column.

CHAPTER

TWO

INTERFACE FOR CT-AUX SPIN-MIXING SOLVER

This module interfaces to the C++ code for the CT-AUX solver, that includes possible spin-mixing processes. In addition, some DMFT routines are included.

By default, the code will look in the following path for the C++ program:

```
~/work3/ccode/Michael_TISM_CT/Spin-mixin/danny_interface
```

If one would like to specify a different path, then one can modify the global variable of the module `program_path` to point to a different location, or one can set the environment variable `CTAUX_CODE_PATH` before starting python.

By default, all the output from the CT-AUX code is suppressed so that the messages from the python code can be identified. If one wants to make this available, set the global variable `global_print_output = True`.

`CT_AUX_SM.`**`CT_AUX_SM`**(*U*, *beta*, *weiss_input*, *num_sweeps*, *use_tmpdir=True*, *combine_Gw=False*, *K=5.0*)

Interface with the C++ code to perform the CT-AUX calculation.

`U` and `beta` are the interaction and inverse temperature respectively. The hybridisation is speficied through the Weiss Green's functions in `weiss_input`. This should be a tuple of three functions, `(weiss_up,weiss_down,weiss_cross)`.

Note that the CT-AUX solver requires a shift of the global chemical potential by U/2. However, as this solver does not take a parameter `mu`, it is necessary to pass an appropriate `weiss_input` that takes this into account.

The number of sweeps is given by `num_sweeps` and the `K` parameter influences the expansion order of the solver, which can improve the speed of the solver in the case of low interaction. However, the default value of `K=5` is recommended.

The option `combine_Gw` exists to allow a mixing of the imaginary time and Matsubara frequency measurements. However, it is recommended to leave this off, as the Matsubara frequency measurements currently behave well for all frequencies.

By default, this function will make a temporary directory to run in, so that no pollution of working directories occurs. This can be disabled with the `use_tmpdir` option.

As it is theoretically possible in the spin-mixing case, for the sign problem to arise, this function raises an exception if the sign difference to 1 becomes great than 0.01.

Returns: `G,GT,pert_order,dbl_occ,WT,G_orig,Gw`

G (tuple of three complex double arrays) The mixed Matsubara frequency Green's functions (equiv to `Gw` with `combine_Gw==False`).

GT (tuple of three complex double arrays) The imaginary time Green's functions.

pert_order (double) The average perturbation order of the CT-AUX.

dbl_occ (double) The double occupancy calculated from the filling and the perturbation order.

WT (tuple of three complex double arrays) The imaginary time Weiss Green's functions.

G_orig (tuple of three complex double arrays) The Matsubara Green's functions Fourier transformed from the imaginary time outputs.

Gw (tuple of three complex double arrays) The Matsubara Green's functions measurements.

CT_AUX_SM.**CT_AUX_SM_Parallel**(**args*, ***kwds*)

An extension of the `CT_AUX_SM()` to run the solver over many cores.

All parameters are the same as `CT_AUX_SM()`. The iterations will be split evenly over the different cores.

Returns:

Same as `CT_AUX_SM()`.

CT_AUX_SM.**CompareWithED**(*U=1.0*, *beta=1.0*, *num_omega=100*, *show_figures=True*, *num_sweeps=100000.0*)

Perform a comparison with the ED solver given a set of Anderson parameters. For large enough sweeps, these values should agree exactly.

CT_AUX_SM.**CompareWithOldCT**(*U=1.0*, *beta=1.0*, *num_omega=100*, *show_figures=True*)

This code is used to compare the solver to Danny's implementation in C++ for the case of no spin-mixing.

CT_AUX_SM.**DMFT**(*U*, *mu*, *beta*, *num_sweeps=10000000.0*, *Bethe_eqn=False*, *fit_large_SE=True*, *allow_AF=True*, *DoS='Bethe'*, *max_iters=20*, *K=5.0*)

Perform a basic DMFT iteration, which considers only a single unit cell.

Given the parameters `U,mu,beta`, the self-consistency starts from a random Weiss Green's function and runs for `max_iters` with `num_sweeps`, and `K` passed to `CT_AUX_SM_Parallel()`. The type of density of states can be chosen with `DoS` to be one of:

- 'Bethe'
- 'square'
- 'cubic'

In the case that `DoS=='Bethe'` then specifying `Bethe_eqn=True` will use the Bethe equation, rather than the Hilbert transform of the DoS.

`allow_AF` allows for asymmetric up and down components, but this does not correctly lead to AF order. Please use `DMFT_TwoSubLattice()` instead.

Returns: `weiss_w,greens_w,GT,dbl_occ,SE`

CT_AUX_SM.**DMFT_TwoSubLattice**(*U*, *mu*, *beta*, *num_sweeps=10000000.0*, *Bethe_eqn=False*, *fit_large_SE=True*, *DoS='Bethe'*, *max_iters=20*)

This is a two sub-lattice version of `DMFT()`. The parameters are identical to that function.

This function differs in that it doesn't return any values, but instead saves each iteration to a file `data_iter<iter>.npz`. See RDMFT_TISM for how to open this type of file.

CHAPTER

THREE

ANALYTICAL CONTINUATION CODES

This module provides functions to analytically continue Green's functions (or other data) given in imaginary time to real-frequency spectra. So far, only a Maximum Entropy method is implemented.

Notes:

1. Data must be transformed to imaginary time. That is, Matsubara frequency data cannot be given to the functions.
2. At the present time this code relies on a couple of other helper libraries for multiprocessing. If you are to use the code that requires this, please contact Danny.

The major function of interest to the end-user here is `ClassicalMaxEnt()`. The function `FitAlpha()` also describes some of the process behind the method.

Changelog (partial listing):

> **13/10/2011: Implemented the option to increase the number of search vectors** used in the algorithm

---

`Continuation.`**`BatchedCTData`**(*first_iter*, *last_iter*, *steps*, *beta*, *omega*, ***keys*)

Temporary function to continue a set of data individually, and then combined the results. The idea is to smooth over spurious fluctuations in the resultant spectra.

`Continuation.`**`BryansMaxEnt`**(*T*, *G*, *covar*, *beta*, *omega*, *guess=None*, ***keys*)

Perform the MaxEnt procedure, but instead of finding the spectra that maximises the probability function with respect to one alpha, return the spectra that is averaged from many values of alpha around the peak. I.e. this function returns

$$\langle A \rangle = \int d\alpha A_\alpha \Pr[\alpha|\bar{G}]$$

where A_α is the spectra that maximises the probability $\Pr[A|\bar{G}, \alpha]$.

Notes:

- The name of this function is a bit of a misnomer. The Bryan's method itself refers to a search procedure in maximising the probability function. However, it has also been used to refer to the averaging procedure.
- Internally this calls `ClassicalMaxEnt()` and passes all addition keywords to it.

- This function is not recommended: the results are nearly always the same to the naked eye as from the plain `ClassicalMaxEnt()` call, and take much more time to run.

`Continuation.`**`ClassicalMaxEnt`**(*T*, *G*, *covar*, *beta*, *omega*, *guess=None*, *start_alpha=100.0*, ***keys*)

This is the main function. It performs the search for the alpha that maximises the probability function $\Pr[A|\bar{G}]$.

The search combines both a bisection method and an iterative search based on the equation for a maximum of the probability function (see `FitAlpha()`). The bisection method takes priority in the search, but new values of alpha are chosen from a substitution into the equation of the maximum where possible.

Returns: `omega,A,alpha`

`omega` (real vector) The omega spacing of the spectra.

`A` (vector) The spectra.

`alpha` (scalar) The value of alpha that was reached for convergence.

Notes:

- It is assumed that the `alpha_diff` results from `FitAlpha()` are monotonically increasing. Each call to `FitAlpha()` is assumed to have converged to the maxima for that value of alpha.
- This function allows for any optional keywords that can be passed onto the `FitAlpha()` function.
- One can uncomment a portion of this function that allows for gradual increase of accuracy in the calls to `FitAlpha()`.
- This function takes the covar matrix instead of the inverse.

`Continuation.`**`ClassicalSmoothedMaxEnt`**(*T*, *G*, *covar*, *beta*, *omega*, *guess=None*, *smoothing_factor=1.0*, ***keys*)

This function does two normal classical MaxEnt runs with a smoothing step in between. The smoothing is applied as a convolution with a Gaussian of width `smoothing_factor`.

It turns out that this is not such a good idea. If the smoothing makes a big difference, then in all likelihood the functions have not actually converged correctly. However, it is possible that the smoothing can avoid a local maxima, and allow the second MaxEnt run to converge to the global maxima.

`Continuation.`**`FitAlpha`**(*T*, *G*, *covar_inv*, *beta*, *omega*, *guess*, *alpha*, *model=1*, *disp_info=0*, *prior='inverse'*, *max_iters=500*, *covar_diag=False*, *tolerance=0.0001*)

Find the spectra that maximises the probability function $\Pr[A|\bar{G},\alpha]$ for one particular value of alpha.

This is base worker function of the MaxEnt procedure. It takes as input the Green's function via `T` and `G` for a system at inverse temperature `beta`, with the covariance information as an inverted matrix in `covar_inv`.

The continued spectra is found on a grid `omega` using the specified value of `alpha`. The procedure maximises the probabilty function for the spectra, starting from an initial spectra of `guess`. The model used is taken from the type specified by the argument `model`:

`model = 0` A normalised flat model, $m(\omega) = 1/(\omega_N - \omega_0)$.

`model = 1` A Gaussian model, $m(\omega) = e^{-\omega^2/\sigma^2}/(\sqrt{2\pi}\sigma)$. Currently with $\sigma = 2$

`model = 2` A flat model, normalised to pi for testing purposes.

The bias for the likelihood of alpha values is given in `prior`. It can be either "none" for no bias or "inverse" for a $1/\alpha$ bias. This seems to be useful if the maximisation procedure keeps trying to make alpha go to infinity.

`max_iters` sets a limit to the maximisation iterations.

At the end of this function, a new guess for alpha is calculated from the maximisation equation $-2S\alpha = \sum_n \lambda_n/(\alpha + \lambda_n)$ where λ_n are the eigenvalues of the Hessian matrix (second derivatives with additional A factors).

Returns: `omega,A,new_alpha,res,converged,alpha_diff`

`omega` (real vector) The omega spacing of the spectra.

`A` (vector) The spectra.

`new_alpha` (scalar) A guess for the next alpha value to try, using the above-mentioned maximisation equation.

`res` (scalar) The exact value of $\Pr[A|\bar{G},\alpha]$ (up to a proportionality on $\Pr[\bar{G}]$).

`converged` (boolean) Whether the function converged to within the tolerance before reaching the maximum number of iterations.

`alpha_diff` (scalar) The difference of the LHS and RHS in the maximum equation as described above. Used in `ClassicalMaxEnt()` to implemented the bisection method.

`tolerance` (scalar) This is the numerical tolerance for the stopping condition of the maximisation loop. The summed difference of the spectra between at two points, 20 iterations apart, are compared as well as the relative difference of the Q value.

Notes:

- If `guess` is `None` then the starting guess will be a spectra very close to the model.
- The model chosen should not significantly affect the results. If it does then there is a larger problem than the choice of model.

Todo

Create an optimised version for when the covariance information is diagonal (e.g. for faked variance information on self-energies). At the moment, `covar_diag` is allowed, but this still extracts the diagonal elements from the matrix which might be a bit slow. Try extracting once at the beginning only.

Todo

An attempt at implementing the Bryan's search method is included here, however it has not been fully successful. If this function gets too slow this should be looked at again.

`Continuation.`**`IterativeModel`**(*first_iter*, *last_iter*, *steps*, *beta*, *omega*, ***keys*)

Temporary function to test an iterative model-update for a batched set of data.

There is a danger that this function does not converge when the model becomes too much like the result.

`Continuation.`**`LoadCTData`**(*first_iter*, *last_iter*, *steps*)

Convenience function to load data from the `*.pickle.gz` DMFT output.

`Continuation.`**`MaxQuad`**(*coeff0*, *coeffs1*, *coeffs2*)

Simple function to solve the basic gradient descent equations. Also handles singular matrix exceptions.

CHAPTER

FOUR

IMAGINARY TIME AND MATSUBARA FREQUENCY CODES

This module provides various functions for defining imaginary time grids or Matsubara frequencies, as well as Fourier transformation of functions between the two.

ImagConv.**FreqToImagTime**(*omega*, *orig*, *T*, *beta*, *leading_order_coeff=1.0*)

This function converts the function `orig`, given in Matsubara frequencies `omega` (which should be the real ω_n values), into imaginary time at locations `T`. The inverse temperature `beta` is a required argument.

The optional keyword `leading_order_coeff` provides a better Fourier transform for Green's functions (or any function with a leading $1/(i\omega_n)$ component. For Green's functions, the default of `leading_order_coeff = 1` is always true, however for other functions, such as self-energies, this should be given explicitly.

ImagConv.**GetT**(*numT*, *beta*, *unevenT=False*)

Return the imaginary time grid with `numT` points for an inverse temperature of `beta`.

If `unevenT` is `False` then this grid is simply linearly spaced between 0 and `beta`. For `unevenT = True` however, the grid is logarithmically spaced and symmetric about `beta/2`.

ImagConv.**ImagTimeToFreq**(*T*, *orig*, *omega*, *beta*)

Fourier transform the function `orig` on the imaginary time grid `T` to Matsubara frequencies that are given by the ω_n values of `omega` (i.e. these are real). The inverse temperature `beta` is a required argument.

ImagConv.**ImagTimeToFreqUnitary**(*T*, *orig*, *omega*, *beta*)

Perform the Fourier transformation using the exact unitary opposite of the forwards transformation (faster and should be more reliable). Note: This will not work with non-linear spacing of T.

ImagConv.**OmegaN**(*numw*, *beta*)

Return a vector of the real value of ω_n for the first `numw` Matsubara frequencies at an inverse temperature of `beta`.

ImagConv.**WeissDiag**(*omega*, *mu*, *V*, *eps*)

This function calculates the inverse Weiss Green's function for the Anderson impurity model using the given `V` and `eps`. The inverse temperature is `beta` and the chemical potential `mu`. Note that this should always be inverted before use.

CHAPTER

FIVE

BLOCK MATRIX INVERSION CODES

This module implements a more explicit inversion for the case of a matrix that is defined on tri-diagonal blocks, as commonly appears in tight-binding models. It also allows for periodic boundary conditions in these models, which have an additional block in the corners of the matrix. These matrices look something like:

$$\boldsymbol{G}^{-1} = \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{B} & 0 & \dots & 0 & 0 & \boldsymbol{B}* \\ \boldsymbol{B}* & \boldsymbol{A}_2 & \boldsymbol{B} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \boldsymbol{B}* & \boldsymbol{A}_{M-1} & \boldsymbol{B} \\ \boldsymbol{B} & 0 & 0 & \dots & 0 & \boldsymbol{B}* & \boldsymbol{A}_M \end{bmatrix}$$

where the individual blocks are not tridiagonal:

$$\boldsymbol{A}_x = \begin{bmatrix} G_{xM}^{-1} & -t_x & 0 & \dots & 0 & 0 & -t_x^* \\ -t_x^* & G_{xM+1}^{-1} & -t_x & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -t_x^* & G_{xM+1}^{-1} & -t_x \\ -t_x & 0 & 0 & \dots & 0 & -t_x^* & G_{xM}^{-1} \end{bmatrix}.$$

Note that with spin-mixing included in the topological insulators system, B is not diagonal and cannot be simulataneously diagonalised with B*.

For more details of the derivation of these methods, please see Danny's notes.

`BlockMatrixInv.`**`BlockedInverseOpen`**(*A*, *B*, *full_matrix=False*)

This function calculates the diagonal elements of the inverted matrix using a blockwise inversion technique.

The matrix is given by m lots of $k \times k$ blocks of `A` along the diagonal, and the $k \times k$ block `B` on all of the first upper diagonal, and `B*` on all of the first lower diagonal, making the matrix tridiagonal in total.

Returns: If `full_matrix` is specified then a $(mk) \times (mk)$ matrix is returned, otherwise a list of m $k \times k$ matrices is returned.

Notes: This inversion procedure scales as m^3k instead of the standard inversion scaling of $(mk)^3$.

`BlockMatrixInv.`**`BlockedInversePeriodic`**(*A*, *B*, *full_matrix=False*)

This function is identical to `BlockedInverseOpen()` but with additional `B` and `B*` terms in the corners of the matrix.

BlockMatrixInv.**NormalInverse**(*A*, *B*, *periodic=False*, *B_inv=False*)

This function is a comparison function, that creates the complete matrix used in `BlockedInverseOpen()` or `BlockedInversePeriodic()` depending on the given value for `peroidic` from the given `A` and `B` and then inverts this using the standard numpy inversion routines. It also returns the uninverted matrix.

The `B_inv` parameter is for testing purposes and includes an additional transformation of B^{-1}.

Returns: `mat,matinv`

mat (matrix) The uninverted matrix.

matinv (matrix) The inverted matrix.

BlockMatrixInv.**CompareWithExact**(*m*, *k*, *periodic=False*)

This function compares the exact inversion solution with the blockwise calculation. It will randomly generate complex `A` and `B` matrices, such that the blocks are of size `k` x `k` and there are a total of `m` blocks on the diagonal. The code is then run for one of the above functions depending on the value of `periodic`. The calculations are also timed.

CHAPTER

SIX

DMFT SELF-CONSISTENCY CODE FOR CT-AUX CODE

Note that this module is designed for Danny's CT-AUX solver only. There are also some other convenient functions for manipulating the Green's functions defined in other modules. This module implements DMFT for the CT-AUX solver with some various support functions for analysis of the results.

SS_CT.**BetheGreensFromSE**(*omega*, *SE*)

A convenience function to determine the Green's function from a given self-energy SE on the given Matsubara frequencies omega.

Do not use this function - it is implemented more rigourously in another module.

SS_CT.**CalcKineticEnergy**(*SE*, *beta*, *U=None*)

An old form to calculate the kinetic energy. Do not use.

SS_CT.**CalcKineticEnergy2**(*SE*, *beta*, *U=None*)

This function calculates the kinetic energy corresponding to a Hubbard model for a given self-energy SE at an inverse temperature beta. If the interaction is given in U, then the self-energy is first extended to very large omega to assist in the reaching convergence in the Mastubara sum.

Note that this function uses the form of the calculation that factorises out the non-interacting part, which is then summed over exactly.

Returns: Ek

SS_CT.**MonteCarloComplete**(*U*, *mu*, *beta*, *K=5.0*, *delta_tol=9.9999999999999995e-07*, *init_weiss_w=None*, *get_covar=True*, *numT=400*, *numw=400*, *init_weiss_T=None*, *max_iters=20*, *num_sweeps=10000000.0*, *match_weiss=False*, *allow_AF=False*, *fit_large_SE=False*, *global_updates=True*, *t=1.0*, *unevenT=False*, *DoS='Bethe'*)

This is function that performs the DMFT self-consistency loops.

The main arguments are given by U, mu and beta. The initial state may be initialised by either a Matsubara frequency Green's function (init_weiss_w) or an imaginary time function (init_weiss_T). In either case, the size of the imaginary time grid and the number of frequencies are given by numT and numw respectively.

There is no convergency check performed here, and the code simply runs for max_iters (an obsolete variable delta_tol originally provided this functionality). Information is saved to a file MC_info.dat in the current directory.

To allow for antiferromagnetic order, enable allow_AF. To fit the self-energy to its analytical asymptotic form, enable fit_large_SE (this is currently fixed to fit values past $\omega_n > 30$). The

density of states used can be changed with `DoS`, which can be set to 'Bethe', 'cubic' or 'square' (note that the cubic and square densities are simply read from a file).

Other variables that are specific to the CT-AUX code are: `get_covar`, `num_sweeps`, `global_updates`, `unevenT`, `t`, `K`. The documentation for these is described in the CT-AUX C++ code.

Returns: `weiss_w,self_energy,(delta_avg,delta_min,delta_max),weiss_T,covar,GT,T`

`weiss_w` (complex vector) The weiss Green's function in Matsubara frequency.

`self_energy` (complex vector) The self-energy in Matsubara frequency.

`delta_avg, delta_min, delta_max` (reals) Relative difference of the Weiss Green's functions of the last two iterations (average, minimum and maximum respectively).

`weiss_T` (real vector) The Weiss Green's function in imaginary time.

`covar` (real matrxi) The covariance matrix of the Green's function.

`GT` (real vector) The Green's function in imaginary time.

`T` (real vector) The imaginary time grid used.

- *genindex*
- *modindex*
- *search*

INDEX

R

S

T

W