Using CorrelationConnection

First, follow the instructions on README.md to install and activate the right Conda enviornment. Once that has been done, you can launch this Jupyter Notebook.

Load necessary libraries:

In [1]:
```
import pickle
import sys
sys.path.append("./bin/")
sys.path.append(".")
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
```

Tell your system where it can find executables:

In [2]:
```
sys.path.append("../../CorrelationConnection/bin/")
```

The line above may change depending on your working directory. The exact way it is written above works if your working directory is a subfolder of `CorrelationConnection` (e.g. `CorrelationConnection/docs`) and `CorrelationConnection` contains another sub-folder called `bin` containing the files `exact_diagonalisation_code.py`, `exact_diagonalisation_code_sparse.py`, etc.

Import some more libraries:

In [140…
```
import scipy
import scipy.sparse.linalg as sprsla
from hamiltonians import NNHamiltonian
from randomizer import RandomizerHamiltonianNNRandomDelta, Randomi
from stability_analysis_class import StabilityAnalysisSparse
from tools import SijCalculator
from exact_diagonalisation_code_sparse import create_sx_sparse, cr
from math import sqrt, pi
```

Let's now get started.

To begin, we will create a Hamiltonian object for `L` sites with interactions up to next nearest neighbours at temperature `temp=0`. This is achieved with the class `NNHamiltonian(L, h, J_onsite, J_nn, J_nnn, temp=0)`. To keep things simple we will make the system a dimer and we will not apply a magnetic field:

In [141…

```
# 1) SET VALUES FOR INPUTS:

L=2                                          # <-- Dimer
h = [[0, 0, 0]]                              # <-- No applied field
J_onsite = np.zeros((3, 3))                  # <-- No onsite interac
J_nnn = np.zeros((3, 3))                     # <-- No nnn interactio
J_nn = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]     # <-- Diagonal nn inter

# 2) CREATE THE HAMILTONIAN:

HAMILTONIAN = NNHamiltonian(L, h, J_onsite, J_nn, J_nnn, temp=0)
```

`NNHamiltonian(L, h, J_onsite, J_nn, J_nnn, temp=0)` is one of three classes of Hamiltonians. The others can be found in `bin/hamiltonians.py`.

Note that the object is not just the Hamiltonian. It includes the temperature as well. The Hamiltonian itself can be extracted using `get_init_ham()`:

In [142…

```
H=HAMILTONIAN.get_init_ham()
```

The Hamiltonian `H` is not an array, it is a sparse matrix object (from `scipy.sparse`). Therefore, only the matrix elements that are non zero are stored. If we try to print `H` we will get a list of its non-zero elements:

In [143…

```
print(H)
```

```
  (0, 0)        (0.5+0j)
  (1, 1)        (-0.5+0j)
  (1, 2)        (1+0j)
  (2, 1)        (1+0j)
  (2, 2)        (-0.5+0j)
  (3, 3)        (0.5+0j)
```

Note the factor of "2" in the definition of the Hamilonian.

Let us now find the ground-state energy and state vector:

In [144…

```
GROUND_STATE=SijCalculator.find_gs_sparse(H)
print("Ground state energy = ",GROUND_STATE[0],"\nGround state vec
```

```
Ground state energy =  [-1.5]
Ground state vector =
 [[-0.  +0.j  ]
 [ 0.28-0.65j]
 [-0.28+0.65j]
 [ 0.  +0.j  ]]
```

Note that we have used np.round to write the grounds state with only 2 decimal points for clarity.

We can also work with a dense-matrix representation:

In [145…
```
H_DENSE=H.todense()
```

In [146…
```
print(H_DENSE)
```

```
[[ 0.5+0.j  0. +0.j  0. +0.j  0. +0.j]
 [ 0. +0.j -0.5+0.j  1. +0.j  0. +0.j]
 [ 0. +0.j  1. +0.j -0.5+0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0.5+0.j]]
```

`find_eigvals` can now be used to find all the energies:

In [147…
```
ENERGIES = SijCalculator.find_eigvals(H_DENSE)
```

In [148…
```
print(ENERGIES)
```

```
[ 0.5+0.j -1.5+0.j  0.5+0.j  0.5+0.j]
```

Now we can carry out the fitness landscape analysis.

In order to do this we need to set a few parameters that specifiy the type of fitness landscape we want to explore and how.

First, we need to set up a randomizer. There are several available, they are all in `randomizer.py` . Here we use `RandomizerStateRandomDelta` which creates random state vectors the real and imaginary parts of whose amplitudes are picked from a flat probability distribution centred on those of a given input state.

In [149…
```
delta= 0.5
no_of_processes= 64
RANDOMIZER = RandomizerStateRandomDelta(HAMILTONIAN, delta, no_of_
```

Now, set up our exploration of the fitness landscape using `StabilityAnalysisSparse` from `stability_analysis_class.py` :

In [150…
```
corr = ["Sxx", "Sxy", "Sxz", "Syx", "Syy", "Syz", "Szx", "Szy", "S
temp= HAMILTONIAN.temp  # <-- Obtain temperature from the Hamilton
save_rhams= "False"     # <-- Do not save random Hamiltonians/stat
temp_type= "value"      # <-- Temperature given explicitly
```

In [151…
```
STABILITY_ANALYSIS = StabilityAnalysisSparse(HAMILTONIAN, RANDOMIZ
```

Finally, we generate the data by using `generate_random_Sij_sparse` which is found in `stability_analysis_class.py` :

In [153…
```
no_of_samples= 5
dist, en, diffSij, diffSq, Sqs, Sijs, Sqints, Sq_int=EXAMPLE=STABI
```

The output is a list of 8 objects so instead of storing it in a single variable we have directly stored it in 8 conevniently-named variables. Let us now see what each of them contains.

`dist` contains the distances of the random states to the ground state of the reference Hamiltonian:

In [154…
```
dist
```

Out[154…
```
[0.0016824121457618135,
 0.00029454398505979196,
 0.016930044619712148,
 0.03324221366399849,
 0.12442835046299772]
```

`en` contains the corresponding ground state energies:

In [155…
```
print(en)
```

```
[array([0.00336482+0.j]), array([0.00058909+0.j]), array([0.033860
09+0.j]), array([0.06648443-1.11022302e-16j]), array([0.2488567+0.
j])]
```

`diffSij` contains the distances between real-space correlators:

In [156…
```
diffSij
```

Out[156…
```
[0.002278734315252842,
 0.0009534602108327649,
 0.00722863916203547,
 0.010129137420771941,
 0.019596890536841664]
```

`Sqs` contains the reciprocal space expressions of all the random correlators, computed on a 100×100 grid in $\mathbf{q}$-space. To be more explicit, let

$$\rho_{i,j}^{\alpha,\beta}\left[\psi_n^{\text{rand}}\right] \equiv \left\langle \psi_n^{\text{rand}} \middle| S_i^\alpha S_j^\beta \middle| \psi_n^{\text{rand}} \right\rangle$$

represent the correlator between the component α of the spin at site i and the component β of the spin at site j ($\alpha, \beta = x, y, z$; $i, j = 0, 1, \ldots, L-1$). We define the reciprocal of this quantity as

$$\tilde{\rho}^{\alpha,\beta}\left[\psi_n^{\text{rand}}\right](\mathbf{q}) \equiv \sum_{i,j=0}^{L-1} e^{i\mathbf{q}.(\mathbf{R}_i - \mathbf{R}_j)} \rho_{i,j}^{\alpha,\beta}\left[\psi_n^{\text{rand}}\right]$$

Note that this is not the same as a Fourier transform, as the quantity $\rho_{i,j}^{\alpha,\beta}\left[\psi_n^{\text{rand}}\right]$ does not depend only on $\mathbf{R}_i - \mathbf{R}_j$ but in general depends on $\mathbf{R}_i$ and $\mathbf{R}_j$

independently. Indeed, this will be the case for a magnetic molecule sitting somewhere in space, since the structure is not periodic in this case. For the same reason, the vector $\mathbf{q}$ is not restricted to a discretised lattice. It represents the momentum transferred between the sample and the neutron, which can vary continuously. In the code we discretise the values of the two-dimensional vactor $\mathbf{q} = (q_x, q_y)$ by allowing 100 values of q_x and 100 values of q_y, each uniformly distributed between -2π and $+2\pi$ (giving 10000 values of the wave vector $\mathbf{q}$ in total).

TO DO: Generalise this, it should be possible to change how fine the q-grid is and also to extend the q values further out e.g. to 3π or focus on a smaller part of q-space, e.g. if we are looking at a big molecule there will be lots of detail at small q.

The structure of `Sqs` is best illustrated through a couple of examples. For instance, in our case,

```
Sqs[3]["Sxy"][23][76]
```

contains the value of $\tilde{\rho}^{x,y}\left[\psi_3^{\mathrm{rand}}\right](\mathbf{q})$ at a wave vector $\mathbf{q}$ given by the coordinates $(23, 76)$ on our discretised reciproclal grid:

In [167…
```
print(Sqs[3]["Sxy"][23][76])
```

```
(-0.005867325933629966-0.0045669562629151j)
```

Similarly

```
Sqs[2]["Syy"][10][35]
```

contains the value of $\tilde{\rho}^{y,y}\left[\psi_2^{\mathrm{rand}}\right](\mathbf{q})$ at a wave vector $\mathbf{q}$ given by the coordinates $(10, 35)$:

In [168…
```
print(Sqs[2]["Syy"][10][35])
```

```
(0.18493480478284136+0j)
```

Note that the xy correlator is a complex number, while the yy correlator is real. This makes sense: $\tilde{\rho}^{x,y}\left[\psi_3^{\mathrm{rand}}\right]$ is not the expectation value of a Hermitian operator, and in general the sum over i and j will have terms of the form

$$\langle S_i^\alpha S_j^\beta \rangle e^{i.(\mathbf{R}_i - \mathbf{R}_j)} + \langle S_i^\beta S_j^\alpha \rangle e^{-i.(\mathbf{R}_i - \mathbf{R}_j)}$$

The two expectation values are both real but in general they are different, and therefore the imaginary parts of the two exponentials do not cancel. For $\alpha = \beta$, they do.

`Sijs` contains the values of real-space correlators, for instance `Sijs[3]`

`['Sxz'][0][1]` is the value of the xz correlator betweent the 1st site and the 2nd site in the 3rd random state, $\langle \psi_3^{\text{rand}} | S_0^x S_1^z | \psi_3^{\text{rand}} \rangle$:

In [170…
```
Sijs[3]['Sxz'][0][1]
```

Out[170…
```
(-0.031917074874466+0j)
```

Sinally, `Sqints` and `Sq_int`:

In [193…
```
print("Dimensions of Sqints:",len(Sqints),len(Sqints[0]),len(Sqint
```

```
Dimensions of Sqints: 5 100 100
```

In [190…
```
Sqints[4][99][99]
```

Out[190…
```
0.4767306353738547
```

In [194…
```
Sq_int
```

Out[194…
```
[0.00020491065213567046,
 8.032773536748678e-06,
 0.060670416995147844,
 0.26579703465365156,
 1.1449728311755316]
```