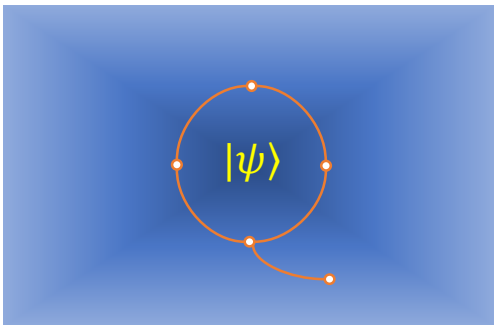


QNEURAL

INTRODUCTION TO QNEURAL PYTHON MODULE

Carlos Pedro Gonçalves

11-11-2019

CONTENTS

1. ABOUT QNEURAL

QNeural is a Python module for the simulation of quantum artificial neural networks (QuANNs). The module is comprised of two main libraries:

- **svect.py:** this is a library that includes main functions using NumPy matrix functionalities, applying them to quantum computation, including the possibility of representing computing results using the Dirac *brak-ket* formalism. Unitary and conditional unitary transformations, density operators and von Neumann entropy are all included so that the library can be implemented for basic design and testing of quantum computation;
- **qneural.py:** this is a library for the design and simulation of quantum neural networks (this library uses **svect**'s functionalities);

QNeural was built based on Dirac's *bra-ket* notation but it also allows one to work with density matrices which, in the case of QuANNs, is particularly useful due to the entangled dynamics that these networks exhibit, one can derive a reduced density and perform von Neumann entropy calculations on a density. The module is aimed, in particular, at the iterative simulation of neural networks introduced in [1].

1.1. Authorship and Copyright

Author: Carlos Pedro dos Santos Gonçalves

e-mail: cgon.aulas@gmail.com

1.2. Citing

To cite the package the following reference should be used:

- Gonçalves, C.P. (2019), QNeural.

1.3. Structure of the Manual

The present text is intended as a brief manual for a quick introduction into the main elements and functionalities of both libraries. In **section 2**, the usage of **svect.py** library is addressed. **Section 3** reviews the main functionalities included in the **qneural.py** library, for the design and simulation of quantum neural networks.

2. THE LIBRARY 'SVECT'

In the present section of the user manual we review the main functions of the library **svect** used for the design and simulation of quantum computation.

2.1. Function basisDef

The function **basisDef** takes as input the number of quantum registers and produces, as output. a list with the symbolic representation of the basis using Dirac's *bra-ket* notation. It is mainly used as an auxiliary function in other functions of **svect**, and **qneural** libraries. In the case of computation using *ket* vectors, rather than density operators, it is used to produce the output of a quantum circuit in Dirac's *bra-ket* notation. The following box contains two examples of a basis with two and three quantum registers:

```
In [1]: import svect

In [2]: svect.basisDef(2)
Out[2]: ['|00>', '|01>', '|10>', '|11>']

In [3]: svect.basisDef(3)
Out[3]: ['|000>', '|001>', '|010>', '|011>', '|100>', '|101>', '|110>', '|111>']

In [4]:
```

Box 1

2.2. Functions getKet, showKet, density, density_vect and entropy

The function **getKet** takes the basis and a list of amplitudes and returns a *ket* object, defined as a Python list with two elements, the basis is element 0, the column vector of the amplitudes is element 1. The **showKet** procedure takes as input a *ket* object and prints the resulting *ket* in Dirac's notation, as exemplified in an example in the box shown below, for an equal superposition *ket* vector.

```
In [1]: import numpy as np

In [2]: import svect as sv

In [3]: basis = sv.basisDef(1)

In [4]: ket = sv.getKet(basis,[1/np.sqrt(2),1/np.sqrt(2)])

In [5]: sv.showKet(ket)
|psi> = 0.7071067811865475|0>+0.7071067811865475|1>
```

Box 2

The **basisDef** function can also be called from inside the **getKet** function, the following box shows the same output as in the previous box but with the **basisDef** function called from inside the **getKet** function.

```
In [1]: import numpy as np

In [2]: import svect as sv

In [3]: ket = sv.getKet(sv.basisDef(1),[1/np.sqrt(2),1/np.sqrt(2)])

In [4]: sv.showKet(ket)
|psi> = 0.7071067811865475|0>+0.7071067811865475|1>
```

Box 3

If we want to get the density matrix corresponding to a specific *ket* vector, we can extract it using the function **density**. The function **density** takes as input a *ket* object and outputs a density matrix, this matrix is actually a NumPy matrix. *Box 4* shows an example of the use of this function expanding from *Box 3*'s code.

If, instead of a *ket* object, which is used for the symbolic Dirac's *bra-ket* notation, we wish to work with the column matrix directly, we can extract the density directly from the column matrix using **density_vect** function, that takes as input a NumPy column matrix and outputs a density matrix. *Box 5* shows an example for the same density as in *Box 4*.

```
In [1]: import numpy as np

In [2]: import svect as sv

In [3]: ket = sv.getKet(sv.basisDef(1),[1/np.sqrt(2),1/np.sqrt(2)])

In [4]: sv.showKet(ket)
|psi> = 0.7071067811865475|0>+0.7071067811865475|1>

In [5]: rho=sv.density(ket)

In [6]: print(rho)
[[0.5 0.5]
 [0.5 0.5]]
```

Box 4

```
In [1]: import numpy as np

In [2]: import svect as sv

In [3]: vector = np.matrix([1/np.sqrt(2),1/np.sqrt(2)]).getT()

In [4]: print(vector)
[[0.70710678]
 [0.70710678]]

In [5]: rho = sv.density_vect(vector)

In [6]: print(rho)
[[0.5 0.5]
 [0.5 0.5]]
```

Box 5

2.3. Functions for Unitary Transformations and Projectors

The library **svect** contains a collection of functions that allow the user to implement unitary transformations on *ket* vectors and density matrices, as well as

projectors used for quantum average calculations. For quantum computation, the single quantum register unitary transformations, included in the library, are:

- **PauliX():** returns Pauli's X matrix:

$$\mathrm{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- **PauliY():** returns Pauli's Y matrix:

$$\mathrm{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

- **PauliZ():** returns Pauli's Z matrix:

$$\mathrm{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- **unit():** returns the unit operator on a single qubit:

$$\mathrm{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- **WHGate():** returns the Walsh-Haddamard gate:

$$\mathrm{W} = \frac{\mathrm{Z} + \mathrm{X}}{\sqrt{2}}$$

- **PShift(phi):** returns the phase shift gate:

$$\mathrm{S}_{\phi} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$$

Besides these built-in unitary gates there is the possibility of designing a 2x2 unitary matrix with the decomposition of a phase transformation and a rotation. So that given a single quantum register Hamiltonian operator of the form:

$$\widehat{H} = -\frac{\omega}{2}\hbar I + \theta\frac{\hbar}{2}\left(n_x\mathrm{X} + n_y\mathrm{Y} + n_z\mathrm{Z}\right),$$

where n_j (for $j = x, y, z$) are the components or a real unit vector, we can, then, write the unitary evolution operator for $\Delta t = t_1 - t_0$ as:

$$U(t_1, t_0) = e^{-\frac{i}{\hbar}\Delta t\widehat{H}} = e^{i\frac{\omega\Delta t}{2}}\left(\cos\left(\frac{\theta\Delta t}{2}\right)I - i\sin\left(\frac{\theta\Delta t}{2}\right)\left(n_x\mathrm{X} + n_y\mathrm{Y} + n_z\mathrm{Z}\right)\right)$$

The function **gate2x2(omega,theta,n,deltaT)** returns such an evolution operator, allowing the user to set the Hamiltonian parameters and time lapse Δt.

The library **svect** also allows for one to build tensor products of operators, this is the **tensorProd** function that takes as input the operators list and returns their tensor product. This can be applied both for unitary and density operators.

Besides unitary operators, projectors are also useful, in particular, in producing conditional computational gates, used extensively in quantum neural computation. The following two functions are related to projectors:

- **proj2x2(value):** the ***value*** argument must be Boolean, if it is False then the projector $|0\rangle\langle 0|$ is returned, if it is True, then, the projector $|1\rangle\langle 1|$ is returned;
- **genProj(size):** returns a list of all the projectors onto the computational basis for a number of quantum registers equal to ***size***.

In each case, both of the unitary operators and projectors, the outputs are NumPy's matrices.

Now, given a unitary operator, the following two functions are used:

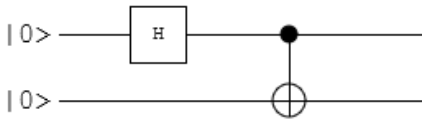
- **transformKet(operator,ket):** takes as input a unitary operator and a *ket* object and returns the transformed *ket* object;
- **transformDensity(operator,density):** takes as inputs the unitary operator and a density matrix and returns the transformed density matrix.

The last function in the **svect** library is the function **entropy(density,printouts)** that takes as inputs a density matrix and a Boolean for printouts, if the Boolean is set to False, then, the function just returns the von Neumann entropy of a density matrix in bits, otherwise, the function also prints the density, the logarithm of the density, the product of the density by the logarithm of the density and the von Neumann entropy. The function takes advantage of the NumPy matrix algebra elements that allows the calculation of the logarithm of a density, the algorithm thus implements the formula, for the von Neumann entropy in bits:

$$S(\hat{\rho}_n(k)) = -\frac{1}{\ln 2} Tr[\hat{\rho}_n(k) \ln(\hat{\rho}_n(k))]$$

2.4. Example: Controlled Not using 'svect'

For an example of a Python program implementing quantum circuits let us consider the following quantum circuit:



Quantum Circuit: Haddamard followed by CNOT

The full code is presented in the next page, the result from running it is presented in the box below:

```
QUANTUM CIRCUIT SIMULATION

Initial ket vector:
|psi> = 1.0|00>

Step 1:
|psi> = 0.7071067811865475|00>+0.7071067811865475|10>

Step2:
|psi> = 0.7071067811865475|00>+0.7071067811865475|11>
```

Analyzing the code, lines 5 to 9, define the main operators and projectors needed, these include the Haddamard gate (defined in line 5), the unit gate, which corresponds to the 2x2 identity matrix (defined in line 6), the Pauli's X gate (defined in line 7), and the two projectors onto the single *qubit* computational basis (defined in lines 8 and 9). In line 12, the first unitary operator is prepared which is the tensor product of the Haddamard gate by the unit gate, since only the first *qubit* is transformed by the Haddamard gate, while the second *qubit* remains unchanged. In line 13 the conditional NOT gate is prepared by a sum of two tensor products, the first is between the projector **P0** by the **unit gate** (this is because when the first *qubit* is 0, the second remains unchanged), the second tensor product is between the projector **P1** and **Pauli's X gate** (that is, when the first *qubit* is 1, the second flips), therefore, in line 13 we have a conditional negation defined in Python code.

Line 16 defines the basis, which is comprised of two quantum registers, while lines 19 and 20 setup the initial configuration to an amplitude where both *qubits* are in

the logical state 0. Line 23 extracts the initial *ket* object. Line 28 prints the initial *ket* using Dirac's *bra-ket* notation. Line 31, transforms the *ket*, implementing the first step of the quantum circuit, line 32 prints the resulting *ket*, then the *ket* is again transformed, in accordance with line 35, that implements the second step of the quantum circuit, while line 36 prints the resulting final *ket*.

```
1 import svect as sv
2 import numpy as np
3
4 # Main Operators Used in the Circuit:
5 H = sv.WHGate() # Walsh-Haddamard transform
6 I = sv.unit() # Unit gate
7 X = sv.PauliX() # Pauli X
8 P0 = sv.proj2x2(False) # Projector P0 = |0><0|
9 P1 = sv.proj2x2(True) # projector P1 = |1><1|
10
11 # Circuit Design:
12 U0 = sv.tensorProd([H,I]) # First state transition in circuit
13 CNOT = sv.tensorProd([P0,I]) + sv.tensorProd([P1,X]) # CNOT gate
14
15 # Basis:
16 basis = sv.basisDef(2) # we are working with a two register basis
17
18 # Initial amplitude:
19 psi0 = np.zeros(len(basis))
20 psi0[0] = 1
21
22 # Initial ket:
23 ket = sv.getKet(basis,psi0)
24
25 # Implementing the quatum circuit:
26 print("\nQUANTUM CIRCUIT SIMULATION")
27 print("\nInitial ket vector:")
28 sv.showKet(ket)
29
30 print("\nStep 1:")
31 ket = sv.transformKet(U0,ket)
32 sv.showKet(ket)
33
34 print("\nStep2:")
35 ket = sv.transformKet(CNOT,ket)
36 sv.showKet(ket)
```

We can simplify the code using fewer lines, basically, if we want to just see the initial and final *ket* vectors, we can write the code below:

```
1 import svect as sv
2 import numpy as np
3
4 # Main Operators Used in the Circuit:
5 H = sv.WHGate() # Walsh-Haddamard transform
6 I = sv.unit() # Unit gate
7 X = sv.PauliX() # Pauli X
8 P0 = sv.proj2x2(False) # Projector P0 = |0><0|
9 P1 = sv.proj2x2(True) # projector P1 = |1><1|
10
11 # Circuit Operator
12 UCircuit = np.dot(sv.tensorProd([P0,I]) + sv.tensorProd([P1,X]),
13                   sv.tensorProd([H,I]))
14
15 # Basis:
16 basis = sv.basisDef(2) # we are working with a two register basis
17
18 # Initial amplitude:
19 psi0 = np.zeros(len(basis))
20 psi0[0] = 1
21
22 # Initial ket:
23 ket = sv.getKet(basis,psi0)
24
25 # Implementing the quatum circuit:
26 print("\nQUANTUM CIRCUIT SIMULATION")
27 print("\nInitial ket vector:")
28 sv.showKet(ket)
29
30 print("\nFinal ket vector:")
31 ket = sv.transformKet(UCircuit,ket)
32 sv.showKet(ket)
```

In this case, we used NumPy's dot product to first multiply the two operators in accordance with the order in which they are applied to the *ket* vector. The **svect** functions are working inside the dot product, to extract its arguments (this is shown in lines 12 and 13), then, only one transformation is needed which is the unitary operator for the circuit applied to the *ket* vector, as per line 31. Running this code gives the correct result:

```
QUANTUM CIRCUIT SIMULATION

Initial ket vector:
|psi> = 1.0|00>

Final ket vector:
|psi> = 0.7071067811865475|00>+0.7071067811865475|11>
```

If we were to print the UCircuit operator we would get the NumPy matrix:

```
[[ 0.70710678  0.          0.70710678  0.        ]
 [ 0.          0.70710678  0.          0.70710678]
 [ 0.          0.70710678  0.         -0.70710678]
 [ 0.70710678  0.         -0.70710678  0.        ]]
```

3. THE LIBRARY 'QNEURAL'

A quantum artificial neural network, or QuANN for short, is a networked quantum computing system with two level firing patterns for each neuron, so that, if we use the computational basis $\{|0\rangle,|1\rangle\}$, the *ket* vector $|0\rangle$ corresponds to a nonfiring neuron, while the *ket* vector $|1\rangle$ corresponds to a firing neuron. These networks' computation involves quantum circuits with conditional unitary gates that follow the network's links. This usually leads to highly entangled dynamics, in this sense, we will work with density operators, and also incorporate methods to get the neuron-level reduced density.

The library 'qneural' has the neural network class, with corresponding instance methods as well as Python functions that are used for simulation and analysis. All example files available at the Github repository https://github.com/cpgoncalves/qneural come from reference [2].

The main class is QNet, an object of this class has as attributes **the number of neurons**, **a density operator**, **projectors for the neural firing basis**, **local neural firing operators**, and **multipliers used for the reduced density calculations**. We will now address each method including ***auxiliary methods*** and ***main methods***. The user will interact with the ***main methods***.

The first method is the ***auxiliary method*** called **local_multipliers_list**, that is used in extracting the local neuron-level reduced densities. It does not depend on the number of neurons, and returns a list of NumPy matrices with the following sequence:

```
[[1 0]
 [0 0]]


[[0 1]
 [0 0]]


[[0 0]
 [1 0]]


[[0 0]
 [0 1]]
```

The second method, **generate_multipliers_neuron**, is also an ***auxiliary method*** that takes as inputs the multipliers list and a neuron index, it is used as an auxiliary method to get the partial trace for a network's density to get a single neuron-level reduced density out of a many neuron density, it returns a list of extended multipliers that are used in order to get the partial trace. This point will be addressed further on, when we review the **local_density** method.

The **setup_basis** method is a ***main method*** that sets up the basis, and main operators, the full script for the method is:

setup_basis(multipliers_list=None, return_ket=True, get_multipliers=False)

The core of the method is to set up the neural firing basis, projectors, local operators and the multipliers attributes, if asked for. The method updates the projectors attribute applying **svect**'s function **genProj** using the number of neurons as input, thus, the projectors attribute will have all the projectors onto the *N-neuron* computational basis.

The local operators correspond to the local neural firing projectors, there is one for each neuron, so that the local operator for the *n-th* neuron has the value of 0 if the corresponding neuron is not firing and 1 if the neuron is firing. These local operators are defined on the *N-neuron* computational basis, and can be evaluated for any density operator on the whole neural network, avoiding the computation of a reduced density for each neuron and then the calculation of the quantum average using the two dimensional space projector.

If one wishes to get local neuron-level reduced densities and analyze the local neurons, then, one needs the neuron-level multipliers extended to the network's firing pattern basis, which means that one needs the **local multipliers list** (obtained from the **local_multipliers_list** method), for the implementation of the **generate_multipliers_neuron** method, in that case, one must supply the **local multipliers list** as input for the **setup_basis** method (by default its value is set to **None**) and set the **get_multipliers** option to **True**. The user can also opt to get an initial nonfiring *ket* to setup the initial network configuration, this is the ***return_ket*** option which is set to **True** by default.

If one does wish to evaluate a local density, for any neuron, for instance, because one wishes to evaluate the von Neumann entropy for the reduced neuron-level density, then one will need, as an auxiliary element, the local multipliers for the reduced densities' calculations the method produces one such multiplier for each neuron.

Thus, the two attributes ***local neural firing operators*** and ***multipliers used for the reduced density calculations***, will consist of two lists, with each list element corresponding, respectively, to the operators and multipliers associated with the respective neuron, such that the list index corresponds to the neuron's index (the quantum register index, running from 0 to *N*-1, in the case of an *N*-neuron neural network).

If one wishes to work with *kets* rather than density operators, or to extract an initial density out of a *ket* vector. Then one uses the return *ket* option, which is the default. The method sets up the initial *ket* vector to the nonfiring *ket* $|00...0\rangle$ and returns it. One can change this *ket* to any other initial *ket* by applying the next ***main method***, which is the **transform_ket** method, this method takes a unitary operator on the network's Hilbert space and the *ket* vector to transform the *ket*, if printouts are asked for, then, the method prints out the *ket*, if the density is asked for, which means that the user will be working with density operators, then, the density operator for the resulting *ket* vector, after the application of the unitary operator on the input *ket* vector, will be extracted and stored in the instance attribute ***rho***, this is the default option. The full script for the method is:

transform_ket(Unitary, ket, density=True, printouts=True)

As can be seen above, the arguments ***density*** and ***printouts*** are both set to True by default. Now, the **local_density** and **calculate_entropy** methods are two ***main methods***, both of them take as inputs the ***multipliers list***, supplied by the **local_multipliers_list** method, and the ***neuron index***, their script is as follows

local_density(neuron_index,multipliers_list)

calculate_entropy(neuron_index,multipliers_list, print_density=False):

The **local_density** method extracts the reduced density matrix for any neuron in the network, from the global density, it uses the two-dimensional Hilbert space ***multipliers list***, the ***neuron index*** and the content of the ***extended local multipliers*** on the *N*-dimensional Hilbert space, for the corresponding neuron, in order to extract the reduced density for the local neuron, returning this reduced density.

The **calculate_entropy** method uses the **local_density** method and returns the von Neumann entropy, measured in *bits*, for the corresponding reduced density, applying the **svect** library. Therefore, the **local_density** method is also an ***auxiliary method*** for the **calculate_entropy** method. If the **print_density** argument is set to True, the **calculate_entropy** method also prints the reduced density for the corresponding neuron.

The next ***main method*** is the **build_quantum_neural_map** method, it takes as input a list of unitary operators corresponding to a quantum circuit that represents a given neural network's computation and builds the corresponding neural map as the sequential product of these operators, the operators are NumPy matrices, and the order in which the operators are supplied in the list follows the order in which they appear in the quantum circuit. The method returns the resulting map operator, which is equivalent to the circuit.

The concept of quantum neural map applies the concept of a unitary quantum map to the neural computational setting and was developed in [1,2], namely, the quantum unitary neural map $\hat{F}$ that reflects the neural network's quantum circuit is defined as:

$$\hat{F} = \hat{U}_C .. \hat{U}_1 \hat{U}_0 \tag{1}$$

The iteration of the map, which corresponds to the iterative activation of the computational circuit, leads to the sequence of densities obeying the rule:

$$\hat{\rho}_{n+1} = \hat{F}\hat{\rho}_n\hat{F}^\dagger = \hat{U}_C..\hat{U}_1\hat{U}_0\hat{\rho}_n\hat{U}_0^\dagger\hat{U}_1^\dagger..\hat{U}_C^\dagger \quad (2)$$

The libraries **qneural** and **svect** are mainly aimed to be used conjointly in order to study and simulate these maps.

The ***main method*** **get_eigenvectors** takes as input a neural map operator, and also a Boolean condition for printouts (the ***printout*** argument is set to False by default), and returns the eigenvalues and the eigenvectors, in that order. The script for the method is:

get_eigenvectors(map_operator, printout=False)

The following three ***main methods*** iterate a given quantum neural map, as introduced in [1]:

- **iterate_ket(map_operator,ket,T,transient):** used for iterating a unitary quantum neural map, given the **map operator**, an initial ***ket*** **vector** (an ***svect ket structure***), the **number of iterations T**, and a **transient** time after which one starts recording the iterations results, the method returns the list of column matrices representing the *kets* stored in a Python list, resulting from the map's iterations.
- **iterate_density(map_operator,T,transient):** the iterate density is the same as the iterate *ket* but instead of a *ket* vector it works with densities, using the neural network's attribute **rho**, the method returns the list for the sequence of *densities* resulting from the map's iterations.
- **iterate(map_operator,T,multipliers_list=None,entropy=False,density =False):** the **iterate** method is more complete than the previous two but slower to implement, the method also iterates the neural map but returns, by default, a NumPy array of sequences of quantum averages calculated with respect to the neural firing operators for each neuron, each column of the array is a quantum average following the sequence of neuron indexes, each line corresponds to the quantum average extracted from the iterations, if the user sets the **entropy** option to True, then, the method also outputs a corresponding matrix with the entropies calculated for each neuron, in this case the user must also provide the **multipliers_list** as input, if, instead,

the user just wishes to extract the densities, then, the **density** option is set to True, in which case the method does not calculate any quantum averages or entropies and reduces to the **iterate_density** method. When the user wants the sequence of quantum averages, this method uses the ***auxiliary method***: **extract_averages** for extracting the quantum averages and entropies (if asked for).

Besides the above methods, the library **qneural** also has a host of functions, some that automate some procedures, others that are used in time series analysis, including recurrence analysis techniques.

The **initialize_network** function takes as arguments the number of neurons and an initial operator that can be either a unitary operator or a density operator, if the initial operator is unitary, then, the function returns an instance of the class QNet, with the **rho** attribute set to the outer product of the *ket*, that results from the application of the initial operator to the nonfiring *ket*, by its conjugate transpose, the function also returns the multipliers list if asked for. The full script for the function is:

initialize_network(num_neurons,initial_operator,type_initial='Unitary', return_multipliers=False)

The argument **initial_operator** must either be a unitary operator or a density operator, if it is a unitary operator then the **type_initial** argument must be set to **'Unitary'**, otherwise it must be set to **'Density'**, the **return_multipliers** is set to **False** by default, but must be set to **True** if the user wishes to extract local densities and entropies further on, in that case, the function not only returns the network, but it also returns the local multipliers list.

If the **initial_operator** is a density then the **type_initial** argument must be set to **'Density'** and, in that case, the **rho** attribute is directly set by the user, without being extracted from a *ket* vector, this is particularly useful when the initial density is mixed rather than pure.

The **get_eigenvectors** function takes as arguments the number of neurons and the neural operators list that represent the quantum circuit, calculates the neural map and prints the eigenvalues, the eigenphases and the eigenvectors for the map, the method also

returns the neural network instance, the local multipliers list and the eigenvectors for further analysis in terms of local entropy. The full script for the function is:

get_eigenvectors(num_neurons, neural_operators)

The **initialize_network_eig** uses as arguments **the number of neurons**, the **neural operators list**, the **eigenvector index** and an additional option on whether the user wants the local multipliers on a single neuron Hilbert space to be returned. The method performs the eigenvalue analysis and returns a network with initial density given by the outer product of the chosen eigenvector and the corresponding conjugate transpose. The function also produces an entropy analysis on this initial density. The full script for this function is:

initialize_network_eig(num_neurons, neural_operators, eigenvector_index, return_multipliers=False)

The **recurrence_matrix** and **recurrence_analysis** functions are to be used conjointly, namely, the **recurrence_matrix** function calculates and plots either a distance matrix (if no radius is supplied) or a recurrence matrix (if a radius is supplied), the colored recurrence plot in the case of a distance matrix is such that darker points represent lower distances and lighter colors represent higher distances, when the recurrence matrix with a radius is extracted then the resulting plot is a black and white recurrence plot, where a point is painted black if the distance is not greater than the radius supplied. The **recurrence_analysis** is applied to a distance matrix producing recurrence stats, for a given radius, that were addressed in [1].

The script for the **recurrence_matrix** function is as follows:

recurrence_matrix(series, radius=None, type_series=1)

The radius is set to None by default, the user can change the radius to any value. If the **type_series** argument is set to the value 1 (which is by default), then the recurrence matrix is being calculated for a one-dimensional series, otherwise, if it is set to zero, then, it is being calculated for an d-dimensional sequence of points, with $d > 1$. **The function only works for these two values.** For a one-dimensional series we use the absolute value

as distance, for a *d*-dimensional sequence of points we use the *d*-dimensional Euclidean metric.

The **recurrence_analysis** script is as follows:

recurrence_analysis(S, radius, printout_lines=False, return_lines=False)

The argument **S** corresponds to the distance matrix, the **radius** is supplied by the user, the **printout_lines** argument (set to False by default), if set to True, then the function prints the lines with 100% recurrence for the chosen radius, the **return_lines** argument (set to False by default), if set to True, then the function returns the lines with 100% recurrence as a Python list for further processing.

The function **calculate_BoxCounting(sequence,max_bins,cutoff)** receives a sequence of values that can be *n*-dimensional, and, given the number of max *bins* and a cutoff of values that specifies a region for the estimation of a straight line, is used to calculate the Box Counting Dimension of a fractal structure in *n*-dimensional Euclidean space. If the cutoff is set to None, the algorithm just calculates the Box Counting dimension for the whole range of values. Since sometimes the power law structures break down in some regions of values, the cutoff defines a lower and upper bound for estimating the fractal dimensions, the user can use the cutoff to only estimate the regression line for values within that range, so as not to bias the estimated dimension.

The function **get_seed(seed_base,n,step)** is used in quantum stochastic process simulation, namely in the simulation of quantum stochastic neural maps. Since we need random numbers to be produced at each iteration step we need an algorithm that outputs random seeds for reproducibility, so that instead of writing a seed for each iteration and storing it, we get a list of random number seeds, as long as we use the same parameters we get the same resulting sequence of random number seeds, and, therefore, the same sequence of random numbers, which makes the experiments reproducible. Changing the parameters leads to a different random seed sequence. The **get_seed** function uses the NumPy RandomState function and sets the seed argument for that function as follows:

np.random.RandomState(seed=seed_base + n * step)

The basic idea is that the value *n* should be set to the iteration number for the quantum stochastic map, the step is a multiplying factor, that the user can control, and the seed base, a constant additive factor that the user can also specify.

1. Gonçalves, C.P. Quantum Neural Machine Learning: Backpropagation and Dynamics. NeuroQuantology. 2017, 15(1), 22-41.
2. Gonçalves C.P. Quantum Stochastic Neural Maps and Quantum Neural Networks. Preprint available as draft paper at https://www.academia.edu/40810532/Quantum_Stochastic_Neural_Maps_and_Quantum_Neural_Networks (all example files for the qneural implementation were taken from this paper).
3. Gonçalves, C.P. Quantum Cybernetics and Complex Quantum Systems Science: A Quantum Connectionist Exploration. NeuroQuantology. 2015, 13(1), 35-48.
4. Braun, D. Dissipative Quantum Chaos and Decoherence. Springer: Berlin, Germany, 2001.