Implementation of ground state search via Neural Network

Federico Agostini, Federico Bottaro

05/04/2020

Abstract

The search for the ground state of a many-body system in physics is a topic in which different approach can be implemented. The birth and development of artificial neural networks in the last years has meant that the many-body problem can be treated with this new concepts. We have that a quantum state can be represented by a neural network [1]. In this work, a reinforcement-learning approach is used to train the network and to extract the ground state energy of a given system. The algorithm is evaluated over 1-D and 2-D transverse-field Ising model. A Lanczos algorithm is used to compare the results of the applications.

1 Introduction

In the field of statistical physics, the behaviour of a complex quantum system can be mimicked using NN such as Restricted Boltzman Machines (RBMs).

A RBM is a particular type of NN in which the bipartite structure is exploited to extract correlation between the visible units (that represent our data) and the internal representation of the data given by the hidden neurons. In the context of statistical physics RBM's can be seen as a particular ansatz for the many-body wave function. To train our NN we can exploit a reinforcement learning approach in which a set of configuration representing the system can be sampled with numerical approaches like quantum Monte Carlo methods.

The goal of this document is evaluating the ground state of a many body system; hence the starting point is the wave function and its representation. Consider a quantum system with N particles, for examples spins, $\mathcal{S} = (s_1, s_2, \dots, s_N)$. The many-body wave function is a mapping of the N-dimensional set $\mathcal{S}$ to complex numbers which fully specify the quantum state. In our context, we want to approximate this black box with a RBM trained to represent $\Psi(\mathcal{S})$.

Focusing on a spin-1/2 quantum system, the visible layer of the RBM is composed by N nodes that corresponds to the physical spins in the chosen basis (in the following $\mathcal{S} = \sigma_1^z \dots \sigma_N^z$), whereas the hidden layer has M auxiliary variables. The quantum states is then described by the expression:

$$\Psi_M(\mathcal{S}, \mathcal{W}) = \sum_{\{h_i\}} e^{\sum_j a_j \sigma_j^z + \sum_i b_i h_i + \sum_{ij} W_{ij} h_i \sigma_j^z}, \tag{1}$$

where $h_i = \{-1, 1\}$ is a set of M hidden units and $\mathcal{W} = \{a_i, b_j, W_{ij}\}$ is the vector of the weights of the network; given an input, they fully specify the response of the network itself. Exploiting the absence of intra-layer connection, we can trace out the hidden variables from the wave function and rewrite it as:

$$\Psi_M(\mathcal{S}, \mathcal{W}) = e^{\sum_i a_i \sigma_i^z} \times \prod_{j=1}^{M} 2\cosh[b_j + \sum_i W_{ij} \sigma_i^z] \tag{2}$$

Since our goal is to evaluate the ground state energy of the system, reinforcement learning proceeds minimizing the expectation value of the energy: $E(\mathcal{W}) = \langle \Psi_M | \mathcal{H} | \Psi_M \rangle \, / \, \langle \Psi_M | \Psi_M \rangle$ with respect to the network weights. In the stochastic setting, this is achieved with an iterative scheme. At each iteration t, a Monte Carlo sampling of $|\Psi_M(\mathcal{S}, \mathcal{W})|^2$ is realized, for a given set of parameters $\mathcal{W}$. At the same time, stochastic estimates of the energy gradient are acquired. In order to obtain an optimal solution for which $\nabla E^* = 0$, Stochastic Reconfiguration (SR) method is used [2].

The implementation of the algorithm is done both with Fortran and Python. The idea behind the code was the same and the functions are tested with both programming languages, in order to evaluate their correctness. For some numerical instability we can present here results obtained only with the Python script. For more detail see Sec. 4.11.

2 Theoretical overview of SR

To reach the goal of finding the ground state energy of a hamiltonian, an iterative procedure is exploited. At each iteration, the value of Ψ_M can be evaluated given the current state of the RBM described by the network parameters $\mathcal{W}$. At the same time, we can sample spin configurations based on $|\Psi_M(\mathcal{S}, \mathcal{W})|^2$, using a Metropolis algorithm. Then the weights of the network are updated using these stochastic estimates, and the process is repeated until convergence.

As marked out in Appendix A of Science 355, 602 (2017) [1], SR weights update at iteration p is:

$$\mathcal{W}(p+1) = \mathcal{W}(p) - \lambda(p) S^{-1}(p) F(p) \tag{3}$$

where λ is the learning rate, S is the covariance matrix

$$S_{kk'}(p) = \langle \mathcal{O}_k^* \mathcal{O}_{k'} \rangle - \langle \mathcal{O}_k^* \rangle \langle \mathcal{O}_{k'} \rangle \tag{4}$$

and F the forces vector

$$F_k(p) = \langle E_{loc} \mathcal{O}_k^* \rangle - \langle E_{loc} \rangle \langle \mathcal{O}_{k'} \rangle. \tag{5}$$

The previous equation introduces two important quantity. The term $\mathcal{O}_k$ is a vector containing the variational derivatives of $\Psi_M(\mathcal{S})$ with respect to the k-th network parameter $\mathcal{W}_k$. Exploiting the description of the wave function of the RBM (Eq. 2) we can explicitly write the components of the vector $\mathcal{O}_k$ as:

$$\frac{1}{\Psi_M(\mathcal{S})} \partial_{a_i} \Psi_M(\mathcal{S}) = \sigma_i^z, \tag{6}$$

$$\frac{1}{\Psi_M(\mathcal{S})} \partial_{b_j} \Psi_M(\mathcal{S}) = \tanh[\theta_j(\mathcal{S})], \tag{7}$$

$$\frac{1}{\Psi_M(\mathcal{S})} \partial_{W_{ij}} \Psi_M(\mathcal{S}) = \sigma_i^z \tanh[\theta_j(\mathcal{S})]. \tag{8}$$

Above we introduced the effective angles, which correspond to

$$\theta_j(\mathcal{S}) = b_j + \sum_i W_{ij} \sigma_i^z. \tag{9}$$

Eq. 6, 7 and 8 produce respectively N, M and $N \times M$ terms.

The second term, introduced in Eq. 5, is the local energy

$$\begin{aligned} E_{loc}(\mathcal{S}) &= \frac{\langle \mathcal{S}|\mathcal{H}|\Psi_M\rangle}{\langle \mathcal{S}|\Psi_M\rangle} \\ &= \sum_{\mathcal{S}'} \frac{\langle \mathcal{S}|\mathcal{H}|\mathcal{S}'\rangle \langle \mathcal{S}'|\Psi_M\rangle}{\langle \mathcal{S}|\Psi_M\rangle} \\ &= \sum_{\mathcal{S}'} \frac{\langle \mathcal{S}|\mathcal{H}|\mathcal{S}'\rangle \, \Psi_M(\mathcal{S}')}{\Psi_M(\mathcal{S})} \end{aligned} \tag{10}$$

where we applied a completeness relation.

To sample spin configurations accordingly to the state of the RBM, Metropolis algorithm is used. Starting from a random configuration $\mathcal{S}$ we flip a spin at random and the new configuration is accepted according to the probability:

$$A(\mathcal{S}^k \to \mathcal{S}^{k+1}) = \min\left(1, \left|\frac{\Psi_M(\mathcal{S}^{k+1})}{\Psi_M(\mathcal{S}^k)}\right|^2\right). \tag{11}$$

3 Theoretical overview of the Lanczos algorithm

Given an hermitian matrix A of size $n \times n$ and a number of iteration m, the Lanczos algorithm provide as output a matrix T, that is tridiagonal and symmetric with dimension $m \times m$. It holds that $T = V^* A V$, where V is a matrix with orthonormal columns of size $n \times m$.

The iterative procedure starts from a normalized random vector v_1 of size n. The fist iteration of the algorithm proceed as follow:

- $w_1' = A v_1$
- $\alpha_1 = w_1'^* \cdot v_1$
- $w_1 = w_1' - \alpha_1 v_1$

and then we can start with the iterative procedure for $j = 2, \ldots, m$:

- $\beta_j = \|w_{j-1}\|$
- $v_j = w_{j-1}/\beta_j$ if $\beta_j \neq 0$ otherwise restart from an arbitrary normalized vector
- $w_j' = A v_j$
- $\alpha_j = w_j'^* \cdot v_j$
- $w_j = w_j' - \alpha_j v_j - \beta_j v_{j-1}$

At the end we have that the tridiagonal matrix T has a diagonal composed by the m terms α_j and the off diagonal made up by the $m-1$ terms β_j. It is possible to demonstrate that the matrix V, which satisfies the equation $T = V^* A V$, has the columns that correspond to the $v_1, \ldots, v_m$ vectors.

Since we are interested in the eigenvalues of a given hermitian matrix A, we exploit the Lanczos algorithm and evaluate the eigenvalues of the matrix T: in fact, A and T have the same eigenvalues, and there are establed procedure to extract this information from a tridiagonal one.

4 Code development

The implementation of the algorithm is achieved both in Fortran and in Python. The reason behind that is to have a double check on the correctness of the functions. Moreover, Fortran code does not produce stable results, even if every single routine is tested against the corresponding one in Python and produces the same output given identical inputs (more on these tests can be found in Sec. 4.11).

In the following, the description takes as example the Fortran code, but in Python functions have same names and logic. The references to the listings with the full code can be found in Appendix B. Results in Sec. 5 however are obtained with the latter one, due to the problem mentioned above.

4.1 RBM initialization

This subroutine randomly initializes the parameters of the RBM, sampling from a normal distribution with mean and standard deviation specified by the user. For more information see Lst. 2.

4.2 Metropolis simulation

The subroutine `Metropolis` in file `simulation.f90` deals with the generation of spin configurations given the state of the RBM (expressed by the current values of the weights $\mathcal{W}$). The description of the parameters can be found in Lst. 1, along with the full code.

Inputs The subroutine requires the RBM parameters (one array for the biases of the visible units, one for the hidden and the matrix of the weights between the two) and how many elements we want to save in the output.

Outputs The outputs are the sampled spin configurations, saved one per row in a matrix and the vectors of the variational derivatives (Eq. 6, 7, 8), saved in a matrix, one per row.

Description Starting from a random configuration (generated by `Random_Configuration` subroutine in file `random.f90`), a random spin is flipped. The new configuration is accepted according to Eq. 11.
The calculus of Ψ_M following Eq.2 involves exponentials and products, and then leads to overflow problems that affects the acceptance ratio. Since we always deal with ratios of these quantities, the implementation uses logarithms to avoid numerical instabilities; in fact it is possible to write:

$$r = \left| \frac{\Psi_M(\mathcal{S}^{k+1})}{\Psi_M(\mathcal{S}^k)} \right|^2 = \left| \exp\left[\log \Psi_M(\mathcal{S}^{k+1}) - \log \Psi_M(\mathcal{S}^k) \right] \right|^2 . \tag{12}$$

Seeing that $\log \Psi_M(\mathcal{S}) = \sum_i a_i \sigma_i^z + \sum_j \log\left(2\cosh\theta_j(\mathcal{S})\right)$, we get:

$$\begin{aligned} &\log \Psi_M(\mathcal{S}^{k+1}) - \log \Psi_M(\mathcal{S}^k) = \\ &\quad = \sum_i a_i(\sigma_{i,k+1}^z - \sigma_{i,k}^z) + \sum_j \log\left[2\cosh\theta_j(\mathcal{S}^{k+1})\right] - \sum_j \log\left[2\cosh\theta_j(\mathcal{S}^k)\right] , \end{aligned} \tag{13}$$

that is what the function `logPsiDiff` returns (Lst. 13).
This process is repeated a number of times equal to the input variable `iter`; the user can also specified the number of iteration to discard before saving the results (through `burnin`) and the number of steps to wait before two successive records (`autocorr`), in order to reduce the autocorrelation between the sampled configurations.

4.3 Covariance Matrix

Function `Skk` (Lst. 4) computes the covariance matrix of Eq.4 and applies a regularization since $S_{kk'}$ could be non invertible.

Inputs The matrix containing the variational derivatives of Ψ_M given as output from the `Metropolis` subroutine and the current iteration of the weights update (in order to apply the regularization) are required.

Output The output is a `DOUBLE COMPLEX` matrix describing Eq. 4.

Description The final matrix is composed by $S_{kk'}$ plus a regularization term $\lambda(p) = \max(\lambda_0 b^p, \lambda_{min})$ on the diagonal terms. Following [1], we set $\lambda_0 = 100$, $b = 0.9$, $\lambda_{min} = 10^{-4}$, while p is the weights update iteration. In order to avoid numerical problems, every element with absolute value lower than 10^{-9} is fixed to 0.

4.4 Local energy

At each spin configuration $\mathcal{S}$ can be associated a local energy as delineated in Eq. 10. The function `LocalEnergy` in Lst. 6 computes this quantity for each configuration generated by the Metropolis algorithm, given the hamiltonian of the system.

Inputs This function requires the configuration matrix given as output from the `Metropolis` subroutine, the matrix representation H of the hamiltonian $\mathcal{H}$ and the RBM parameters.

Output The output is a `DOUBLE COMPLEX` array with local energy values for each configuration passed in input.

Description The code implementation works with the last form of Eq. 10, after the application of the completeness relation.

The term $\langle \mathcal{S} | \mathcal{H} | \mathcal{S}' \rangle$ involves matrix products; however, it is not required to compute them explicitly. In fact, we can rewrite the configuration $\mathcal{S}$ with N spins as a vector with 2^N elements, which components are all 0 except for exactly one 1 in position `i`. Hence, we get $\sum_{\mathcal{S}'} \langle \mathcal{S} | \mathcal{H} | \mathcal{S}' \rangle \cdots = \sum_{j=1}^{2^N} H_{ij} \cdots$, that correspond to the sum of the elements i-th row of matrix H.
To retrieve the index `idx`, we use the function `idx_from_config` (Lst. 10), that converts $\mathcal{S}$ to its integer representation: in practice, this is a conversion from a binary number (here 0 is substituted with -1) to its decimal representation.

The term $\Psi_M(\mathcal{S}')$ requires the spin representation of $\mathcal{S}'$ defined by integer j; to achieve that, the function `config_from_idx` (Lst. 11) is available. However, as already pointed out in Sec. 4.2, the computation of the wave function using Eq. 2 leads to overflow problems. Hence, we compute the logarithm of the ratio $\Psi_M(\mathcal{S}')/\Psi_M(\mathcal{S})$ using `logPsiDiff` (Lst. 13) and then we exponentiate the result.

4.5 Forces

Given the vector with the local energies from `LocalEnergy` function and the matrix with the variational derivatives from `Metropolis`, the calculus of the forces is straight forward applying Eq. 5. Even in this case, elemnts with absolute value lower tan 10^{-9} are set equal to 0, to avoid numerical problems. Code is available in Lst. 5.

4.6 RBM update

Given the results from the previous function, the update of the RBM follows Eq. 3.

Inputs The subroutine requires the current parameters of the RBM along with the covariance matrix and the forces array. The user can also specify the learning rate. For more information, see Lst. 3.

Outputs Biases and weights of the RBM are overwritten with the updated values at the end of the subroutine.

Description As shown in Eq. 3, the covariance matrix needs to be inverted in order to update the weights. To accomplish this requirement, `LAPACK` package is employed (function `Inverse` in Lst. 14). First of all the matrix is decomposed according to the LU decomposition throw `ZGETRF` and then the results are exploited to call `ZGETRI` subroutine, which actually performs the inversion.

4.7 Lanczos algorithm

The code implementation (Lst. 15) of the algorithm follows the steps described in Sec. 3, given the matrix representation of the hamiltonian and the number of steps to perform in order to retrieve the tridiagonal matrix. This one is then passed to `LAPACK` `ZSTMR` subroutine, that calculates the eigenvalues of a tridiagonal matrix.

4.8 `debug` module

This module is used to test and debug the code; since it was already delivered during the course, only a brief description is provided. It consists of several subroutine with a common interface. A boolean variable sets the subroutine to be active and to print the debug message; the quantity to be monitored could be of any fortran intrinsic type, up to rank 2 arrays; optionally a message and an output file can be specified to personalize the debugging string.

4.9 `hamiltonian` module

Also these scripts were developed during the course, and they provides the matrix representation of the Ising hamiltonian in the transverse field.

4.10 `run` script

In order to provide a better interface to the main file of the program, a Python script is developed. It is based on the `argparse` module, which enables a clear command line interface to gather user supplied variables. This arguments are then passed to the main fortran file in the correct order, which will execute the RBM. To get more information, check Lst. 16 or invoke the help command: `python run.py --help`

4.11 Checking Fortran implementation agianst Python

To test the implementation of the RBM algorithm and the related functions, both Fortran and Python implementations are called using equal inputs defined by the user. The script used for testing is in Lst. 17, while the full output in Fortran and Python can be found respectively in Lst. 18 and 19.

In particular, we set a RBM with 3 visible and 3 hidden units with

$$A_i = \begin{bmatrix}-0.05 & 0.08 & 0.02\end{bmatrix}, \quad B_j = \begin{bmatrix}0.1 & -0.05 & -0.08\end{bmatrix}, \quad W_{ij} = \begin{bmatrix}-0.03 & 0.02 & 0.1\\ 0.07 & -0.12 & 0.03\\ -0.1 & -0.03 & 0.05\end{bmatrix}$$

as the biases and weights.
Using these values, `Metropolis` subroutine is checked producing 100 spin configurations and then counting the number of spins ± 1; since random numbers are involved in this procedure, we cannot find the same results in both Fortran and Python; however, the final results seems to be comparable:

- Fortran: 159 spins '+1', 141 spins '-1';
- Python: 153 spins '+1', 147 spins '-1'.

Then we introduced a dummy configuration matrix

$$\mathcal{S} = \begin{bmatrix}1 & 1 & 1\\ 1 & -1 & -1\\ 1 & -1 & -1\\ 1 & 1 & -1\\ 1 & -1 & -1\end{bmatrix}$$

and 1-dimensional Ising hamiltonian with $h = 0.2$.
The results from the functions `LocalEnergy` and `logPsiDiff` (between the two rows of $\mathcal{S}$) are:

Fortran

```
==================================================
Eloc
[DOUBLE COMPLEX array, dimension=  5]
    (-2.5865575144810693,0.0000000000000000)
   (-0.67569521762084805,0.0000000000000000)
   (-0.67569521762084805,0.0000000000000000)
   (-0.60241333036069289,0.0000000000000000)
   (-0.67569521762084805,0.0000000000000000)

==================================================

==================================================
log Psi diff
[DOUBLE COMPLEX]             (-0.20793134812583486,0.0000000000000000)
==================================================
```

Python

```
Local Energy = [-2.58655751 -0.67569522 -0.67569522 -0.60241333 -0.67569522]
Log(Psi2) - log(Psi1) = -0.20793134812583486
```

At last, to monitor the update of the RBM, we introduced also the covariance matrix S, the forces vector and the learning rate is set to 0.5. S has dimension 15 and it is filled with 1 except $S_{3,3} = -0.5$; F has $F_8 = -0.7$ and $F_{14} = 0.25$, while the other entries are 1. After the subroutine `RBM_update`, the new network parameters are:

Fortran

```
==================================================
Ai after update
[DOUBLE COMPLEX array, dimension=  3]
            (-0.55000000000000004,0.0000000000000000)
            (-0.41999999999999998,0.0000000000000000)
              (1.0200000000000000,0.0000000000000000)

==================================================



==================================================
Bj after update
[DOUBLE COMPLEX array, dimension=  3]
            (-0.40000000000000002,0.0000000000000000)
            (-0.55000000000000004,0.0000000000000000)
            (-0.57999999999999996,0.0000000000000000)

==================================================



==================================================
Wij after update
[DOUBLE COMPLEX matrix, dimension=  3, col=  3]
( -0.53000   ,    0.0000   )  (  0.37000   ,    0.0000   )  ( -0.40000   ,    0.0000   )
( -0.43000   ,    0.0000   )  ( -0.62000   ,    0.0000   )  ( -0.47000   ,    0.0000   )
( -0.60000   ,    0.0000   )  ( -0.15500   ,    0.0000   )  ( -0.45000   ,    0.0000   )
==================================================
```

Python

```
After update:
Ai = [-0.55 -0.42  1.02]
Bj = [-0.4  -0.55 -0.58]
Wij =
 [[-0.53   0.37  -0.4  ]
 [-0.43  -0.62  -0.47 ]
 [-0.6   -0.155 -0.45 ]]
```

5 Results

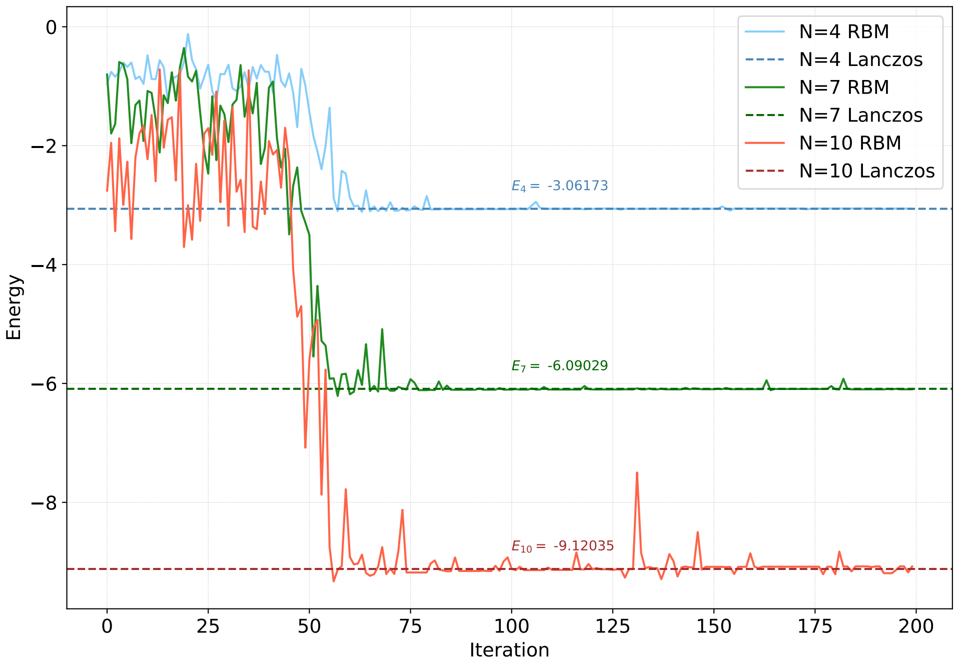
The algorithm developed is tested against the 1-D and 2-D Ising model in transverse field. The hamiltonian describing the system is

$$\mathcal{H} = -h\sum_i \sigma_i^x - \sum_{\langle i,j\rangle} \sigma_i^z \sigma_j^z \tag{14}$$

and in our implementation we do not apply any boundary condition. In addition, the number of sites is kept lower than 14, and the reasons behind this are explained in Sec. 6. Moreover, as already mentioned, the results are gathered using the Python implementation, since the Fortran one does not produce stable conclusions(see Fig. 5 in Appendix A).

1-dimensional Ising model Fig. 1 shows the behaviour of the RBM in evaluating the energy of the ground state for three different systems (size $= 4, 7, 10$ respectively); at each step, this correspond to the real part of the mean between the local energies obtained from the configurations sampled by the Metropolis (energy must be a real value, and numerically the imaginary part is of order 10^{-3}). For the first 50 iterations we can see the same trend for all the values of N: the energy has a lot of fluctuations due to the fact that the weights are initialized at random, and they are not close yet to the best representation of the system. From iteration 50 to 75, the RBM adapts its parameters following Eq. 3 to better illustrate the problem, and then the energy approaches the correct value. Then the weights remains more or less stable, and also the energy, besides some fluctuations, has the same behaviour. Oscillations are more important for larger systems (in our case for $N = 10$). The dashed lines show the true value of the ground state energy, which is obtained with the Lanczos algorithm.

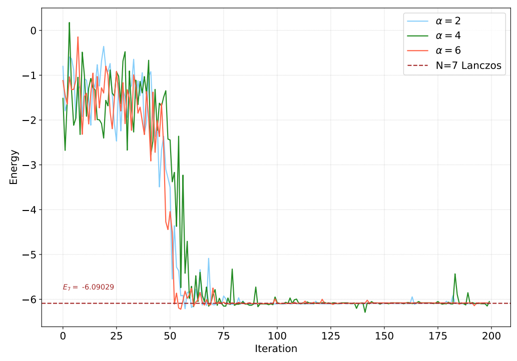
Figure 1: *RBM ground state energy prediction as function of iteration in the weights update procedure. The results are presented for systems with 4, 7, and 10 spins, and the number of hidden units of the network is twice the visible ones ($\alpha = 2$). The dashed lines represents the true energy, which is retrieved by the Lanczos procedure.*



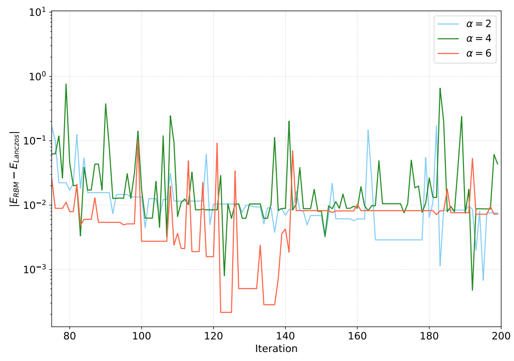
An import parameter of the RBM is the hidden unity density α, that is the ratio between the number of hidden and visible neurons. Fig. 2 shows the behavior of the RBM with $\alpha = 2, 4, 6$ when describing a system with 7 spins. In our case, as displayed by Fig. 2a, we can reach the convergence to the right energy independently on the value of α. However, even if the system is quite small, the difference with the Lanczos solution is overall smaller when α is bigger. This fact is presented in Fig. 2b, where these differences are enhanced using the log-scale. We can see that the orange line (corresponding to $\alpha = 6$) reaches (at iteration 120-140) an accuracy between 10^{-3} and 10^{-4}. It is to mention that α controls the complexity of the RBM, hence we expect that higher values of this parameter are needed when describing more complex systems, e.g. when the number of spins is larger.

2-dimensional Ising model The second application is the Ising model with transverse field in two dimensions. The hamiltonian of the system is the same of the 1-D case (Eq. 14) with the difference that now the lattice is a 2-D square one with dimension $N = L^2$ and a given particle has neighbours not only at its left/right but also above and below itself. We still do not set boundary conditions. In our tests, the system has 9 spins in a 3×3 lattice. As for the 1-D cases, Fig. 3 shows the behaviour of the RBM during the training.

Figure 2: *Performace of the RBM as function of the hidden unity density for a system with 7 spins described by a 1-dimensional Ising model in transverse field, without boundary conditions.*



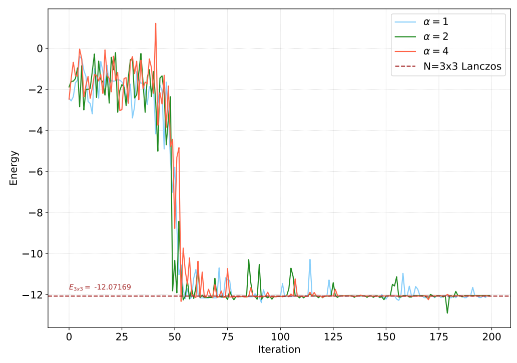
(A) *Predicted ground state energy for different vales of the hiddden unity density α.*



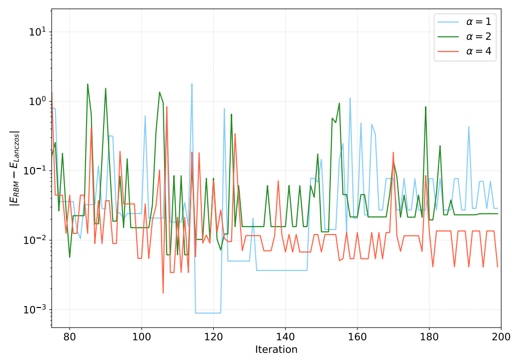
(B) *Differences between the ground state energy predicted by the RBM with and the true value given by the Lanczos algorithm; y-axis is in log-scale.*

Similar considerations as before can be made: the system oscillates a lot in the first part of the training ($\sim$ 50 iterations), until it reaches convergence. We can distinguish a different response looking at Fig. 3b. In this case we have that $\alpha = 1$ reach the best value of accuracy (around 10^{-3}) but using $\alpha = 4$ we get a more stable behaviour (e.g. better value of the mean of the accuracy).

Figure 3: *RBM ground state energy estimates for a system of 9 particles in a 3×3 square lattice, described by a transverse-field Ising model without boundary conditions.*



(A) *Ground state energy as function of the training iteration.*



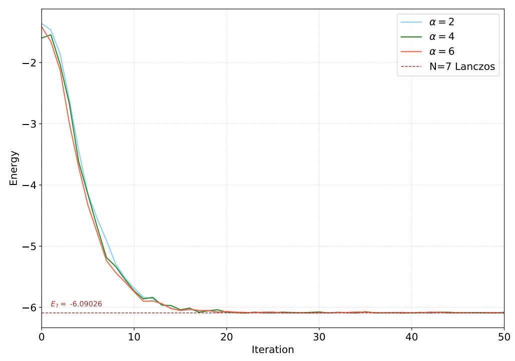
(B) *Differences between the ground state energy predicted by the RBM and the true value given by the Lanczos algorithm; y-axis is in log-scale.*

NetKet implementation The most accurate implementation of neural networks description of a many-body quantum system can be obtained by the package NetKet[3]. It is build on C++ and features a Python interface. This paragraph provides a sight to the results that can be reached by state of the art implementation of neural networks quantum states.

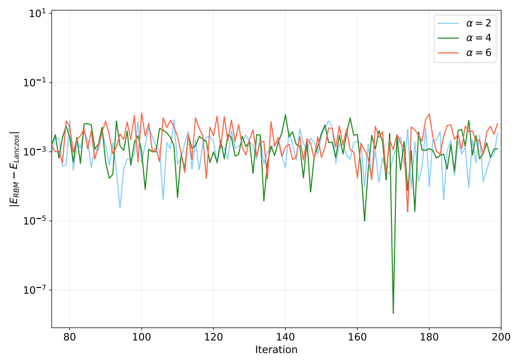
Fig. 4a shows the ground state energy for a $N = 7$ particles system using different values of α. If we compare this results against our algorithm (Fig. 2), we can see that NetKet is smoother and faster during the learning. This could be due to more complex regularization applied to the algorithm, that enhances numerical stability. Nevertheless, at equilibrium fluctuations are still present (even if this implementation is more stable), and the gain in accuracy with respect to our method is below a order of magnitude.

Netket is also optimized to work with systems with more than 14 particles; Fig. 4d shows the behavior of the RBM using bigger values of N.

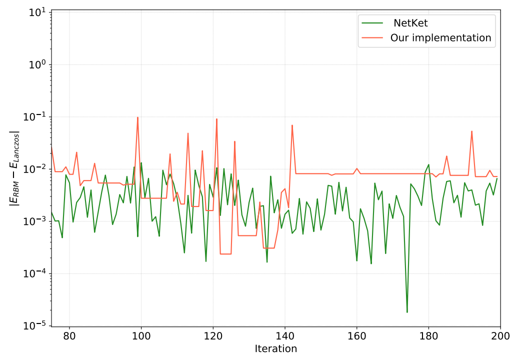
Figure 4: *NetKet RBM implementation results.*



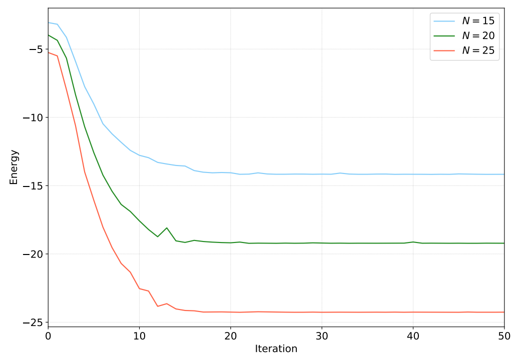
(A) *Description of a system with 7 particles for different values of α.*



(B) *Difference between the ground state energy predicted by RBM using NetKet and the true value by the Lanczos*



(C) *Difference between Lanczos solution and ground state energy obtained for a fixed $\alpha = 6$ using NetKet and our algorithm.*



(D) *Ground state energy for system with larger amount of spins.*

6 Conclusions and possible developments

Machine learning techniques can be applied in the field of quantum many-body systems to solve problems such as the estimate of the ground state energy. This work focuses on the reproduction of a RBM as described in [1].

As presented in Sec. 5, the implementation produces reasonable results in the description of the 1 and 2-dimensional Ising model in transverse field. The comparison against the solution provided by the Lanczos algorithm, shows an accuracy in the estimates the order of 10^{-2}. This fact highlights that there is room for improvements, as confirmed by the state of the art implementation given by NetKet.

In particular, regularizations to control the numerical stability should be investigated more in detail, since our results manifest the presence of noise that is absent in NetKet together with a slower convergence. Moreover, this point could be the main problem that afflicts the Fortran implementation and prevents it to reach the same results as the Python one (even if we are not able to precisely determine that, since the tested performed in Sec. 4.11 did not supply clues on where the error can reside).

Another upgrade relies on the description of larger systems. The scripts developed only works up to 14 particles (in our machines with 8 Gb), since there is the need to keep in memory the matrix representation of the hamiltonian, that requires $2^N \times 2^N$ values to be stored if we are working with N 1/2-spins. This is actually not mandatory, since most elements of H are equal to 0: hence sparse matrices can be a way to push forward this limit. An additional solution may even not require to store any elements of the matrix. In fact, we just need one element H_{ij} at a time (see Sec. 4.4), so instead of calculate the whole matrix, we could compute just the element we need at demanding.

An additional way to refine the results, could be to perform more run for each single system and mediate over the iteration; in this way we obtain a smoother behavior of the energy, since stochasticity is involved in the simulations.

References

[1] G. Carleo and M. Troyer, “Solving the quantum many-body problem with artificial neural networks,” *Science*, vol. 355, no. 6325, p. 602–606, Feb 2017. [Online]. Available: http://dx.doi.org/10.1126/science.aag2302

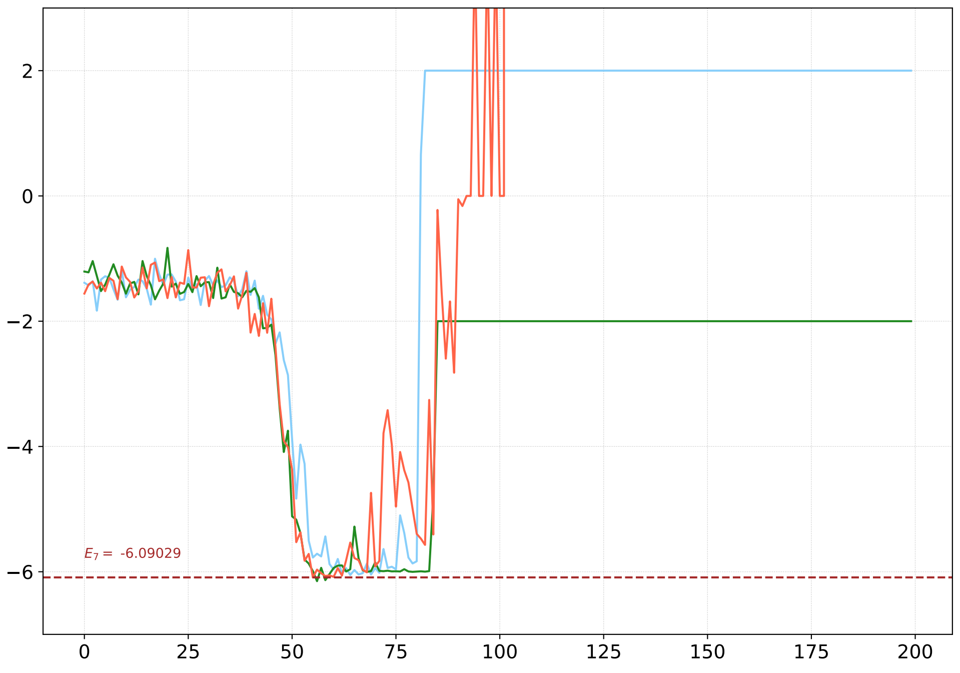
[2] S. Sorella, M. Casula, and D. Rocca, “Weak binding between two aromatic rings: Feeling the van der waals attraction by quantum monte carlo methods,” *The Journal of Chemical Physics*, vol. 127, no. 1, p. 014105, 2007. [Online]. Available: https://doi.org/10.1063/1.2746035

[3] G. Carleo, K. Choo, D. Hofmann, J. E. T. Smith, T. Westerhout, F. Alet, E. J. Davis, S. Efthymiou, I. Glasser, S.-H. Lin, M. Mauri, G. Mazzola, C. B. Mendl, E. van Nieuwenburg, O. O’Reilly, H. Théveniaut, G. Torlai, F. Vicentini, and A. Wietek, “Netket: A machine learning toolkit for many-body quantum systems,” *SoftwareX*, p. 100311, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711019300974

Appendix A Instability of Fortran implementation

Fortran script does not provide stable results, even if each piece of code was tested against the counter part in Python (Sec. 4.11). Fig. 5 clearly explains this problem: the results are obtained starting from the same conditions for a 1-dimensional system with 7 particles. It can be noticed that the covergens occurs at different values, and in some cases overflow leads to `NaN`.

Figure 5: *Ground state energy predictions through Fortran implementation of the RBM for a system with 7 spins in 1-dimension. All the results are produced starting from the same conditions; hence clearly there is a problem of numerical stability.*



Appendix B Code Listings

This appendix comprises the listings of code described in the report. In particular, the Fortran implementation is presented; the Python scripts follow the same logic and names, and they can be retrieved from the additional files delivered with the study.

Modules already developed during the classes or files that just gather the different functions to produce the results are not displayed here, but can be found in the support materials.

B.1 `simulation` module

This module contains the subroutine to generate random configurations according to the current state of the RBM, using the acceptance rule of Eq. 11.

Listing 1: *Metropolis simulation subroutine to generate spin configurations from according to the state of the RBM.*

```fortran
subroutine Metropolis(Ai, Bj, Wij, iter, burnin, autocorr, config, Opk)
!
! Metropolis simulation to generate many-body configurations based of the RBM state
!
!   Parameters
!   ----------
```

```
    !   [IN] Ai : DOUBLE COMPLEX, DIMENSION(N)
    !       Biases of the visible units
    !   [IN] Bj : DOUBLE COMPLEX, DIMENSION(M)
    !       Biases of the hidden units
    !   [IN] Wij : DOUBLE COMPLEX, DIMENSION(N,M)
    !       Weights between visible and hidden units
    !   [IN] iter : INTEGER
    !       Number of steps of the simulation
    !   [IN] burnin : INTEGER
    !       Number of steps to discard before saving the results.
    !       Must be burnin < iter.
    !   [IN] autocorr : INTEGER
    !       Number of steps between two consecutive records of the results
    !   [OUT] config : INTEGER, DIMENSION((iter-burnin)/autocorr, N))
    !       Matrix with the spin configurations generated by the Metropolis.
    !       Each row represents a different realization.
    !   [OUT] Opk : DOUBLE COMPLEX, DIMENSION((iter-burnin)/autocorr, N+M+N*M))
    !       Matrix with the derivatives of the RBM wave function
    !       wrt to the network parameters.
    !       Each row represents a different realization.
    !
        implicit none
        ! parameters
        DOUBLE COMPLEX, INTENT(IN) :: Ai(:), Bj(:), Wij(:,:)
        INTEGER,        INTENT(IN) :: iter, burnin, autocorr
        INTEGER, ALLOCATABLE, INTENT(OUT) :: config(:,:)
        DOUBLE COMPLEX, ALLOCATABLE, INTENT(OUT) :: Opk(:,:)

        ! local variables
        INTEGER :: N, M, t, idx, idy, idz, idk
        INTEGER, ALLOCATABLE :: Si(:), Si_new(:)
        DOUBLE PRECISION :: u, r
        DOUBLE COMPLEX :: log_r
        DOUBLE COMPLEX, ALLOCATABLE :: Tj(:)

        ! get dimension of visible and hidden layer
        N = SIZE(Ai)
        M = SIZE(Bj)

        ! checks on RBM input dimensions
        if ( SIZE(Wij, 1) /= N .OR. SIZE(Wij, 2) /= M ) then
            print*, "[ABORT] Wrong shapes in RBM parameters"
            print*, "Visible units dimension = ", N
            print*, "Hidden  units dimension = ", M
            print*, "Weights dimensions      = ", SIZE(Wij,1), " ,", SIZE(Wij,2)
            print*, "(expected = ", N, " ,", M, ")"
            CALL ABORT()
        end if

        ! generate random initial configuration
        Si = Random_Configuration(N)
        !ALLOCATE(Si(N))
        !Si = 1
        ALLOCATE(Si_new(N))
        ! effective angles
        ALLOCATE(Tj(M))

        ! checks on iter, burnin and autocorr
        if ( iter <= 0 ) then
            print*, "[ABORT] 'Iter' must be > 0"
            CALL ABORT()
        end if

```

```fortran
        if ( burnin < 0 .OR. burnin >= iter ) then
            print*, "[ABORT] 'burnin' must be >= 0 and < 'iter'"
            CALL ABORT()
        end if

        if ( autocorr <= 0 .OR. autocorr > iter-burnin ) then
            print*, "[ABORT] 'autocorr' must be > 0 and <= 'iter-burnin'"
        end if

        ! configurations to store
        if ( ALLOCATED(config) ) DEALLOCATE(config)
        ALLOCATE(config((iter-burnin)/autocorr, N))
        ! Opk
        if ( ALLOCATED(Opk) ) DEALLOCATE(Opk)
        ALLOCATE(Opk((iter-burnin)/autocorr, N+M+N*M))

        idy = 1

        do t = 1, iter
            CALL debugging(.FALSE., var=Si, message="+++ Configuration")
            ! choose a random spin to flip
            idx = Random_Integer(1, N+1)
            Si_new = Si
            Si_new(idx) = -Si_new(idx)
            ! calculate the log of ratio of the wave functions
            log_r = logPsiDiff(Si_new, Si, Ai, Bj, Wij)
            ! metropolis ratio
            r = ABS(EXP(log_r))**2
            ! accept the new config with probability 'r'
            CALL RANDOM_NUMBER(u)
            if ( u < r ) then
                ! accept new configuration
                Si = Si_new
            end if

            if ((t > burnin) .AND. (MOD(t, autocorr) == 0)) then
                ! calculate effective angles
                Tj = Bj + MATMUL(TRANSPOSE(Wij), Si)
                ! save the spin configuration (one per row)
                config(idy, :) = Si
                ! save the derivatives
                !   wrt Ai
                Opk(idy, 1:N) = Si
                !   wrt Bj
                Opk(idy, N+1:N+M) = TANH(Tj)
                !   wrt Wij
                idk = 1
                do idx = 1, N
                    do idz = 1, M
                        Opk(idy, N+M+idk) = Si(idx)*TANH(Tj(idz))
                        idk = idk+1
                    end do
                end do
                idy = idy + 1
            end if
        end do
        CALL debugging(.FALSE., var=Opk, message="Opk")

        ! deallocate arrays
        DEALLOCATE(Si)
        DEALLOCATE(Tj)

        RETURN
```

```fortran

end subroutine Metropolis
```

B.2 `rbm` module

All the subroutine and functions that are needed to create and update the state of the RBM are included here.

Listing 2: *Random initialization of the parameters of the RBM (biases and weights).*

```fortran
subroutine RBM_init(N, M, Ai, Bj, Wij, mean, std)
!
! Randomly initializes the weights of the RBM,
! with normal distribution
!
!   Parameters
!   ----------
!   [IN] N : INTEGER
!       number of visible units
!   [IN] M : INTEGER
!       number of hidden units
!   [OUT] Ai : DOUBLE COMPLEX, DIMENSION(N)
!       Biases of the visible units; the subroutine allocates the space.
!   [OUT] Bj : DOUBLE COMPLEX, DIMENSION(M)
!       Biases of the hidden units; the subroutine allocates the space.
!   [OUT] Wij : DOUBLE COMPLEX, DIMENSION(N, M)
!       Weights between visible and hidden units; the subroutine allocates the space.
!   [IN] mean : DOUBLE PRECISION
!       Mean of the normal distribution
!   [IN] std : DOUBLE PRECISION
!       Standard deviation of the normal distribution
!
    implicit none
    ! parameters
    INTEGER, INTENT(IN) :: N, M
    DOUBLE PRECISION, INTENT(IN) :: mean, std
    DOUBLE COMPLEX, ALLOCATABLE, INTENT(OUT) :: Ai(:), Bj(:), Wij(:,:)
    ! local variables
    DOUBLE COMPLEX, ALLOCATABLE :: r(:)

    if ( ALLOCATED(Ai)  ) DEALLOCATE(Ai)
    if ( ALLOCATED(Bj)  ) DEALLOCATE(Bj)
    if ( ALLOCATED(Wij) ) DEALLOCATE(Wij)

    ALLOCATE( Ai(N) )
    ALLOCATE( Bj(M) )
    ALLOCATE( Wij(N,M))

    ! check on std
    if ( std < 0.d0 ) then
        print*, "[WARNING] 'std' is negative, using the absolute value"
    end if

    Ai = CMPLX(Random_Normal(N, mean, ABS(std)), &
               Random_Normal(N, mean, ABS(std)), KIND=8)

    Bj = CMPLX(Random_Normal(M, mean, ABS(std)), &
               Random_Normal(M, mean, ABS(std)), KIND=8)

    ALLOCATE(r(N*M))

    r = CMPLX(Random_Normal(N*M, mean, ABS(std)), &
              Random_Normal(N*M, mean, ABS(std)), KIND=8)
    Wij = RESHAPE( r, (/N,M/) )

    DEALLOCATE(r)

    RETURN

end subroutine RBM_init
```

Listing 3: *RBM weights update subroutine (Eq. 3).*

```fortran
    subroutine RBM_update(Ai, Bj, Wij, S_kk, Fk, g)
    !
    ! Update the weights of the RBM.
    !
    !   Parameters
    !   ----------
    !   [INOUT] Ai : DOUBLE COMPLEX, DIMENSION(N)
    !       Biases of the visible units
    !   [INOUT] Bj : DOUBLE COMPLEX, DIMENSION(M)
    !       Biases of the hidden units
    !   [INOUT] Wij : DOUBLE COMPLEX, DIMENSION(N,M)
    !       Weights between visible and hidden units
    !   [INOUT] S_kk : DOUBLE COMPLEX, DIMENSION(k,k)
    !       Regularized covariance matrix.
    !       k = N + M + N*M (# of network weights)
    !   [IN] Fk : DOUBLE COMPLEX, DIMENSION(k)
    !       Forces array.
    !       k = N + M + N*M (# of network weights)
    !   [IN] g : DOUBLE PRECISION
    !       Learning rate.
    !


        implicit none
        ! parameters
        DOUBLE COMPLEX, INTENT(INOUT) :: Ai(:), Bj(:), Wij(:,:), S_kk(:,:)
        DOUBLE COMPLEX, INTENT(IN)    :: Fk(:)
        DOUBLE PRECISION, INTENT(IN)  :: g

        ! local variables
        INTEGER :: N, M, k
        DOUBLE COMPLEX, ALLOCATABLE :: SF(:)

        N = SIZE(Ai) ! # visible units
        M = SIZE(Bj) ! # hidden units

        ! checks on RBM input dimensions
        if ( SIZE(Wij, 1) /= SIZE(Ai) .OR. SIZE(Wij, 2) /= SIZE(Bj) ) then
            print*, "[ABORT] Wrong shapes in RBM parameters"
            print*, "Visible units dimension = ", SIZE(Ai)
            print*, "Hidden  units dimension = ", SIZE(Bj)
            print*, "Weights dimensions      = ", SIZE(Wij,1), ", ", SIZE(Wij,2)
            print*, "(expected = ", SIZE(Ai), ", ", SIZE(Bj), ")"
            CALL ABORT()
        end if

        k = N+M+N*M

        ! checks on covariance matrix S_kk
        if ( SIZE(S_kk, 1) /= k .OR. SIZE(S_kk, 2) /= k ) then
            print*, "[ABORT] Wrong shape in covariance matrix"
            print*, "Found    = ", SIZE(S_kk, 1), ", ", SIZE(S_kk, 2)
            print*, "Expected = ", k, ", ", k
            CALL ABORT()
        end if

        ALLOCATE(SF(k))

        CALL Inverse(S_kk)
        S_kk = MERGE(S_kk, CMPLX(0.d0, 0.d0, KIND=8), ABS(S_kk)>1d-09)

        CALL debugging(.FALSE., var=SIZE(S_kk, dim=2), message="S_kk")
        CALL debugging(.FALSE., var=SIZE(Fk), message="Fk")

        SF = g*MATMUL(S_kk, Fk)
        SF = MERGE(SF, CMPLX(0.d0, 0.d0, KIND=8), ABS(SF)>1d-09)

        CALL debugging(.FALSE., var=SUM(SF)/SIZE(SF), message="Weigths update")
        ! update weights
        Ai = Ai - SF(:N)
        Bj = Bj - SF(N+1:M)
        Wij = Wij - TRANSPOSE(RESHAPE(SF(N+M+1:), (/M,N/)))

        DEALLOCATE(SF)

        RETURN

    end subroutine RBM_update
```

Listing 4: *Covariance matrix (Eq.4).*

```fortran
    function Skk(Opk, iter) result(S_kk)
    !
    ! Returns the covariance matrix given in (A4).
    ! Explicit regularization is applied as described in Appendix A.
    !
    !   Parameters
    !   ----------
    !   Opk : DOUBLE COMPLEX, DIMENSION(p,k)
    !       Matrix with the derivatives of the RBM wave function
    !       wrt to the network parameters.
    !       Each row represents a different realization.
    !       p = # of different realizations
    !       k = N + M + N*M (# of network weights)
    !   iter : INTEGER
    !       Weights update iteration. It is needed to apply the regularization.
    !
    !   Return
    !   ------
    !   S_kk : DOUBLE COMPLEX, DIMENSION(k,k)
    !       Regularized covariance matrix.
    !
        implicit none
        ! parameters
        DOUBLE COMPLEX, INTENT(IN) :: Opk(:,:)
        INTEGER, INTENT(IN) :: iter
        ! return
        DOUBLE COMPLEX, ALLOCATABLE :: S_kk(:,:)

        ! local variables
        INTEGER :: k, p, ii, jj, mm
        DOUBLE PRECISION :: lp
        DOUBLE COMPLEX, ALLOCATABLE :: Ok_mean(:)
        ! constant
        DOUBLE PRECISION, PARAMETER :: l0   = 100.d0, &
                                       b    = 0.9d0,  &
                                       lmin = 1d-04

        ! get the dimension of Opk
        p = SIZE(Opk, DIM=1) ! number of realizations
        k = SIZE(Opk, DIM=2) ! N+M+N*M

        if ( ALLOCATED(S_kk) ) DEALLOCATE(S_kk)
        ALLOCATE( S_kk(k,k) )

        ! < Ok* Ok' >
        S_kk = CMPLX(0.d0, 0.d0, KIND=8)
        ! loop over rows of Opk
        do ii = 1, p
            ! loop over elements of a row
            do jj = 1, k
                ! loop over elements of a row
                do mm = 1, k
                    S_kk(jj,mm) = S_kk(jj,mm) + CONJG(Opk(ii,jj))*Opk(ii,mm)
                end do
            end do
        end do
        S_kk = S_kk / p

        ! < Ok >
        ALLOCATE(Ok_mean(k))
        do ii = 1, k
            Ok_mean(ii) = SUM(Opk(:, ii)) / p
        end do

        ! < Ok* Ok' > - < Ok* > < Ok' >
        lp = MAX(l0*b**iter, lmin)
        do ii = 1, k
            do jj = 1, k
                S_kk(ii,jj) = S_kk(ii,jj) - CONJG(Ok_mean(ii))*Ok_mean(jj)
                ! regularization on diagonal terms
                if ( ii == jj ) then
                    S_kk(ii,ii) = S_kk(ii,ii) + lp
                end if
            end do
        end do

        DEALLOCATE(Ok_mean)

        S_kk = MERGE(S_kk, CMPLX(0.d0, 0.d0, KIND=8), ABS(S_kk)>1d-09)

        RETURN

    end function Skk
```

Listing 5: *Forces vector calculation (Eq. 5).*

```

        SF = g*MATMUL(S_kk, Fk)
        SF = MERGE(SF, CMPLX(0.d0, 0.d0, KIND=8), ABS(SF)>1d-09)

        CALL debugging(.FALSE., var=SUM(SF)/SIZE(SF), message="Weigths update")
        ! update weights
        Ai = Ai - SF(:N)
        Bj = Bj - SF(N+1:M)
        Wij = Wij - TRANSPOSE(RESHAPE(SF(N+M+1:), (/M,N/)))
```

```fortran

        DEALLOCATE(SF)

        RETURN

    end subroutine RBM_update

    function Skk(Opk, iter) result(S_kk)
    !
    ! Returns the covariance matrix given in (A4).
    ! Explicit regularization is applied as described in Appendix A.
    !
    !   Parameters
    !   ----------
    !   Opk : DOUBLE COMPLEX, DIMENSION(p,k)
    !       Matrix with the derivatives of the RBM wave function
    !       wrt to the network parameters.
    !       Each row represents a different realization.
    !       p = # of different realizations
    !       k = N + M + N*M (# of network weights)
    !   iter : INTEGER
    !       Weights update iteration. It is needed to apply the regularization.
    !
    !   Return
    !   ------
    !   S_kk : DOUBLE COMPLEX, DIMENSION(k,k)
    !       Regularized covariance matrix.
    !
        implicit none
        ! parameters
        DOUBLE COMPLEX, INTENT(IN) :: Opk(:,:)
        INTEGER, INTENT(IN) :: iter
        ! return
        DOUBLE COMPLEX, ALLOCATABLE :: S_kk(:,:)

        ! local variables
        INTEGER :: k, p, ii, jj, mm
        DOUBLE PRECISION :: lp
        DOUBLE COMPLEX, ALLOCATABLE :: Ok_mean(:)
        ! constant
        DOUBLE PRECISION, PARAMETER :: l0   = 100.d0, &
                                       b    = 0.9d0,  &
                                       lmin = 1d-04

        ! get the dimension of Opk
        p = SIZE(Opk, DIM=1) ! number of realizations
        k = SIZE(Opk, DIM=2) ! N+M+N*M

        if ( ALLOCATED(S_kk) ) DEALLOCATE(S_kk)
        ALLOCATE( S_kk(k,k) )

        ! < Ok* Ok' >
        S_kk = CMPLX(0.d0, 0.d0, KIND=8)
        ! loop over rows of Opk
        do ii = 1, p
            ! loop over elements of a row
            do jj = 1, k
                ! loop over elements of a row
                do mm = 1, k
                    S_kk(jj,mm) = S_kk(jj,mm) + CONJG(Opk(ii,jj))*Opk(ii,mm)
                end do
            end do
        end do
        S_kk = S_kk / p

        ! < Ok >
        ALLOCATE(Ok_mean(k))
        do ii = 1, k
            Ok_mean(ii) = SUM(Opk(:, ii)) / p
        end do

        ! < Ok* Ok' > - < Ok* > < Ok' >
        lp = MAX(l0*b**iter, lmin)
        do ii = 1, k
            do jj = 1, k
                S_kk(ii,jj) = S_kk(ii,jj) - CONJG(Ok_mean(ii))*Ok_mean(jj)
                ! regularization on diagonal terms
                if ( ii == jj ) then
                    S_kk(ii,ii) = S_kk(ii,ii) + lp
                end if
            end do
        end do

        DEALLOCATE(Ok_mean)

        S_kk = MERGE(S_kk, CMPLX(0.d0, 0.d0, KIND=8), ABS(S_kk)>1d-09)

        RETURN

    end function Skk

    function Forces(Opk, Eloc) result(Fk)
    !
    ! Returns the forces as in (A5).
    !
    !   Parameters
    !   ----------
    !   Opk : DOUBLE COMPLEX, DIMENSION(p,k)
    !       Matrix with the derivatives of the RBM wave function
    !       wrt to the network parameters.
    !       Each row represents a different realization.
    !       p = # of different realizations
    !       k = N + M + N*M (# of network weights)
    !   Eloc : DOUBLE COMPLEX, DIMENSION(p)
    !       Array with the local energies of the MC spin configurations
    !
    !   Return
    !   ------
    !   Fk : DOUBLE COMPLEX, DIMENSION(k)
    !       Forces array
    !
        implicit none
        ! parameters
        DOUBLE COMPLEX, INTENT(IN) :: Opk(:,:), Eloc(:)
        ! return
        DOUBLE COMPLEX, ALLOCATABLE :: Fk(:)

        ! local variables
        INTEGER :: p, k, ii
        DOUBLE COMPLEX, ALLOCATABLE :: El_Okstar(:)

        p = SIZE(Opk, 1)
        k = SIZE(Opk, 2)

        ! checks on inputs dimensions
        if ( SIZE(Eloc) /= p ) then
```

```fortran
            print*, "[ABORT] Wrong dimensions in input"
            print*, "'Opk'  dimension = ", p, ", ", k
            print*, "'Eloc' dimension = ", SIZE(Eloc)
            print*, "(expected = ", p, ")"
            CALL ABORT()
        end if

        CALL debugging(.FALSE., var=k, message="Forces - k")

        if ( ALLOCATED(Fk) ) DEALLOCATE(Fk)
        ALLOCATE(Fk(k))
        ALLOCATE(El_Okstar(k))

        ! < Eloc Ok* >
        do ii = 1, k
            El_Okstar(ii) = DOT_PRODUCT(Eloc, CONJG(Opk(:,ii)))
        end do
        El_Okstar = El_Okstar / p

        CALL debugging(.FALSE., var=El_Okstar, message="< Eloc Ok* >")
        CALL debugging(.FALSE., var=SUM(Eloc)/p * SUM(CONJG(Opk), DIM=1)/p, message="<
    Eloc > < Ok* >")

        ! < Eloc Ok* > - < Eloc > < Ok* >
        Fk = El_Okstar - SUM(Eloc)/p * SUM(CONJG(Opk), DIM=1)/p

        CALL debugging(.FALSE., var=SIZE(Fk), message="Forces - Fk size - end")

        DEALLOCATE(El_Okstar)

        Fk = MERGE(Fk, CMPLX(0.d0, 0.d0, KIND=8), ABS(Fk)>1d-09)

        RETURN

    end function Forces
```

Listing 6: *Local energy derivation from a set of spin configurations (Eq.10).*

```fortran
    function LocalEnergy(config, H, Ai, Bj, Wij) result(Eloc)
    !
    ! Returns the array with local energy for each spin
    ! configuration sampled from the Metropolis.
    !
    !   Parameters
    !   ----------
    !   config : INTEGER, DIMENSION(p,N)
    !       Matrix in which each row corresponds to a configuration
    !       sapled by Metropolis.
    !       p = # of configurations
    !       N = # number of spins (eg. visible units)
    !   H : DOUBLE COMPLEX, DIMENSION(2**N, 2**N)
    !       Matrix describing the hamiltonian of the system
    !   Ai : DOUBLE COMPLEX, DIMENSION(N)
    !       Biases of visible units
    !   Bj : DOUBLE COMPLEX, DIMENSION(M)
    !       Biases of hidden units
    !   Wij : DOUBLE COMPLEX, DIMENSION(N,M)
    !       Weights between visible and hidden units
    !
    !   Return
    !   ------
    !   Eloc : DOUBLE COMPLEX, DIMENSION(p)
    !       Array with local energy values for each configuration
    !
        implicit none
        ! parameters
        INTEGER, INTENT(IN) :: config(:,:)
        DOUBLE COMPLEX, INTENT(IN) :: H(:,:), Ai(:), Bj(:), Wij(:,:)
        ! return
        DOUBLE COMPLEX, ALLOCATABLE :: Eloc(:)

        ! local variables
        INTEGER :: p, N, ii, jj, idx
        INTEGER, ALLOCATABLE :: S1(:), S2(:)
        DOUBLE COMPLEX :: Hij

        p = SIZE(config, 1) ! # of configurations
        N = SIZE(config, 2) ! # of spins

        ! checks on hamiltonian dimension
        if ( SIZE(H,1) /= 2**N .OR. SIZE(H,2) /= 2**N ) then
            print*, "[ABORT] Wrong shape in hamiltonian"
            print*, "Found    = ", SIZE(H,1), ", ", SIZE(H,2)
            print*, "Expected = ", 2**N, ", ", 2**N
            CALL ABORT()
        end if

        ! checks on visible units
        if ( SIZE(Ai) /= N ) then
            print*, "[ABORT] Wrong dimension in visible units"
            print*, "Found    = ", SIZE(Ai)
            print*, "Expected = ", N
            CALL ABORT()
        end if

        ! checks on RBM input dimensions
        if ( SIZE(Wij, 1) /= SIZE(Ai) .OR. SIZE(Wij, 2) /= SIZE(Bj) ) then
            print*, "[ABORT] Wrong shapes in RBM parameters"
            print*, "Visible units dimension = ", SIZE(Ai)
            print*, "Hidden  units dimension = ", SIZE(Bj)
            print*, "Weights dimensions      = ", SIZE(Wij,1), ", ", SIZE(Wij,2)
            print*, "(expected = ", SIZE(Ai), ", ", SIZE(Bj), ")"
            CALL ABORT()
        end if

        if ( ALLOCATED(Eloc) ) DEALLOCATE(Eloc)
        ALLOCATE(Eloc(p))
        Eloc = CMPLX(0.d0, 0.d0, KIND=8)
        CALL debugging(.FALSE., var=SUM(Eloc), message="+++ Eloc sum")

        ALLOCATE(S1(N))
        ALLOCATE(S2(N))

        CALL debugging(.FALSE., message="+++ before loop")
        ! loop over MC configurations
        do ii = 1, p
            CALL debugging(.FALSE., var=ii, message="+++Outer loop")
            ! get S
            S1 = config(ii,:)
            ! get integer representation of spin configuration
            idx = idx_from_config(S1)
            CALL debugging(.FALSE., var=idx, message="integer S1")
            ! loop over all S'
            do jj = 1, 2**N
                ! need to pass idx+1 since Fortran starts from 1,
                ! while the integer representation from 0
                Hij = H(idx+1,jj) ! < S | H | S' >
                CALL debugging(.FALSE., var=jj, message="+++Inner loop")
                if ( ABS(Hij) > 1d-09 ) then ! if ( Hij /= 0 ) then
                    ! need to pass jj-1 since Fortran starts from 1,
                    ! while the integer representation from 0
                    S2 = config_from_idx(jj-1,N)
                    Eloc(ii) = Eloc(ii) + Hij*EXP(logPsiDiff(S2, S1, Ai, Bj, Wij))
                end if
            end do
            CALL debugging(.FALSE., var=Eloc(ii), message="+++ Eloc sum")
        end do
        CALL debugging(.FALSE., message="+++ after loop")

        DEALLOCATE(S1)
        DEALLOCATE(S2)

        RETURN

    end function LocalEnergy
```

B.3 random module

Small set of functions to provide random number in different contexts from the uniform distribution.

Listing 7: *Random number from normal distribution.*

```
    function Random_Normal(N, mean, std) result(Normal)
    !
    ! Retruns an array of dimension N with normal distributed numbers
    !
    !   Parameters
    !   ----------
    !   N : INTEGER
    !       Dimension of the returned array
    !   mean : DOUBLE PRECISION
    !       Mean of the distribution
    !   std : DOUBLE COMPLEX
    !       Standard deviation of the distribution
    !
    !   Return
    !   ------
    !   Normal : DOUBLE PRECISION, DIMENSION(N)
    !       Array with N normal random number
    !
        implicit none
        ! parameters
        INTEGER, INTENT(IN) :: N
        DOUBLE PRECISION, INTENT(IN) :: mean, std
        ! returns
        DOUBLE PRECISION, ALLOCATABLE :: Normal(:)

        ! local variables
        DOUBLE PRECISION, ALLOCATABLE :: u1(:), u2(:)

        if ( std < 0.d0 ) then
            print*, "[WARNING] 'std' is negative, using the absolute value"
        end if

        if ( ALLOCATED(Normal) ) DEALLOCATE(Normal)
        ALLOCATE(Normal(N))
```

```fortran

        ALLOCATE(u1(N))
        ALLOCATE(u2(N))

        CALL RANDOM_NUMBER(u1)
        CALL RANDOM_NUMBER(u2)

        Normal = SQRT(-2.d0*DLOG(u1))*DCOS(2.d0*pi*u2)
        Normal = mean + ABS(std)*Normal

        DEALLOCATE(u1)
        DEALLOCATE(u2)

        RETURN

    end function Random_Normal
```

Listing 8: *Random integer number.*

```fortran
    function Random_Integer(low, high) result(randint)
        !
        ! Returns a random integer in the range [low, high)
        !
        !   Parameters
        !   ----------
        !   low : INTEGER
        !       Lowest (signed) integer to be drawn from the distribution
        !   high : INTEGER
        !       One above the largest (signed) integer to be drawn from the distribution
        !
        !   Returns
        !   -------
        !   randint : INTEGER
        !       Random integer in [low, high)
        !
            implicit none
            ! parameters
            INTEGER, INTENT(IN) :: low, high
            ! returns
            INTEGER :: randint

            ! local variables
            REAL :: r

            ! check if low < high
            if ( low > high ) then
                print*, "[ABORT] 'low' must be <= 'high'"
                CALL ABORT()
            end if

            CALL RANDOM_NUMBER(r)
            randint = low + FLOOR((high-low)*r)

            RETURN

        end function Random_Integer
```

Listing 9: *Random spin configuration.*

```fortran
    function Random_Configuration(N) result(config)
        !
        ! Generates a random configuration of spins +/- 1
        !
        !   Parameters
        !   ----------
        !   N : INTEGER
        !       Number of spins in the configuration
        !
        !   Returns
        !   -------
        !   config : INTEGER, DIMENSION(N)
        !       Array where to store the configuration.
        !

            implicit none
            ! parameters
            INTEGER, INTENT(IN) :: N
            ! returns
            INTEGER, ALLOCATABLE :: config(:)

            ! local variables
            REAL, ALLOCATABLE :: r(:)

            if ( ALLOCATED(config) ) DEALLOCATE(config)
            ALLOCATE(config(N))

            ALLOCATE(r(N))
            CALL RANDOM_NUMBER(r)

            config = -1 +2*FLOOR(r*2.)

            RETURN

        end function Random_Configuration
```

B.4 others module

Routines needed to perform auxiliary tasks.

Listing 10: *Integer representation of a spin configuration.*

```fortran
    function idx_from_config(config) result(idx)
    !
    ! Returns the integer representation of a spin configuration.
    ! Each configuration can be represented in the 2**N space as
    ! a vector of 0, and one 1 in position 'idx'.
    !
    !   Parameters
    !   ----------
    !   config : INTEGER, DIMENSION(N)
    !       Spin configuration (array with +-1)
    !
    !   Returns
    !   -------
    !   idx : INTEGER
    !       Integer representation of config in the 2**N vector.
    !       It corresponds to the index of the element 1 in that array.
    !
        implicit none
        ! parameters
        INTEGER, INTENT(IN) :: config(:)
        ! return
        INTEGER :: idx

        ! local variables
        INTEGER :: ii, N

        N = SIZE(config)
        idx = 0
        do ii = 1, N
            if ( config(ii) == 1 ) idx = idx + 2**(N-ii)
        end do

        RETURN

    end function idx_from_config
```

Listing 11: *Spin configuration from its integer representation.*

```fortran
    function config_from_idx(idx, N) result(config)
    !
    ! Returns the spin configuration given the integer representation.
    ! It is the inverse of 'idx_from_config'
    !
    !   Parameters
    !   ----------
    !   idx : INTEGER
    !       Integer representation of config in the 2**N vector.
    !       It corresponds to the index of the element 1 in that array.
    !   N : INTEGER
    !       Total number of spis in the configuration.
    !
    !   Returns
    !   -------
    !   config : INTEGER, DIMENSION(N)
    !       Spin configuration (array with +-1)
    !
        implicit none
        ! parameters
        INTEGER, INTENT(IN) :: idx, N
        ! return
        INTEGER, ALLOCATABLE :: config(:)

        ! local variables
        INTEGER :: ii
        INTEGER, ALLOCATABLE :: bit(:)

        ! checks on inputs
        if ( N < 1 ) then
            print*, "[ABORT] 'N' must be > 1"
            print*, "Found = ", N
        end if

        if ( idx < 0 .OR. idx > 2**N-1 ) then
            print*, "[ABORT] 'idx' must be >= 0 and < 2**N"
            print*, "Found       = ", idx
            print*, "Max allowed = ", 2**N-1
            CALL ABORT()
        end if

        if ( ALLOCATED(config) ) DEALLOCATE(config)
        ALLOCATE(config(N))
        config = -1

        ! get the base-2 value of idx
        bit = int2bit(idx)
        do ii = 1, size(bit)
            if ( bit(ii) == 1 ) config(ii) = 1
        end do
        ! need to reverse config vector
        config = config(N:1:-1)

        RETURN
```

Listing 12: *Integer conversion to its binary representation.*

```fortran
function int2bit(Number) result(bit)
!
! Returns the binary representation of the integer as an array.
! First element of the array is the LSB.
!
!   Parameters
!   ----------
!   Number : INTEGER
!       Integer to convert. Must be >= 0.
!
!   Return
!   ------
!   bit : INTEGER, DIMENSION(INT(log2(N) + 1))
!       Bit representation of N. First element is the LSB.
!
    implicit none
    ! parameters
    INTEGER, INTENT(IN) :: Number
    ! return
    INTEGER, ALLOCATABLE :: bit(:)

    ! local variables
    INTEGER :: N, nbits, ii

    ! checks on N
    if ( N < 0 ) then
        print*, "[ABORT] 'N' must be >= 0"
        print*, "Found = ", N
        CALL ABORT()
    end if

    N = Number

    if ( N == 0 ) then
        nbits = 1
    else
        nbits = INT(LOG(REAL(N)) / LOG(2.) + 1)
    end if

    ALLOCATE(bit(nbits))

    do ii = 1, nbits
        bit(ii) = MOD(N, 2)
        N = N / 2
    end do

    RETURN

end function int2bit
```

Listing 13: *Logarithm of the ration between two wave wave functions $\Psi_M(S_2)$ and $\Psi_M(S_1)$.*

```fortran
    function logPsiDiff(Si2, Si1, Ai, Bj, Wij) result(logPsi)
    !
    ! Returns the log of the ratio between the
    ! Network Quantum States described by spin configurations
    ! Si2 and Si1:
    !
    ! log( Psi(Si2) / Psi(Si1) ) = log( Psi(Si2) ) - log( Psi(Si1) )
    !
    !   Parameters
    !   ----------
    !   Si2 : INTEGER, DIMENSION(N)
    !       Final spin configuration (+- 1 array)
    !   Si1 : INTEGER, DIMENSION(N)
    !       Initial spin configuration (+- 1 array)
    !   Ai : DOUBLE COMPLEX, DIMENSION(N)
    !       Biases of the visible units
    !   Bj : DOUBLE COMPLEX, DIMENSION(M)
    !       Biases of the hidden units
    !   Wij : DOUBLE COMPLEX, DIMENSION(N,M)
    !       Weights between visible and hidden units
    !
    !   Return
    !   ------
    !   logPsi : DOUBLE COMPLEX
    !       log( Psi(Si2) ) - log( Psi(Si1) )
    !
        implicit none
        ! parameters
        INTEGER, INTENT(IN) :: Si2(:), Si1(:)
        DOUBLE COMPLEX, INTENT(IN) :: Ai(:), Bj(:), Wij(:,:)
        ! return
        DOUBLE COMPLEX :: logPsi

        ! local variables
        INTEGER :: M
        DOUBLE COMPLEX, ALLOCATABLE :: Tj2(:), Tj1(:)

        ! checks on Si
        if ( SIZE(Si2) /= SIZE(Si1) ) then
            print*, "[ABORT] Wrong dimensions between spin configurations"
            print*, "'Si2' has shape = ", SIZE(Si2)
            print*, "'Si1' has shape = ", SIZE(Si1)
            print*, "Expected equal shape"
            CALL ABORT()
        end if

        ! checks on RBM input dimensions
        if ( SIZE(Wij, 1) /= SIZE(Ai) .OR. SIZE(Wij, 2) /= SIZE(Bj) ) then
            print*, "[ABORT] Wrong shapes in RBM parameters"
            print*, "Visible units dimension = ", SIZE(Ai)
            print*, "Hidden  units dimension = ", SIZE(Bj)
            print*, "Weights dimensions      = ", SIZE(Wij,1), ", ", SIZE(Wij,2)
            print*, "(expected = ", SIZE(Ai), ", ", SIZE(Bj), ")"
            CALL ABORT()
        end if

        M = SIZE(Bj)
        ALLOCATE(Tj2(M))
        ALLOCATE(Tj1(M))

        Tj2 = Bj + MATMUL(TRANSPOSE(Wij), Si2)
        Tj1 = Bj + MATMUL(TRANSPOSE(Wij), Si1)

        ! log( Psi(Si2) ) - log( Psi(Si1) )
        logPsi = SUM(Ai*(Si2-Si1)) + SUM(LOG(COSH(Tj2))) - SUM(LOG(COSH(Tj1)))

        DEALLOCATE(Tj2)
        DEALLOCATE(Tj1)

        RETURN

    end function logPsiDiff
```

Listing 14: *Inverse of a square matrix.*

```fortran
subroutine Inverse(A)
!
! Computes the inverse of the matrix using LAPACK subroutine ZGETRF().
!
!   Parameters
!   ----------
!
!   [INOUT] A : DOUBLE COMPLEX, DIMENSION(N,N)
!       Square matrix to be inverted.
!
    implicit none
    ! parameters
    DOUBLE COMPLEX, INTENT(INOUT) :: A(:,:)

    ! local variables
    INTEGER :: N, info, Lwork
    INTEGER, ALLOCATABLE :: ipiv(:,:)
    DOUBLE COMPLEX, ALLOCATABLE :: work(:)

    ! check that A is square matrix
    if ( SIZE(A, 1) /= SIZE(A,2) ) then
        print*, "[ABORT] Matrix is not squared, cannot be inverted"
        CALL ABORT()
    end if

    N = SIZE(A, 1)
    ALLOCATE(ipiv(N,N))

    ! call LAPACK zgtrf() to get LU decomposition
    CALL ZGETRF(N, N, A, N, ipiv, info)
    ! check exit
    if ( info < 0 ) then
        print*, "[ABORT] Illegal value found in LU decomposition"
        CALL ABORT()
    end if

    ! call LAPACK zgetri() to get optimal value of Lwork
    ALLOCATE(work(1))
    Lwork = -1
    CALL ZGETRI(N, A, N, ipiv, work, Lwork, info)
    Lwork = INT(work(1))
    DEALLOCATE(work)
    ALLOCATE(work(Lwork))
    ! call LAPACK zgetri() to get the inverse
    CALL ZGETRI(N, A, N, ipiv, work, Lwork, info)
    if ( info < 0 ) then
        print*, "[ABORT] Illegal argument value"
        CALL ABORT()
    else if ( info > 0 ) then
```

```fortran
            print*, "[ABORT] Matrix is singular"
            CALL ABORT()
        end if

        RETURN

    end subroutine Inverse
```

B.5 `Lanczos_mod` module

Lanczos subroutine is used to retrieve the correct solution given the hamiltonian of a system.

Listing 15: *Lanczos algorithm.*

```fortran
    function Lanczos(H, N, n_iter) result(W)
        !
        ! Compute the eighenvalues of a given hamiltonian
        ! using the Lanczos algorithm
        !
        !   Parameters
        !   ----------
        !   H  : DOUBLE COMPLEX, DIMENSION(2**N, 2**N)
        !       Matrix representation of the Hamiltonian
        !   N  :  INTEGER
        !       number of particles in the system
        !   n_iter  : INTEGER
        !       number of iter taken by the algorithm
        !
        !   Return
        !   ------
        !   W : DOUBLE PRECISION
        !       Eigenvalues in ascending order.

        implicit none

        ! Parameters
        DOUBLE COMPLEX , DIMENSION(:,:) :: H
        INTEGER :: n_iter, N

        ! Return
        DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: W

        ! Local variable
        INTEGER :: M, LWORK, LIWORK, INFO, i, LDZ, NZC, IL, IU
        DOUBLE PRECISION :: VL, VU
        DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: v_j_last, v_j, w_j_, w_j, WORK
        DOUBLE COMPLEX:: alpha, beta
        DOUBLE PRECISION, DIMENSION(:),   ALLOCATABLE:: alpha_list, beta_list, D, E
        DOUBLE COMPLEX  , DIMENSION(:,:), ALLOCATABLE :: Z
        INTEGER, DIMENSION(:), ALLOCATABLE:: ISUPPZ, IWORK
        LOGICAL:: TRYRAC

        !For the Lanczos algorithm we have to build a tridiagonal matrix.
        !The diagonal is represented by alpha_list while the off-diagonal is
        !represented by beta_list. Here beta_list is of the same dimension of
        !alpha list because the subroutine for the diagonalizarion require
        !that specific but in practice beta_list has n_iter - 1 elements.
        ALLOCATE(alpha_list(n_iter))
        ALLOCATE(beta_list (n_iter))

        ALLOCATE(v_j(2**N))
        ALLOCATE(v_j_last(2**N))

        !inizialize a random vector in the Hilbert Space of dimension 2**N,
        !the same dimension of the Hamiltonian, and normalize it
        call random_number(v_j)
        v_j = v_j / norm2(v_j)


        ALLOCATE(w_j_(2**N))
        ALLOCATE(w_j (2**N))

        w_j_ = matmul(H,v_j)

        alpha= dot_product(w_j_,v_j)
        !save the value of alpha
        alpha_list(1) = alpha
        w_j = w_j_ - alpha*v_j

        !Iterative procedure of Lanczos
        do i=2,n_iter
            beta= norm2(w_j)
            !save beta
            beta_list(i-1) = beta

            if (abs(beta) < 1d-12) then
                print *,"ERROR: beta=0"
                exit
            end if

            v_j_last= v_j
            v_j = w_j / beta

            w_j_= matmul(H,v_j)
            alpha=dot_product(w_j_,v_j)
            alpha_list(i) = alpha !save alpha

            w_j = w_j_ - alpha*v_j - beta* v_j_last

        end do


        !Copy the diagonal and the off diagonal. We pass the copy to the
        !subroutine beacuse D and E are overwrited and this first call of
        ! the diagonalization subroutine id for setup the parameters.
        ALLOCATE(D(n_iter))
        ALLOCATE(E(n_iter))

        D  = alpha_list
        E  = beta_list
        VL = 0.d0
        VU = 0.d0
        IL = 0
        IU = 0
        ALLOCATE(W(n_iter))
        M = 1
        ALLOCATE(Z(LDZ,max(1,M)))
        LDZ = n_iter
        NZC = -1
        ALLOCATE(ISUPPZ(2*max(1,M)))
        TRYRAC = .TRUE.
        ALLOCATE(WORK(1))
        LWORK = -1
        ALLOCATE(IWORK(1))
        LIWORK = -1

        !Call subroutine with NCZ=-1,LWORK=-1 and LIWORK =-1 to get the
        !values that optimize the subroutine itself
        call zstemr ('N', 'A', n_iter, D, E, VL, VU, IL, IU, M, W, Z, LDZ, NZC,&
                      ISUPPZ, TRYRAC, WORK, LWORK, IWORK, LIWORK, INFO)

        !We print INFO. If INFO = 0 we have that the subroutine was called
        ! without any problem
        if ( info /= 0 ) then
            print*, "[ABORT] First call to ZSTMR exit without success"
            call abort()
        end if

        M   = INT(Z(1,1))
        NZC = n_iter
        LWORK  = WORK(1)
        LIWORK = IWORK(1)

        DEALLOCATE(WORK)
        DEALLOCATE(IWORK)
        DEALLOCATE(Z)
        DEALLOCATE(ISUPPZ)

        ALLOCATE(WORK(LWORK))
        ALLOCATE(IWORK(LIWORK))
        ALLOCATE(Z(LDZ,max(1,M)))
        ALLOCATE(ISUPPZ(2*max(1,M)))

        TRYRAC=.TRUE.

        !Second call of the subroutine for the diagonalization with the setup
        !obtained from the previous call. We have that W is overwrited with
        !the eigenvalues of the hamiltonian.
        call zstemr ('N', 'A', n_iter, D, E, VL, VU, IL, IU, M, W, Z, LDZ, NZC,&
                      ISUPPZ, TRYRAC, WORK, LWORK, IWORK, LIWORK, INFO)

        if ( info /= 0 ) then
            print*, "[ABORT] Second call to ZSTMR exit without success"
            call abort()
        end if

        RETURN

    end function Lanczos
```

B.6 Python interface to Fortran

The script through which Fortran code should be called is presented here.

Listing 16: *Python interface to Fortran code.*

```
# %%
import argparse
from os import remove
from glob import glob
from subprocess import run
import sys
import json

# %%
parser = argparse.ArgumentParser(description='Ground state search through RBM')

```

```python
parser.add_argument("--noplot",
                    action="store_true",
                    help="Disable the energy plot")

ising = parser.add_argument_group("Ising Model")
ising.add_argument("--N",
                   type=int,
                   choices=range(1, 15),
                   metavar="[1,14]",
                   required=True,
                   help="Lattice size; if in 2D, it is the length of lattice side and must be in [1,3])")
ising.add_argument("--ll",
                   type=float,
                   default=0.2,
                   help="Strength of self interaction [default=0.2]")
ising.add_argument("--dim2",
                   action="store_const",
                   const=1,   # True
                   default=0,  # False
                   help="Whether to use 2D square lattice instead of 1D [default=False]")

rbm = parser.add_argument_group("RBM settings")
rbm.add_argument("--alpha",
                 type=int,
                 choices=range(1, 1000),
                 metavar='[> 0]',
                 default=2,
                 help="Hidden unit density (integer) [default=2]")
rbm.add_argument("--g",
                 type=float,
                 default=0.1,
                 help="Learning rate [default=0.1]")
rbm.add_argument("--updates",
                 type=int,
                 choices=range(1, int(1e6)),
                 metavar="[> 0]",
                 default=200,
                 help="Number of iterations in the weights update procedure [default=200]")

sim = parser.add_argument_group("Metrpolis settings")
sim.add_argument("--iter",
                 type=int,
                 choices=range(1, int(1e06)),
                 metavar="[> 0]",
                 default=500,
                 help="Number of iterations of the Metropolis algorithm [default=500]")
sim.add_argument("--burnin",
                 type=int,
                 choices=range(0, int(1e06)),
                 metavar="[0, iter-1]",
                 default=450,
                 help="Number of iterations to discard before saving the results [default=450]")
sim.add_argument("--autocorr",
                 type=int,
                 choices=range(1, int(1e06)),
                 metavar="[0, iter-burnin]",
                 default=1,
                 help="Number of iterations between two consecutives records [default=1]")

output = parser.add_argument_group("Output controls")
output.add_argument("--print",
                    type=int,
                    choices=range(1, int(1e06)),
                    metavar="P",
                    default=1,
                    help="Training information will be printed every P iterations
    [default=1]")
output.add_argument("--out",
                    type=str,
                    default="out.txt",
                    help="File where to store the output")
# %%
args = parser.parse_args()
with open('params.json', 'w') as f:
        json.dump(vars(args), f, indent=4)

if (args.dim2):
    if (args.N > 3):
        print("For square lattice, the maximum size is 3x3; setting to 3")
        args.N = 3
    args.N = args.N**2

if (args.burnin >= args.iter):
    print("'burnin' must be lower than 'iter'; setting to 0")
    args.burnin = 0

if (args.autocorr > args.iter-args.burnin):
    print("'autocorr' must be lower than 'iter' - 'burnin'; setting to 1")
    args.autocorr = 1

# %%
needed = [
    "debug.f90",
    "hamiltonian.f90",
    "Lanczos.f90",
    "main.f90",
    "others.f90",
    "random.f90",
    "rbm.f90",
    "simulation.f90"
]
found = glob("*.f90")
ok = True
for f in needed:
    if f not in found:
        print("Missing file:", f)
        ok = False
if not ok:
    sys.exit(
        "Unable to find all necessary files\n" +
        "[If you want Python implementation use '--python' argument]"
    )

run(["gfortran"] + needed + ["-o", "main.exe", "-llapack"])

fargs = list(vars(args).values())[1:]
fargs = list(map(lambda x: str(x), fargs))
run(["./main.exe"] + fargs)

# %%
if not args.noplot:
    import numpy as np
    import matplotlib.pyplot as plt

    data = np.loadtxt(args.out, comments="#")
    gs = float(np.loadtxt("tmp.txt"))
    plt.plot(data, color="C1", label="RBM")
    plt.axhline(y=gs, xmin=0, xmax=args.updates,
                color="C0", ls="--", lw=1, label="Lanczos")
    plt.text(0, gs+0.1, "E = {:.4f}".format(gs), color="C0")
    plt.legend()
    plt.show()

remove("tmp.txt")
```

B.7 Test script and results between Fortran and Python

The test script used to check Fortran code against the Python implementation is presented here, along with the full output.

Listing 17: *Script for testing the correctness of the code*

```
program test
    use simulation
    use rbm
    use debug
    use hamiltonian
    use others
    implicit none

    INTEGER :: S(5,3), ii
    INTEGER, ALLOCATABLE :: config(:,:)
    DOUBLE PRECISION :: g
    DOUBLE COMPLEX :: Ai(3), Bj(3), Wij(3,3), Psi
    DOUBLE COMPLEX, ALLOCATABLE :: Opk(:,:), H(:,:), Eloc(:), S_kk(:,:), Fk(:)

    Ai = (/-0.05d0, 0.08d0, 0.02d0/)
    CALL debugging(.TRUE., var=Ai, message="Ai")
    Bj = (/0.1d0, -0.05d0, -0.08d0/)
    CALL debugging(.TRUE., var=Bj, message="Bj")

    Wij(1,:) = (/-0.03d0,  0.02d0, 0.1d0 /)
    Wij(2,:) = (/ 0.07d0, -0.12d0, 0.03d0/)
    Wij(3,:) = (/-0.1d0 , -0.03d0, 0.05d0/)
    CALL debugging(.TRUE., var=Wij, message="Wij")

    CALL Metropolis(Ai, Bj, Wij, 100, 0, 1, config, Opk)
    CALL debugging(.TRUE., var=COUNT(config ==  1), message="'+1' in Metropolis
    configurations")
    CALL debugging(.TRUE., var=COUNT(config == -1), message="'-1' in Metropolis
    configurations")

    S(1, :) = (/1,  1,  1/)
    S(2, :) = (/1, -1, -1/)
    S(3, :) = (/1, -1, -1/)
    S(4, :) = (/1,  1, -1/)
    S(5, :) = (/1, -1, -1/)
    CALL debugging(.TRUE., var=S, message="S")

    H = Ising_1D(3, 0.2d0)
    CALL debugging(.TRUE., var=H, message="H")

    Eloc = LocalEnergy(S,H,Ai,Bj,Wij)
    CALL debugging(.TRUE., var=Eloc, message="Eloc")

    Psi = logPsiDiff(S(2,:), S(1,:), Ai, Bj, Wij)
```

```fortran
    CALL debugging(.TRUE., var=Psi, message="log Psi diff")

    ALLOCATE(S_kk(15,15))
    S_kk = 0.d0
    do ii = 1, 15
        S_kk(ii,ii) = 1.d0
    end do
    S_kk(3,3) = -0.5d0
    CALL debugging(.TRUE., var=S_kk, message="S_kk")

    ALLOCATE(Fk(15))
    Fk = 1.d0
    Fk(8) = -0.7d0
    Fk(14) = 0.25d0
    CALL debugging(.TRUE., var=Fk, message="Fk")

    g = 0.5d0
    CALL RBM_update(Ai, Bj, Wij, S_kk, Fk, g)
    CALL debugging(.TRUE., var=Ai, message="Ai after update")
    CALL debugging(.TRUE., var=Bj, message="Bj after update")
    CALL debugging(.TRUE., var=Wij, message="Wij after update")



end program test
```

Listing 18: *Results of the test produced by Fortran code*

```

 ==================================================
 Ai
 [DOUBLE COMPLEX array, dimension=  3]
        (-5.0000000000000028E-002,0.0000000000000000)
         (8.0000000000000017E-002,0.0000000000000000)
         (2.0000000000000004E-002,0.0000000000000000)

 ==================================================


 ==================================================
 Bj
 [DOUBLE COMPLEX array, dimension=  3]
              (0.10000000000000001,0.0000000000000000)
        (-5.0000000000000028E-002,0.0000000000000000)
        (-8.0000000000000017E-002,0.0000000000000000)

 ==================================================


 ==================================================
 Wij
 [DOUBLE COMPLEX matrix, dimension=  3, col=  3]
( -0.30000E-01,    0.0000    )  (  0.20000E-01,    0.0000    )  (  0.10000    ,    0.0000
      )
(  0.70000E-01,    0.0000    )  ( -0.12000    ,    0.0000    )  (  0.30000E-01,    0.0000
      )
( -0.10000    ,    0.0000    )  ( -0.30000E-01,    0.0000    )  (  0.50000E-01,    0.0000
      )
 ==================================================


 ==================================================
 '+1' in Metropolis configurations
 [INTEGER]         159
 ==================================================


 ==================================================
 '-1' in Metropolis configurations
 [INTEGER]         141
 ==================================================


 ==================================================
 S
 [INTEGER matrix, row=  5, col=  3]
        1          1          1
        1         -1         -1
        1         -1         -1
        1          1         -1
        1         -1         -1
 ==================================================


 ==================================================
 H
 [DOUBLE COMPLEX matrix, dimension=  8, col=  8]
(  -2.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )  ( -0.20000    ,   -0.0000
      )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,
    -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )
( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000
      )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,
    -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )
( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  (   2.0000    ,   -0.0000
      )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,
    -0.0000    )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )
(  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )  ( -0.20000    ,   -0.0000
      )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,
    -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )
( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000
      )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,
    -0.0000    )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )
(  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000
      )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )  (   2.0000    ,
    -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )
(  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000
      )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,
    -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,   -0.0000    )
(  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  (  -0.0000    ,   -0.0000
      )  ( -0.20000    ,   -0.0000    )  (  -0.0000    ,   -0.0000    )  ( -0.20000    ,
    -0.0000    )  ( -0.20000    ,   -0.0000    )  (  -2.0000    ,   -0.0000    )
 ==================================================


 ==================================================
 Eloc
 [DOUBLE COMPLEX array, dimension=  5]
              (-2.5865575144810693,0.0000000000000000)
             (-0.67569521762084805,0.0000000000000000)
             (-0.67569521762084805,0.0000000000000000)
             (-0.60241333036069289,0.0000000000000000)
             (-0.67569521762084805,0.0000000000000000)

 ==================================================


 ==================================================
 log Psi diff
 [DOUBLE COMPLEX]             (-0.20793134812583486,0.0000000000000000)
 ==================================================


 ==================================================
 S_kk
 [DOUBLE COMPLEX matrix, dimension= 15, col= 15]
(   1.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   1.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  ( -0.50000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   1.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   1.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   1.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   1.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   1.0000    ,   0.0000    )  (
     0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )
(   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
      0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (
     1.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000
      )  (   0.0000    ,   0.0000    )  (   0.0000    ,   0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )
(   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (
    0.0000    ,    0.0000    )  (   1.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )
(   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (
    0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   1.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )
(   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (
    0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   1.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )
(   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (
    0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   1.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )
(   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (
    0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   1.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )
(   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (
    0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000
      )  (   0.0000    ,    0.0000    )  (   0.0000    ,    0.0000    )  (   0.0000    ,
     0.0000    )  (   1.0000    ,    0.0000    )
 ==================================================


 ==================================================
 Fk
 [DOUBLE COMPLEX array, dimension= 15]
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
             (-0.69999999999999996,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)
              (0.25000000000000000,0.0000000000000000)
               (1.0000000000000000,0.0000000000000000)

 ==================================================


 ==================================================
 Ai after update
 [DOUBLE COMPLEX array, dimension=  3]
            (-0.55000000000000004,0.0000000000000000)
            (-0.41999999999999998,0.0000000000000000)
              (1.0200000000000000,0.0000000000000000)

 ==================================================



 ==================================================
 Bj after update
 [DOUBLE COMPLEX array, dimension=  3]
            (-0.40000000000000002,0.0000000000000000)
            (-0.55000000000000004,0.0000000000000000)
            (-0.57999999999999996,0.0000000000000000)

 ==================================================



 ==================================================
 Wij after update
 [DOUBLE COMPLEX matrix, dimension=  3, col=  3]
( -0.53000    ,    0.0000    )  (  0.37000    ,    0.0000    )  ( -0.40000    ,    0.0000
      )
( -0.43000    ,    0.0000    )  ( -0.62000    ,    0.0000    )  ( -0.47000    ,    0.0000
      )
( -0.60000    ,    0.0000    )  ( -0.15500    ,    0.0000    )  ( -0.45000    ,    0.0000
      )
 ==================================================
```

Listing 19: *Results of the test produced by Python code*

```
**************
DEBUGGING TEST
**************
N = 3 | M = 3 | alpha = 1 | k = N+M+N*M=15
p = # MC samples = 5
Ising 1D with h = 0.2
Learning rate g = 0.5

Ai = [-0.05  0.08  0.02]
Bj = [ 0.1  -0.05 -0.08]
Wij =
 [[-0.03  0.02  0.1 ]
 [ 0.07 -0.12  0.03]
 [-0.1  -0.03  0.05]]

'+1' in Metropolis configurations 153
'-1' in Metropolis configurations 147

MC spin configurations S =
 [[ 1  1  1]
 [ 1 -1 -1]
 [ 1 -1 -1]
 [ 1  1 -1]
 [ 1 -1 -1]]

Ising hamiltonian =
 [[-2.  -0.2 -0.2 -0.  -0.2 -0.  -0.  -0. ]
 [-0.2 -0.  -0.  -0.2 -0.  -0.2 -0.  -0. ]
 [-0.2 -0.   2.  -0.2 -0.  -0.  -0.2 -0. ]
 [-0.  -0.2 -0.2 -0.  -0.  -0.  -0.  -0.2]
 [-0.2 -0.  -0.  -0.  -0.  -0.2 -0.2 -0. ]
 [-0.  -0.2 -0.  -0.  -0.2  2.  -0.  -0.2]
 [-0.  -0.  -0.2 -0.  -0.2 -0.  -0.  -0.2]
 [-0.  -0.  -0.  -0.2 -0.  -0.2 -0.2 -2. ]]
Local Energy = [-2.58655751 -0.67569522 -0.67569522 -0.60241333 -0.67569522]

Log(Psi2) - log(Psi1) = -0.20793134812583486

Covariance matrix S_kk =
 [[ 1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.  -0.5  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.
   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   1. ]]

Forces Fk = [ 1.    1.    1.    1.    1.    1.    1.   -0.7   1.    1.    1.    1.
  1.    0.25  1.  ]

After update:
Ai = [-0.55 -0.42  1.02]
Bj = [-0.4  -0.55 -0.58]
Wij =
 [[-0.53   0.37  -0.4  ]
 [-0.43  -0.62  -0.47 ]
 [-0.6   -0.155 -0.45 ]]
```