1 Local Algorithm

1.1 description

This section focuses on a way to move through the configurations. The local Algorithm need to implement two distinct moves :

— the splitline move This move is supposed to inverse a complete line. To be implemented we firstly chose a spin, then check if the all the spins upward have the same value, and in that case invert all opf them.

— the kink move This move move represents the effect of the quantic spins. The part Jx and Jy create variation from the usual spin model. It is taken in account with this move, which flip the 4 spins around a black case, if the two left spins are the same and are the oposite of the two right spins.

1.2 domain of validity

1.2.1 Winding number

It can not deal with periodic boundary conditions. Indeed, because the updates are either local or along a temporary line, the winding number is fixed. The winding number is an integer counting the number of period that the world-line takes to return to the same position.

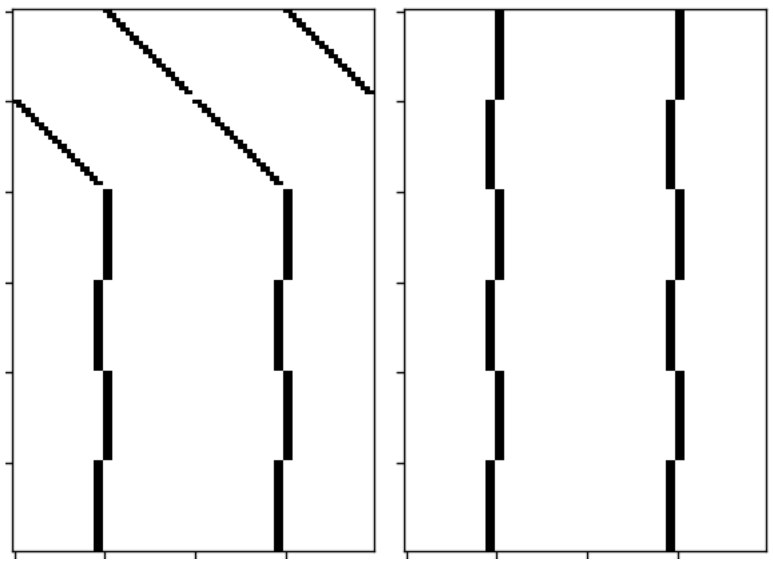


Figure 1 – Configurations with W = 2 (left) and W = 1 (right)

1.2.2 straight line only can be flipped

After a few attempts to split any world-line we only add wrong result. Indded we realized it caused an error in the detailed balance. Indeed if we start from the following situation 1 and split the worldline 3 we get to the situation 2.

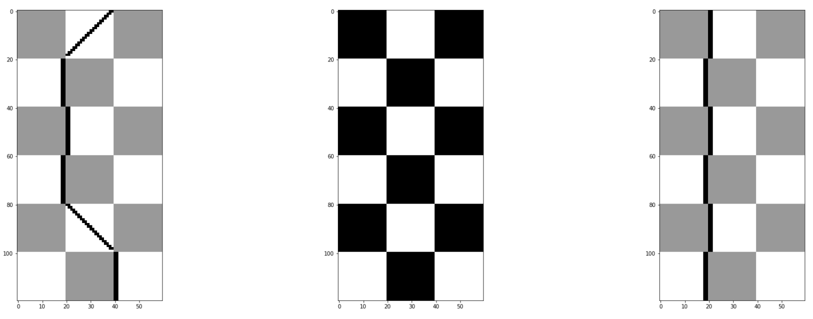


FIGURE 2 – situation 1 (left), situation 2 (middle), situation 3 (right)

From the situation 2 it is not possible to come back at the situation 1 with a simple splitline. Indeed such move only creates situation 3. To do so a splitline and a kink move is mandatory. The detailed balance is in this case unbalanced. The solution is to accept to flip straight world-line only.

To code a Monte Carlo algoriothm, we then had to condensed the two moves in one. We called it stoch_move(threshold), this algo has a probability threshold to make a splitline move, and 1-threshold to make a kink move.

This algorithms

1.3 first results

For the proper parameter the algorithm enable to compute the mean energy. In the following we'll focuse on the case $Jx = Jy = Jz = 1, nspins = 4, beta = 1$. Firstly we made a few run of the montecarlo algorithm, with a small length_cycle (10) and a small number of measures(100 000), to have a first look.

The found the following results :

— theorical energy : -0.6330214400416617

— Energy for m=2 : -0.6368988448758103 +/- 0.002502566964381992

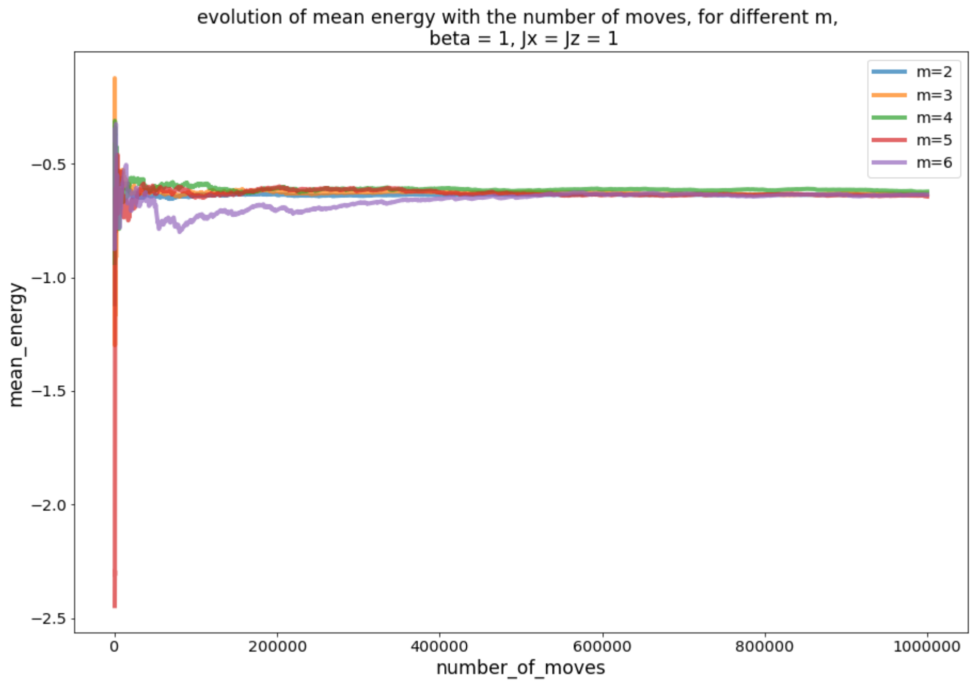
— Energy for m=3 : -0.6338973315707276 +/- 0.0027341143579975723

— Energy for m=4 : -0.6229915669089899 +/- 0.002819998026568314

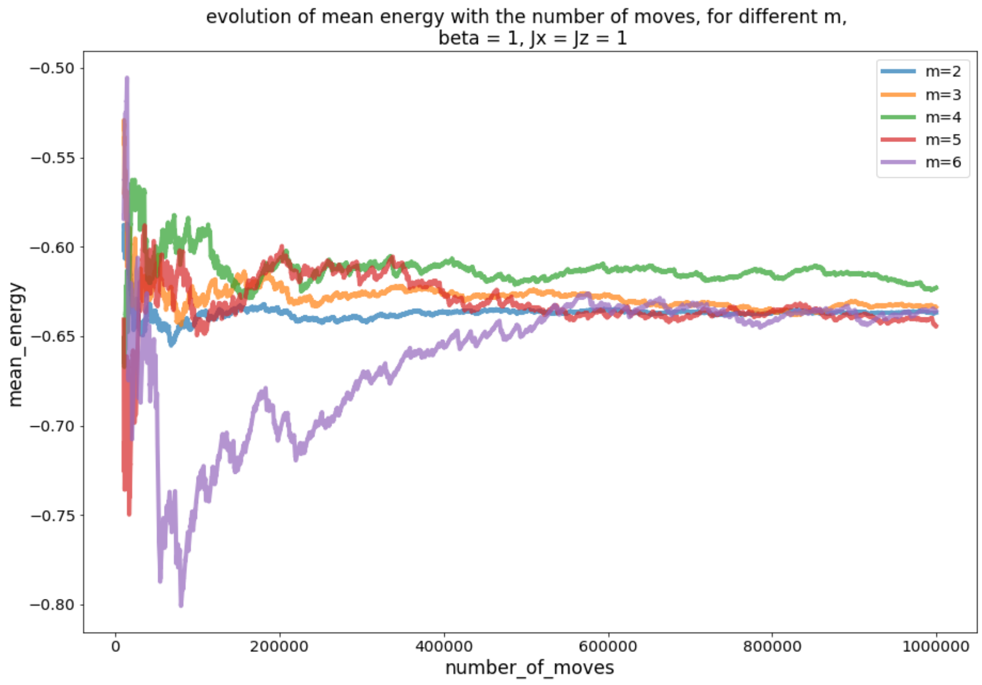
— Energy for m=5 : -0.6445715838659634 +/- 0.0029559595617620286

— Energy for m=6 : -0.6358614671234866 +/- 0.0029930079421800816

At the first glance it seems that the algorithm give a good approxiamtion of the result. We firstly wanted to have an idea of the evolution of the mean energy with the number of measures.



We clearly feel that the 5 results converge to the same limit around -0.63. We also observe that the case m = 6 takes more moves to converge. The beginning of the curves doesn't have much meaning. It has an high dependance regarding the initial configuration. We can have a better look on the end of the graph :



This graph enables us to see two things more clearly. Firstly the number of moves necessary for the mean to converge rises with m. Secondly the choice of m seem to involve a bias in the mean. While mtrotter rises, the number of configurations follows and the weight changed by the local_update move decrease

1.4 autocorrelation function

To have a better idea of the length between two measures, we computed the autocorelation function. This function is defined for an observable O by the following formula

1.5 limits

2 Loop Algorithm

In this section, we will discuss the loop algorithm, introduced in the Assaad and Evertz article page 12.

2.1 Failure of the local algorithm

The previous algorithm is not reliable for the three following reasons.

2.1.1 Acceptance rate

The rate of acceptance of the move is highly dependant on the parameters. For instance, should ...

2.1.2 Decorrelation

The decorrelation between the configuration is rather slow ...

2.2 The loop update

2.2.1 Example

The idea of Assaad and Evertz is to map the periodic boundary conditions model onto another one called "six-vertices model". Thanks to the result obtained on this model, we are now able to implement an update which allows fast decorrelation, high acceptance rate and changing winding number. Let us imagine each tile is replaced by its representation in the six-vertices model which is described in figure 3.

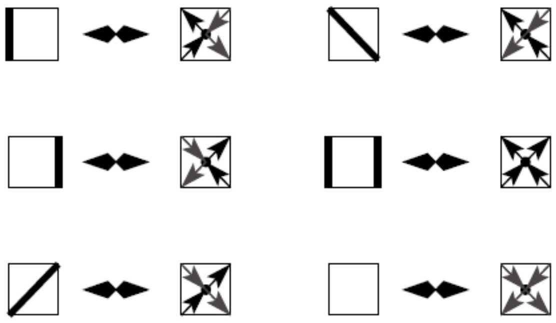


FIGURE 3 – Rule to change a tile in a vertex plaquette

The rule is the following : each spin up is transformed into an arrow going up ; each spin down is transformed into an arrow going down. Given this transformation one is now able to imagine loops on the pattern. One starts from a given spin and a given vertex plaquette, and go along the arrow in the same direction. If one does not choose two times the same arrow, he will come back to the first spin and thus create a loop. This mechanism is illustrated by the figure 4. The described loop "goes through" the right side to come back to the left side thanks to the periodic boundary conditions.

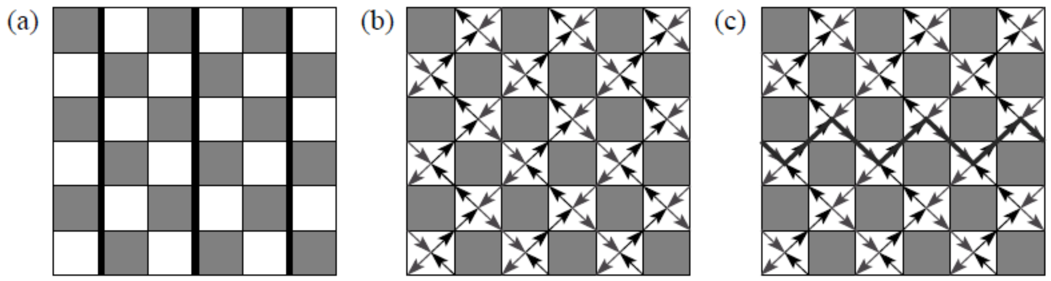


Figure 4 – Illustration of the creation of a loop

With the chosen loop, one can now flip the spins along the loop, which corresponds to changing the direction of all the arrows. This generates a new configuration. In the described one in figure 4 and figure 5, the winding number is changed from 1 to 3. One can observe that even this quite simple update (only one loop has been studied), the new configuration seems less dependant that any other that would have been generated thanks to the local algorithm.

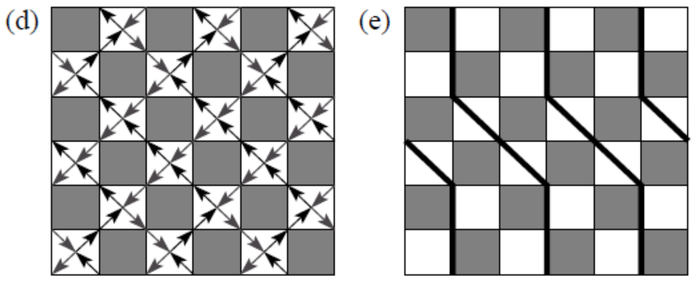


Figure 5 – Update by flipping the spins along the loop

2.2.2 Formalism

For the quantum Monte-Carlo algorithm, the detailed balance must be fulfilled. For a given tile in the configuration S, and for any other tile S', the detailed balance is given by equation (1)

$$W(S)P(S \to S') = W(S')P(S' \to S) \tag{1}$$

In order to make a move on the configuration, one has to change all the tiles in vertex plaquettes, then to choose loops and to choose whether to flip the spins along it or not. The process is clearer if one creates a virtual configuration called graph. For each tile, the associated vertex plaquette will be replaced by a graph-tile representing the choice of the arrows. The choices are either the loop goes vertically, either diagonally, either horizontally along the tile. Actually, there is another graph possible which switch all the spins on the tile, but it will not be considered here. Summing all the graph will create loops on the configuration. Figure 6 is an example of total graph

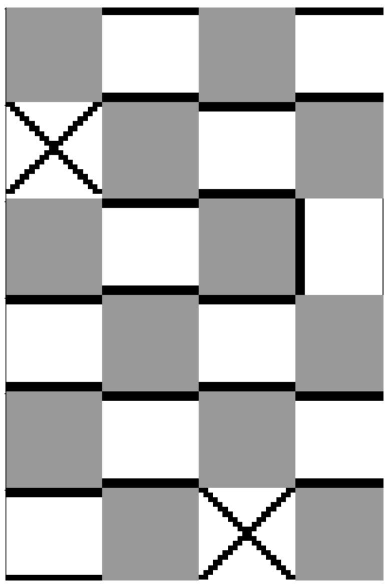


FIGURE 6 – Example of graph ; one can verify that all loops are closed

First of all, one needs to know which graphs can be reached by a given tile. Let us give an example for the tile with all the spins down, the white tile. The associated vertex plaquette is the one with all the arrow heading down. If the loop arrives from the upper left side of the tile, it can either go to the bottom left or to the bottom right. Thus, only 2 graphs are allowed : either a vertical graph or a cross graph. The illustration is given in figure 7 for all tiles, one can verify the association following the previous logic applied to the white tile.

However, depending on the interactions and the temperature, the graphs will not be equally possible. One has to choose weights for each of them. Let us call $W(S)$ the weight of the tile S and $W(S, G)$ the weight of the graph

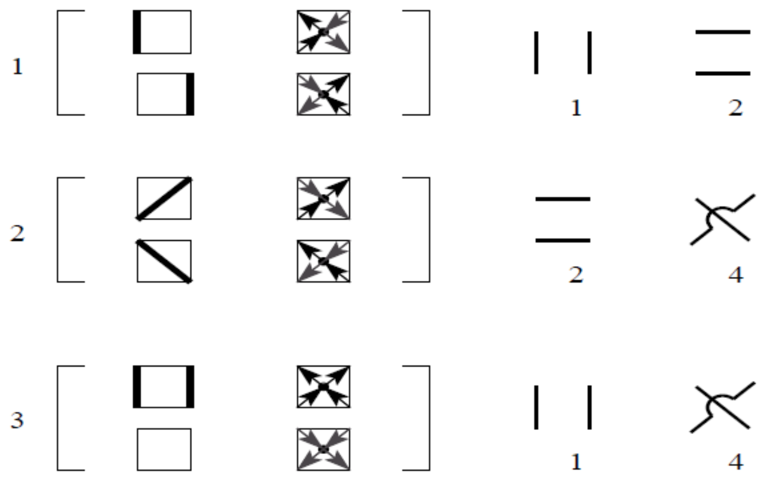


FIGURE 7 – Possible graphs for tiles

G for a tile S. The equation (2) is required to have a total probability equal to one, and then be sure that every tile is replaced by a graph.

$$\sum_G W(S,G) = W(S) \tag{2}$$

So we choose a graph from a given plaquette with probability

$$P(S \to (S,G)) = \frac{W(S,G)}{W(S)} \tag{3}$$

The detailed balance written with the graph transition

$$\begin{aligned} W(S)\sum_G P(S \to (S,G))P((S,G) \to (S',G)) = \\ W(S')\sum_{G'} P(S' \to (S',G'))P((S',G') \to (S,G')) \end{aligned} \tag{4}$$

By using equation (3) is can be rewritten

$$\begin{aligned} \sum_G W(S,G)P((S,G) \to (S',G)) = \\ \sum_{G'} W(S',G')P((S',G') \to (S,G')) \end{aligned} \tag{5}$$

Then one can choose the different parameters to make the computation easier. For instance, the weights of the graph is set constant, not dependant on the spin configuration. This is easier to encode and clearer for the detailed balance :

$$\forall G, \forall (S, S'), W(S,G) = W(S',G) = V(G) \tag{6}$$

Moreover, with the use of the heat-bath algorithm for flipping, the probability to change the spin from S to S' given a choice of graph G can be set to $P((S,G) \to (S',G)) = \frac{W(S',G)}{W(S',G)+W(S,G)}$. Thus the detailed balance is verified and the code is easy to encode. Indeed

$$\begin{aligned} P((S,G) \to (S',G)) &= \frac{W(S',G)}{W(S',G)+W(S,G)} \\ &= \frac{V(G)}{V(G)+V(G)} \\ &= \frac{1}{2} \end{aligned} \tag{7}$$

The equation (7) means that each loop is flipped with probability 1/2.
To set the weights of the graphs $V(G)$, equations (2) and (6), with the use of the notations of figure 7 give

$$\begin{cases} W(S=1) = V(G=1) + V(G=2) \\ W(S=2) = V(G=2) + V(G=4) \\ W(S=3) = V(G=1) + V(G=4) \end{cases} \tag{8}$$

This set of equation (8) has the solution :

$$\begin{cases} V(G=1) = \frac{1}{2}[W(S=1) + W(S=3) - W(S=2)] \\ V(G=2) = \frac{1}{2}[W(S=1) + W(S=2) - W(S=3)] \\ V(G=4) = \frac{1}{2}[W(S=2) + W(S=3) - W(S=1)] \end{cases} \tag{9}$$

Then, the weights $W(S)$ are

$$\begin{cases} W(S=1) = \exp(\Delta\tau J_z/4)\cosh(\Delta\tau J_x/2) \\ W(S=2) = \exp(\Delta\tau J_z/4)\sinh(\Delta\tau J_x/2) \\ W(S=3) = \exp(-\Delta\tau J_z/4) \end{cases} \tag{10}$$

The weights of each graph is now set. All the problem is characterized and the implementation is possible. However, the *Metropolis* algorithm is valid only for positive weights for $W(S)$ and $V(G)$. This fixes conditions on J_z and

J_x. With the condition $V(G=2) > 0$:

$$\begin{aligned}
&\exp(\Delta\tau J_z/2)\sinh(\Delta\tau J_x/2) + \exp(\Delta\tau J_z/2)\cosh(\Delta\tau J_x/2) - 1 > 0 \\
&\exp(\Delta\tau J_z/2)\exp(\Delta\tau J_x/2) > 1 \\
&J_x > -J_z
\end{aligned} \tag{11}$$

With the condition $V(G=4) > 0$:

$$\begin{aligned}
&\exp(\Delta\tau J_z/2)\sinh(\Delta\tau J_x/2) + 1 - \exp(\Delta\tau J_z/2)\cosh(\Delta\tau J_x/2) > 0 \\
&\exp(-\Delta\tau J_z/2) > \exp(-\Delta\tau J_x/2) \\
&-J_z > -J_x \\
&J_x > J_z
\end{aligned} \tag{12}$$

The final condition is then

$$J_x > |J_z| \tag{13}$$

If ever $|J_z|$ is higher than J_x, the algorithm can be implemented with another type of graph, called $G = 3$, that propagates the loop along all the arrows, which means all the spins on the tile are flipped. This is a graph accessible by all the vertex plaquettes, which allows a new system (8) and a supplementary degree of freedom to set the weights $V(G)$. This option will not be considered in our algorithm.

2.3 Encoding the loop algorithm

The whole methods will be encoded in an algorithm

2.3.1 Parameters

First, we create a class containing the necessary parameters.

- ⋆ The interactions J_x and J_z
- ⋆ The number of spins *n_spins*
- ⋆ The division of the imaginary time *m_trotter* and $\Delta\tau$
- ⋆ The representation of the spins *spins*
- ⋆ The pattern configuration *pattern*
- ⋆ The graph configuration *total_graph*
- ⋆ The list of energies depending on the concerned tile *energymatrix*
- ⋆ The list of tile weights depending on the concerned tile *weightmatrix*
- ⋆ The list of arrays imaging the tiles *cases*
- ⋆ The list of arrays imaging the graphs *graphs*
- ⋆ The graph weights depending on the graphs $w11, w12, w22, w24, w31, w34$

2.3.2 Methods

Here are the methods used in the Quantum Monte Carlo algorithm.

- $\star$ *total_energy*() : Thanks to array combination and the use of masks, this methods extract the energy of each tile on the pattern then sum those energies.
- $\star$ *weight*() : Thanks to array combination and the use of masks, this methods extract the weight of each tile on the pattern then make the product of those weights to return the full weight of the combination.
- $\star$ *spins_to_pattern*() : With the use of array combination, this method computes the pattern from the spin configuration. At each coordinate (i, j) is a number corresponding to the tile.
- $\star$ *createimage*() : Returns an array imaging the total world-line pattern.
- $\star$ *tile_in_graph*(*pos*) : Chooses a graph for the tile in position $(pos[0], pos[1])$ with probability $\frac{V(G)}{W(S)}$.
- $\star$ *set_total_graph*() : Goes over the whole parameter *total_graph* and choose for each position a graph thanks to the previous method *tile_in_graph*(*pos*). ...
- $\star$ $find\_next(pos, graph)$: This method is used to get the loops from the graph. The *pos* and *graph* parameters corresponds respectively to the position of a spin and the one of a graph. The method returns a new spin position and a new graph position after propagating the loop through the given *graph* from the *spin* position. For instance, if the graph is $G = 1$, *i.e.* vertical, and the spin is on the upper left side on the graph, the method will return the spin just below the first spin and the graph down left the first graph.
- $\star$ $find\_loops()$: This method will find all the loops and flip them with probability $\frac{1}{2}$. It goes over all the spin, and when any spin is not already in a loop, is uses the method $find\_next()$ to know the following spin in the associated loop, thus until is goes back to the same spin. At each step, is stores the spin to know which spins have already been treated. When it finds a not treated spin, it chooses with probability $\frac{1}{2}$ whether to flip all the spin along it or not.
- $\star$ *creategraph*() : Returns an array imaging the total graph configuration.
- $\star$ *pattern_to_string*() : Transforms the *spins* into a string to allow the script *check_loop_ergodicity.py* to check if the algorithm is ergodic, i.e. that all configurations are reachable.

- ⋆ $QMC\_step()$: This method applies a step on the configuration. First, is computes the pattern from the spins. Then, it uses the computed pattern to set the $total\_graph$. Then, thanks to the $find\_loop()$ method it changes the spins to a new configuration.
- ⋆ $Quantum\_Monte\_Carlo(n\_warmup, n\_cycles, length\_cycle)$: This method uses the metropolis algorithm to compute the mean energy. First, the initial state with all spins down is "warmed up" by n_warmup cycles. Then, there are n_cycles cycles of $length\_cycle$ steps. For each cycle, the energy is measured once. The parameter $length\_cycle$ is used to get rid of the correlation between two successive configuration. This parameter can be optimize.

2.4 Results

2.4.1 Case $J_x = J_z$

Parameters For this case only, the graph $G = 4$ is not considered, because it's weight is null.

Validity of the algorithm This case allowed us to check that our algorithm worked well with simple parameters. We used our code $ExactComputation$ to know the theoretical values for the following parameters : $J_x = J_z = -1$, $m\_trotter = 10$, $N\_spins = 8$. The results for the Quantum Monte Carlo are computed thanks to the script *testloop.py*.

β	Theoretical mean energies	Monte Carlo computed mean energies
0.1	-0.146105	-0.175369 ± 0.045031
0.2	-0.283938	-0.318095 ± 0.052067
0.3	-0.412980	-0.428188 ± 0.047555
0.4	-0.532973	-0.550819 ± 0.046009
0.5	-0.643891	-0.614586 ± 0.039800
0.6	-0.745901	-0.775494 ± 0.040258
0.7	-0.839321	-0.858818 ± 0.039029
0.8	-0.924584	-0.912076 ± 0.034779
0.9	-1.002198	-0.960219 ± 0.035408
1.0	-1.072715	-1.081081 ± 0.035624

This computation confirms that our code runs and produces the good results for parameters with $J_x = J_z$. This was tested for fairly few n_cycles in the Metropolis algorithm (10000), however it is sufficient.

Ergodicity In order to test our algorithm, we counted the number of different patterns that could be reached. If the algorithm is ergodic, is reaches all those patterns, that we are able to count. The script *check$_{l}$oop$_{e}$rgodicity.py* allowed us to verify this property.

Autocorrelation

Result dependence on m

2.4.2 Case $J_x > |J_z|$

Parameters For this case, all graphs are considered. *During the encoding, because we faced difficulties, we tried to implement the code with another rule of repartition of the graphs' weights $V(G)$ such as the graph $G = 4$ was accessible only from the diagonal tiles $S = 2$ (i.e. $W(S = 3, G = 4) = 0$). This algorithm did not work, and the article written by Wessel that proposed such a repartition was wrong. Indeed, in this particular case, the detailed balance is not verified, as through the graph $G = 4$, the tiles $S = 2$ are transformed to $S = 3$, and tiles $S = 3$ can not reach tiles $S = 2$.*

Validity of the algorithm We used our code *ExactComputation* to know the theoretical values for the following parameters : $J_x = 1.0$, $J_z = 0.5$, $m\_trotter = 10$, $N\_spins = 8$. The results for the Quantum Monte Carlo are computed thanks to the script *testloop.py*.

β	Theoretical mean energies	Monte Carlo computed mean energies
0.1	-0.114286	0.175369 ± 0.045031
0.2	-0.231759	0.318095 ± 0.052067
0.3	-0.351769	0.428188 ± 0.047555
0.4	-0.473585	-0.550819 ± 0.046009
0.5	-0.596416	-0.614586 ± 0.039800
0.6	-0.719437	-0.775494 ± 0.040258
0.7	-0.841814	-0.858818 ± 0.039029
0.8	-0.962735	0.912076 ± 0.034779
0.9	-1.081434	-0.960219 ± 0.035408
1.0	-1.197217	-1.081081 ± 0.035624