Gradient Descent Optimization of MPS for Ground State (MPS)

Cristian Emiliano Godinez Ramirez
(Dated: August 5, 2022)

In this paper I will explore the implementation of the gradient descent method (GDS) to find the ground state of the Ising Hamiltonian. After a brief theoretical introduction, I explore the parameter regimes that best suit the algorithm, including the optimization of the ansatz state, learning rate, and Hamiltonian parameters. The comparison of GDS against other well-known algorithms for ground state preparation, such as TEBD and DMRG, showcases the deficiencies of the algorithm. I then finalize the work with a discussion of possible improvements as well as comments on its implementation.

I. INTRODUCTION

The analysis of quantum many-body systems is of utter importance in areas such as condensed matter physics, as they give rise to interesting quantum phenomena. However, in order to do so, it is necessary to deal with the exponential growth in computational resources that results from the exponential growth in the Hilbert space when increasing the system size. In order to perform large scale numerical simulations, tensor networks emerge as an ubiquitous tool that can be used to address this challenge [1].

In this report, I will explore a gradient descent-based algorithm that optimizes an MPS to obtain the ground state of the well known Ising Hamiltonian in 1D. The following equation describes the Hamiltonian of the system in question:

$$H = -J\sum_i \sigma_i^x \sigma_{i+1}^x - g\sum_i \sigma_i^z \tag{1}$$

Such Hamiltonian can also be expressed as a Matrix Product Operator, as explained in [1]. The matrix at each site i is then expressed:

$$W^{[i]} = \begin{bmatrix} I & \sigma^x & -g\sigma^z \\ 0 & 0 & -J\sigma^Z \\ 0 & 0 & I \end{bmatrix} \tag{2}$$

It is important to notice that each component is in itself an operator of dimension $d \times d$, with d the local site basis dimension ($d = 2$ for 1/2 spin systems).

MPS

Here, the quantum states are represented as matrix product states (MPS), as they are the building blocks of many tensor networks algorithms. This is a useful ansatz class that allows us to represent a pure quantum state in a 2^N dimensional Hilbert space as a tensor train, where N is the **number of sites** (spins) in our system. This representation is particularly useful for area law states, i.e. states for which the entanglement grows with the area of the cut, which correspond to the ground states of a 1D gapped and local Hamiltonian, such as the Ising Model [2]. An intuitive explanation relies on the fact that on a gapped ground state, fluctuations only act within the correlation length such that only sites near the cut (partition) are entangled.

These observations justify the choice to represent the groundstate in this problem as an MPS. In this representation, the state is written as [1]:

$$|\Psi\rangle = \sum_{j_1,\ldots,j_N} M^{[1]j_1} M^{[2]j_2} \ldots M^{[N]j_N} |j_1, j_2, \ldots, j_N\rangle \tag{3}$$

Where each $M^{[n]j_n}$ are $\chi_n \times \chi_{n+1}$ dimensional matrices, and the χ are the virtual bond dimensions. This means that we have $d = 2$ matrices per site.

Gradient Descent

A common situation in the field of optimization is the problem of finding input values $\vec{x}$ that minimize a cost function $f(\vec{x}) : \mathbb{R}^N \to \mathbb{R}$. Among the many approaches that exist out there, a simple yet useful method is the Gradient Descent (GDS), a first-order iterative algorithm.

For a function $f(\vec{x})$ that is defined and differentiable in the neighbourhood of $\vec{x} = w^{[i]}$, the gradient $\nabla f(w^{[i]})$ is defined as the vector valued function that points locally in the direction of fastest increase with its magnitude equal to the rate of this change [3]. Using this idea, we can take steps of size η (learning rate) in the direction of the negative gradient, as this points locally in the direction of steepest descent of f [4]. The update looks as follows:

$$w^{[i+1]} = w^{[i]} - \eta \nabla f(w^{[i]}) \tag{4}$$

By choosing an appropriate initial guess $w^{[0]}$ we can iteratively update the components of w until convergence, or up to a certain number of iterations. The summarized algorithm is:

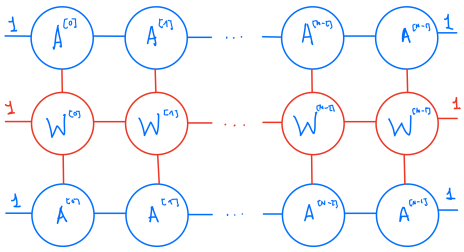

FIG. 1: Expectation value $\langle\psi|\, H\, |\psi\rangle$ of MPS ψ represented by the tensors $A^{[i]}$, and the Hamiltonian in MPO form represented by the $W^{[i]}$. System of size N. The last and first dimensions in the MPO and MPS are one to ensure the contraction returns a scalar.

Algorithm 1 Gradient Descent

Require: cost function: f, step: $\eta > 0$, ansatz: $w^{[0]}$, iterations: T
for `i = 0,1,...,T-1` **do**
$w^{[i+1]} = w^{[i]} - \eta \nabla f(w^{[i]})$
end for
return $w^{[i+1]}$ ▷ these are the values that minimize f(x)

Gradient of a Tensor

Some important nuances about this algorithm must be taken into account when applying it to tensor networks. In particular, to optimize our MPS it is necessary to calculate the derivative of the cost function $f(\psi)$ with respect to each of the tensors $M^{[i]}$ in the tensor train. Which will be from now on referred to as $A^{[i]}$, in allusion to its future left-orthonormalization in the implementation. To obtain this derivative, we should now look at how the expectation value $\langle\psi|\, H\, |\psi\rangle$ looks like in a tensor diagram, as shown in figure 1.

Now, there are two important properties to consider that will ease the computation of the derivative:

$$\frac{\partial f(\psi)}{\partial A^{[i]}} = \frac{\partial \langle\psi|\, H\, |\psi\rangle}{\partial A^{[i]}} \tag{5}$$

First, from the field of complex analysis, an interpretation of the Cauchy-Riemann equations is the **independence of the variable $A^{[i]}$ and its complex conjugate** $A^{[i]*}$ [5]. Taking the Wirtinger derivative of a function depending only on $A^{[i]}$ with respect to $A^{[i]*}$ is then:

$$\frac{\partial f(A^{[i]})}{\partial A^{[i]*}} = 0 \tag{6}$$

Using this property, we can then see the expectation value $f(\psi)$ as a linear function of $A^{[i]}$, for $i \in [0, ..., N-1]$.

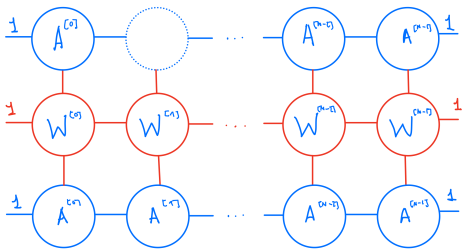

FIG. 2: Diagramatic representation of $\partial \langle\psi|\, H\, |\psi\rangle / \partial A^{[i]}$ with $i = 1$.

In addition, if a function f is linear in $A^{[i]}$, and is scalar-valued, i.e. it can be represented as a tensor network with no open legs, then taking its derivative with respect to $A^{[i]}$ is equivalent to **redrawing the whole network without** $A^{[i]}$ [6].

These two properties, can be then used together to evaluate equation 5. A diagramatic representation of this is shown in figure 2.

A caveat, however, is that using the GDS method with the current cost function can yield undesired behaviours. This is because the updated states resulting from one step of the algorithm might not be normalized, since this is not imposed in any of the steps. To solve this, we then redefine our cost function as follows:

$$f(\psi) = \frac{\langle\psi|\, H\, |\psi\rangle}{\langle\psi|\psi\rangle} = \frac{g(\psi)}{h(\psi)} \tag{7}$$

Which will then transform the gradient expression using the quotient rule to:

$$\partial_i f(\psi) = \frac{\partial_i g(\psi) h(\psi) - g(\psi) \partial_i h(\psi)}{h(\psi)^2} \tag{8}$$

Where we used the short-hand notation ∂_i to refer to the partial derivative with respect to $A^{[i]}$. The explanations from this section can be applied to compute $\partial_i g(\psi)$ and $\partial_i h(\psi)$ individually, and then combine them to obtain the complete expression in equation 8.

Notes:

- The substraction in equation 8 is well defined since both $\partial_i g(\psi)$ and $\partial_i h(\psi)$ are rank 3 tensors with the same dimensions - equal to the dimensions of $A^{[i]}$.
- Both $g(\psi)$ and $h(\psi)$ evaluate to a scalar. Its multiplication with a tensor is defined as multiplying the scalar with all the elements of the tensor (i.e. elementwise).

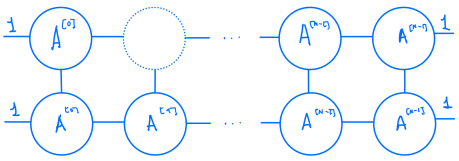

FIG. 3: Diagramatic representation of $\partial \langle\psi|\psi\rangle / \partial A^{[i]}$ with $i = 1$.

II. IMPLEMENTATION

In the current section I first discuss some technical details related to the implementation of the GDS algorithm, then I demonstrate its performance for different parameter regimes, and end with a comparison against other well known ground-state algorithms such as TEBD and DMRG.

Update

Combining the results obtained in the previous two subsections, we can then write the update step of the GDS algorithm as:

$$\begin{bmatrix} A^{[0]n+1} \\ A^{[1]n+1} \\ \vdots \\ A^{[N-1]n+1} \end{bmatrix} = \begin{bmatrix} A^{[0]n} \\ A^{[1]n} \\ \vdots \\ A^{[N-1]n} \end{bmatrix} - \eta \begin{bmatrix} \partial_0 f(\psi)^n \\ \partial_1 f(\psi)^n \\ \vdots \\ \partial_{N-1} f(\psi)^n \end{bmatrix} \tag{9}$$

Good Ansatz

When using the GDS algorithm to find ground states, one must be particularly mindful of the initial state (ansatz), as this can play an important role in the success of the algorithm. From equation 9, we see that the update step involves only a change in the entries of the individual tensors making up the MPS, and it does not update their bond dimensions. This means that in order to obtain an accurate solution, we must ensure that the bonds of our ansatz can appropriately depict the actual ground state.

For a general state to be exactly represented in MPS form, we can make use of the TT-SVD algorithm as explained in [6]. By focusing on the ℓ bond with dimension D_ℓ as shown in figure 4, we can split the tensor train in two. Then, transforming the tensors $T_{\leq \ell-1}$ and $T_{\geq \ell}$ into matrices, and looking at their ranks, we obtain the following relation [6]:

$$D_\ell = min\left(\Pi_{j=0}^{\ell} d_j, \Pi_{j=N-1}^{\ell} d_j\right) = min\left(2^\ell, 2^{N-\ell}\right) \tag{10}$$

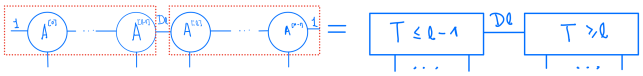

FIG. 4: Splitting of tensor resulting from TT-SVD algorithm into the tensors before bond ℓ ($T_{\geq \ell-1}$) and the ones after ($T_{\geq \ell}$)

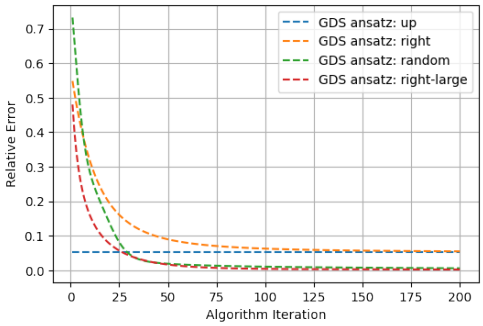

FIG. 5: Comparison of the GDS algorithm using different ansatz for L=8, J=1, and g=2.0

Where the second equality results from having equal local dimension ($d = 2$) in all sites. From this we see that, to represent any state of an N-sites system as an MPS, the bond dimensions will be at most:

$$\begin{aligned} D = &[1, 2, ..., 2^{N/2}, ...2, 1] (\text{N even}) \\ &[1, 2, ..., 2^{(N-1)/2}, 2^{(N-1)/2}, ...2, 1] (\text{N odd}) \end{aligned} \tag{11}$$

In addition, the initial values of the ansatz also play a role in the converge rate of GDS. In the TEBD and DMRG algorithms we usually start with an "all-spins-up" ansatz. For these algorithms, both the values and bond dimensions change quickly, thus, even with a relatively simple initial state we can achieve accurate results after a few iterations. Figure 5 shows the comparison of the GDS algorithm using different ansatz, starting from the simplest "all-spins-up", then an "all-spins-right" state, to the more general "random" and "right-large" states. The two latter use the bond dimensions as explained in 11, but the "random" tensors are filled randomly according to a gaussian distribution and "right-large" emulates a right spin with all the remaining entries equal to zero.

From this comparison, we observe that the simulations using the two product states ansatz (virtual bonds all equal 1) start with a lower error than the "random" ansatz, but their performance is bounded by their size, and they both end up converging to the same wrong state. On the other hand, both "random" and "right-large" tensors achieve a low error, but the second converges faster in most of the cases.

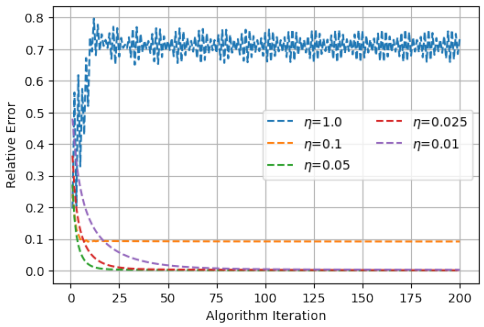

FIG. 6: Comparison of the GDS algorithm using different learning rates for L=8, J=1, and g=2.0

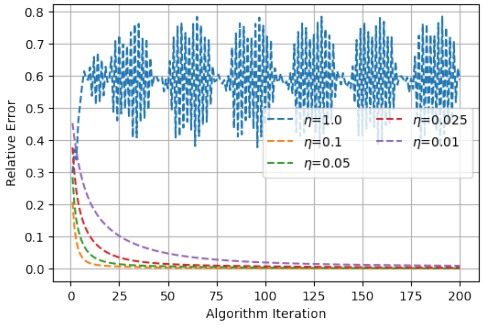

FIG. 7: Comparison of the GDS algorithm using different learning rates for L=5, J=1, and g=1.5

Learning Rate

An additional parameter in GDS that is not present in other algorithms is the learning rate η. Figure 6 shows a comparison of the GDS performance when varying the learning rate, and keeping the rest of the parameters fixed.

It is direct to see that varying the learning rate changes the convergence rate of GDS, however, for some cases (such as $\eta = 1.0$ in fig 6) it can also determine whether the algorithm will even find the correct minima. Moreover, it is important to notice that there is no "one-size-fits-all" learning rate value, as this depends on the size of the system, as well as the specific hamiltonian under study. To illustrate this, fig 7 shows a similar comparison, but now for a smaller system size and lower g. One can extrapolate that having a smaller learning rate might be more beneficial when dealing with larger sytems.

Iterations

The size of the system under study is also a determinant factor to take into account for our simulations. Figure 8 shows a comparison of the number of iterations necessary for the GDS algorithm to convergence for different system sizes. Here, we determine the algorithm has "converged" if the relative error is smaller than a given tolerance (set to 10^{-5}).

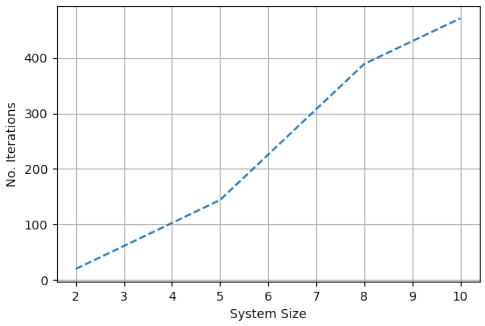

FIG. 8: Comparison of the GDS algorithm convergence for different system sizes. The learning rates were changed for each simulation according to the optimal value for each size. Hamiltonian parameters: J=1 and g=1.5

Different Parameters

The previous subsections were useful to determine the dependence of the GDS algorithm on the ansatz and learning rates, which are inherent parameters of this specific algorithm. Based on these comparisons, I decided to run the simulations with a "right-large" ansatz and a learning rate $\eta = 0.05$, as this is the rate that achieves accurate enough results for the system sizes under consideration.

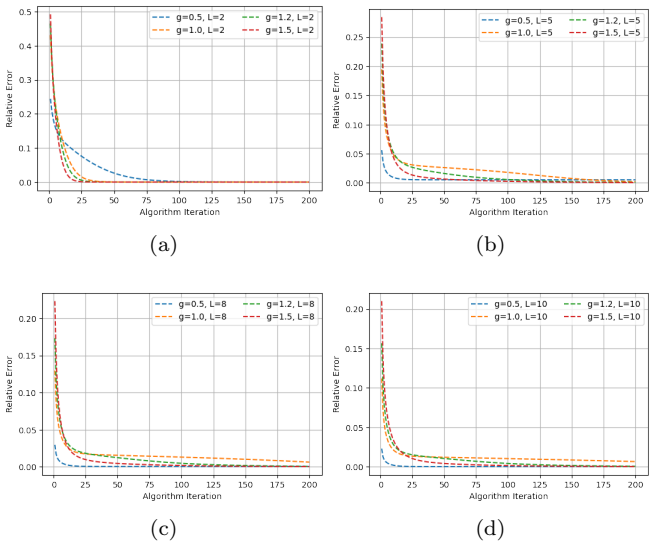

FIG. 9: GDS performance for different set of parameters. (a) Varying g for size L=2. (b)Varying g for size L=5. (c) Varying g for size L=8. (d) Varying g for size L=10.

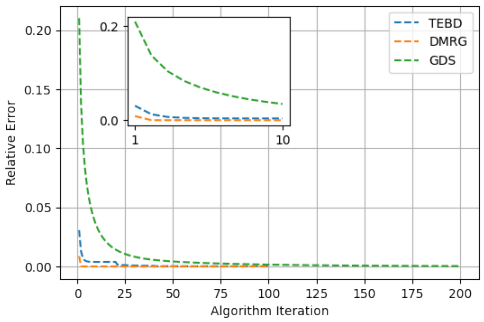

FIG. 10: Comparison of the GDS algorithm against the TEBD and DMRG. Sytem size $L = 10$, learning rate $\eta = 0.05$ and parameters $J = 1$ and $g = 1.5$.

Besides the expected fact that the simulation gets better for smaller system sizes, it can also be observed on figure 9 that better results are achieved for g farther from 1, i.e. g far from the critical value.

Comparison

Finally, I test the algorithm in a parameter regime where the GDS performs best, and tune the GDS parameters according to the aforementioned observations. On this light, figure 10 shows a performance comparison between the GDS, TEBD, and DMRG algorithms. This is done for a system size $L = 10$, learning rate $\eta = 0.05$ and parameters $J = 1$ and $g = 1.5$.

This comparison in the performance of the three algorithms demonstrates how fast the DMRG algorithm converges, followed closely by the TEBD. Although the GDS algorithm takes several more iterations to converge, we can see from the graph that eventually the algorithm achieves the desired relative error. The difference, however, in the iterations necessary for DMRG and GDS is close to a **factor of 10!**. Without improvements in the performance of GDS, such as second order gradient methods [7], it is clear that DMRG is the best of the three algorithms for finding ground states of the Ising Hamiltonian.

Additional Comments

- In order for the comparison between the algorithms to be fair, the iterations for TEBD are defined as $N_{steps}/len(dts) = Iterations$, where N_{steps} are the imaginary time steps performed with each of the elements on the array of dts: dts (see Appendix C).

- Similarly, the iterations for DMRG are defined as $2 \times sweep = Iterations$, where each sweep of the algorithm includes updating the MPS tensors from left to right and right to left, hence the factor of 2.

- While DMRG and GDS share similar methods to calculate their effective operators on the updating step (at least in my implementation, see Appendix F), one of the biggest differences between them is the way the MPS tensors are updated. In the former, the update is done locally two sites at a time, and proceeding to the next site. On the other hand, GDS updates all tensors simultaneously in an attempt to achieve the greatest descent in the rayleigh quotient, in what can be seen as a "greedy" optimization. Initially, this approach might seem more efficient, but turns out to incur in several complications.

- Among the complications referenced in the previous point, one of the first ones I encountered was the gradient being zero. This is a well known problem of GDS as the update step depends on the gradient, which can be zero for local minima or saddle points, without necessarily being in the global minima of our cost function. As a remedy, we can employ a stochastic gradient descent method (SGD), whose noise helps to escape the local minima [8].

- On a similar manner, we could encounter the so called "Barren plateaus" problem, which shows how for reasonably parametrized quantum states, the gradient in every direction is zero up to some fixed precision [9]. This is what I reckon happened when looking at the "all-spins-up" ansatz on figure 5.

A. Citations

[1] J. Hauschild and F. Pollmann, Efficient numerical simulations with tensor networks: Tensor network python (TeNPy), SciPost Physics Lecture Notes 10.21468/scipostphyslectnotes.5 (2018).

[2] A. M. Dalzell and F. G. S. L. Brandão, Locally accurate MPS approximations for ground states of one-dimensional gapped local Hamiltonians, Quantum **3**, 187 (2019).

[3] Wikipedia contributors, Gradient — Wikipedia, the free

encyclopedia (2022), [Online; accessed 3-August-2022].
[4] N. A. . D. Learning, Gradient descent and autograd (2022).
[5] Wikipedia contributors, Cauchy–riemann equations — Wikipedia, the free encyclopedia (2022), [Online; accessed 3-August-2022].
[6] C. Mendl, Lecture notes in tensor networks (2022).
[7] H. H. Tan and K. H. Lim, Review of second-order optimization techniques in artificial neural networks backpropagation, IOP Conference Series: Materials Science and Engineering **495**, 012003 (2019).
[8] Wikipedia contributors, Stochastic gradient descent — Wikipedia, the free encyclopedia (2022), [Online; accessed 4-August-2022].
[9] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven, Barren plateaus in quantum neural network training landscapes, Nature Communications **9**, 4812 (2018).

Appendix A: MPS and MPO

Creating MPS class for states and Ising Hamiltonian as MPO, including functions to convert to full matrix representation.

```
"""Toy code implementing a matrix product state."""

import numpy as np
from scipy.linalg import svd


class MPS:
    """Class for a matrix product state.

    We index sites with `i` from 0 to L-1; bond `i` is left of site `i`.
    We *assume* that the state is in right-canonical form.

    Parameters
    ----------
    Bs, Ss:
        Same as attributes.

    Attributes
    ----------
    Bs : list of np.Array[ndim=3]
        The 'matrices' in right-canonical form, one for each physical site.
        Each `B[i]` has legs (virtual left, physical, virtual right), in short ``vL i vR``
    Ss : list of np.Array[ndim=1]
        The Schmidt values at each of the bonds, ``Ss[i]`` is left of ``Bs[i]``.
    L : int
        Number of sites.
    """

    def __init__(self, Bs, Ss):
        self.Bs = Bs
        self.Ss = Ss
        self.L = len(Bs)

    def copy(self):
        return MPS([B.copy() for B in self.Bs], [S.copy() for S in self.Ss])

    def get_theta1(self, i):
        """Calculate effective single-site wave function on sites i in mixed canonical form.

        The returned array has legs ``vL, i, vR`` (as one of the Bs)."""
        return np.tensordot(np.diag(self.Ss[i]), self.Bs[i], [1, 0])  # vL [vL'], [vL] i vR

    def get_theta2(self, i):
        """Calculate effective two-site wave function on sites i,j=(i+1) in mixed canonical
form.

        The returned array has legs ``vL, i, j, vR``."""
        j = i + 1
        return np.tensordot(self.get_theta1(i), self.Bs[j], [2, 0])  # vL i [vR], [vL] j vR
```

```
50
51          def get_chi(self):
52              """Return bond dimensions."""
53              return [self.Bs[i].shape[2] for i in range(self.L - 1)]
54
55          def site_expectation_value(self, op):
56              """Calculate expectation values of a local operator at each site."""
57              result = []
58              for i in range(self.L):
59                  theta = self.get_theta1(i)  # vL i vR
60                  op_theta = np.tensordot(op, theta, axes=[1, 1])  # i [i*], vL [i] vR
61                  result.append(np.tensordot(theta.conj(), op_theta, [[0, 1, 2], [1, 0, 2]]))
62                  # [vL*] [i*] [vR*], [i] [vL] [vR]
63              return np.real_if_close(result)
64
65          def bond_expectation_value(self, op):
66              """Calculate expectation values of a local operator at each bond."""
67              result = []
68              for i in range(self.L - 1):
69                  theta = self.get_theta2(i)  # vL i j vR
70                  op_theta = np.tensordot(op[i], theta, axes=[[2, 3], [1, 2]])
71                  # i j [i*] [j*], vL [i] [j] vR
72                  result.append(np.tensordot(theta.conj(), op_theta, [[0, 1, 2, 3], [2, 0, 1, 3]]))
73                  # [vL*] [i*] [j*] [vR*], [i] [j] [vL] [vR]
74              return np.real_if_close(result)
75
76          def entanglement_entropy(self):
77              """Return the (von-Neumann) entanglement entropy for a bipartition at any of the bonds.
    """
78              result = []
79              for i in range(1, self.L):
80                  S = self.Ss[i].copy()
81                  S[S < 1.e-20] = 0.  # 0*log(0) should give 0; avoid warning or NaN.
82                  S2 = S * S
83                  assert abs(np.linalg.norm(S) - 1.) < 1.e-14
84                  result.append(-np.sum(S2 * np.log(S2)))
85              return np.array(result)
86
87
88      def init_spinup_MPS(L):
89          """Return a product state with all spins up as an MPS"""
90          B = np.zeros([1, 2, 1], np.float)
91          B[0, 0, 0] = 1.
92          S = np.ones([1], np.float)
93          Bs = [B.copy() for i in range(L)]
94          Ss = [S.copy() for i in range(L)]
95          return MPS(Bs, Ss)
96
97
98      def split_truncate_theta(theta, chi_max, eps):
99          """Split and truncate a two-site wave function in mixed canonical form.
100
101         Split a two-site wave function as follows::
102             vL --(theta)-- vR     =>    vL --(A)--diag(S)--(B)-- vR
103                     |   |                       |             |
104                     i   j                       i             j
105
106         Afterwards, truncate in the new leg (labeled ``vC``).
107
108         Parameters
109         ----------
110         theta : np.Array[ndim=4]
111             Two-site wave function in mixed canonical form, with legs ``vL, i, j, vR``.
112         chi_max : int
113             Maximum number of singular values to keep
114         eps : float
115             Discard any singular values smaller than that.
116
117         Returns
118         -------
```

```
        A : np.Array[ndim=3]
            Left-canonical matrix on site i, with legs ``vL, i, vC``
        S : np.Array[ndim=1]
            Singular/Schmidt values.
        B : np.Array[ndim=3]
            Right-canonical matrix on site j, with legs ``vC, j, vR``
        """
        chivL, dL, dR, chivR = theta.shape
        theta = np.reshape(theta, [chivL * dL, dR * chivR])
        X, Y, Z = svd(theta, full_matrices=False)
        # truncate
        chivC = min(chi_max, np.sum(Y > eps))
        assert chivC >= 1
        piv = np.argsort(Y)[::-1][:chivC]  # keep the largest `chivC` singular values
        X, Y, Z = X[:, piv], Y[piv], Z[piv, :]
        # renormalize
        S = Y / np.linalg.norm(Y)  # == Y/sqrt(sum(Y**2))
        # split legs of X and Z
        A = np.reshape(X, [chivL, dL, chivC])
        B = np.reshape(Z, [chivC, dR, chivR])
        return A, S, B


    class TFIModel:
        """Class generating the Hamiltonian of the transverse-field Ising model.

        The Hamiltonian reads
        .. math ::
            H = - J \\sum_{i} \\sigma^x_i \\sigma^x_{i+1} - g \\sum_{i} \\sigma^z_i

        Parameters
        ----------
        L : int
            Number of sites.
        J, g : float
            Coupling parameters of the above defined Hamiltonian.

        Attributes
        ----------
        L : int
            Number of sites.
        d : int
            Local dimension (=2 for spin-1/2 of the transverse field ising model)
        sigmax, sigmay, sigmaz, id :
            Local operators, namely the Pauli matrices and identity.
        H_bonds : list of np.Array[ndim=4]
            The Hamiltonian written in terms of local 2-site operators, ``H = sum_i H_bonds[i]``.
            Each ``H_bonds[i]`` has (physical) legs (i out, (i+1) out, i in, (i+1) in),
            in short ``i j i* j*``.
        """

        def __init__(self, L, J, g):
            self.L, self.d = L, 2
            self.J, self.g = J, g
            self.sigmax = np.array([[0., 1.], [1., 0.]])
            self.sigmay = np.array([[0., -1j], [1j, 0.]])
            self.sigmaz = np.array([[1., 0.], [0., -1.]])
            self.id = np.eye(2)
            self.init_H_bonds()

        def init_H_bonds(self):
            """Initialize `H_bonds` hamiltonian. Called by __init__()."""
            sx, sz, id = self.sigmax, self.sigmaz, self.id
            d = self.d
            H_list = []
            for i in range(self.L - 1):
                gL = gR = 0.5 * self.g
                if i == 0: # first bond
                    gL = self.g
                if i + 1 == self.L - 1: # last bond
```

```
                    gR = self.g
                H_bond = -self.J * np.kron(sx, sx) - gL * np.kron(sz, id) - gR * np.kron(id, sz)
                # H_bond has legs ``i, j, i*, j*``
                H_list.append(np.reshape(H_bond, [d, d, d, d]))
            self.H_bonds = H_list

        def energy(self, psi):
            """Evaluate energy E = <psi|H|psi> for the given MPS."""
            assert psi.L == self.L
            return np.sum(psi.bond_expectation_value(self.H_bonds))

    class TFIModel(TFIModel):
        """Extension of the TFIModel to define the MPO as well."""
        def __init__(self, L, J, g):
            super().__init__(L, J, g)  # calls the __init__() of b_model.TFIModel

            self.H_mpo = self.init_H_mpo()

        def init_H_mpo(self):
            """Initialize `H_mpo` Hamiltonian. Called by __init__().

            vL, vR, i, j (left, right, up, down)
            """
            self.zero = np.zeros([2,2])

            self.W = np.array([[self.id, self.sigmax, -self.g*self.sigmaz],
                               [self.zero, self.zero, -self.J*self.sigmax],
                               [self.zero, self.zero, self.id]])

            vR = np.array([[0,0,1]]).T #shape: 3,1 (out, in)
            vL = np.array([[1,0,0]]) #shape: 1,3 (out, in)

            W0 = np.tensordot(vL, self.W, axes=(1,0)) #out (in), (vL) vR i j-> out vR i j
            WLm1 = np.tensordot(self.W, vR, axes=(1,0)) #vL (vR) i j, (out) in -> vL i j in
            WLm1 = np.transpose(WLm1, axes=(0,3,1,2)) # vL in i j

            return [W0 if i==0 else self.W if i < self.L-1 else WLm1 for i in range(self.L)]


    def merge_mpo_tensor_pair(A0, A1):
        """
        Merge two neighboring MPO tensors.
        """
        A = np.tensordot(A0, A1, (1, 0)) #A0L (A0R) A0U A0D, (A1L) A1R A1U A1D -> A0L A0U A0D A1R
    A1U A1D
        # pair original physical dimensions of A0 and A1
        A = np.transpose(A, (0, 3, 1, 4, 2, 5)) # -> A0L A1R (A0U x A1U) (A0D x A1D)
        # combine original physical dimensions
        A = A.reshape((A.shape[0], A.shape[1], A.shape[2]*A.shape[3], A.shape[4]*A.shape[5]))
        return A

    def as_matrix(MPO_list):
        """Merge all tensors to obtain the matrix representation on the full Hilbert space."""
        op = MPO_list[0]
        for i in range(1, len(MPO_list)):
            op = merge_mpo_tensor_pair(op, MPO_list[i])
        assert op.ndim == 4
        # contract leftmost and rightmost virtual bond (has no influence if these virtual bond
    dimensions are 1)
        op = np.trace(op, axis1=0, axis2=1)
        return op
```

Appendix B: Exact diagonalization

Method that provides the exact ground state energies for the Ising model. Just as used in the lecture.

```

    """Provides exact ground state energies for the transverse field ising model for comparison.
```

```
    The Hamiltonian reads
    .. math ::
        H = - J \\sum_{i} \\sigma^x_i \\sigma^x_{i+1} - g \\sum_{i} \\sigma^z_i
    """
    import numpy as np
    import scipy.sparse as sparse
    import scipy.sparse.linalg.eigen.arpack as arp
    import warnings
    import scipy.integrate


    def finite_gs_energy(L, J, g):
        """For comparison: obtain ground state energy from exact diagonalization.

        Exponentially expensive in L, only works for small enough `L` <~ 20.
        """
        if L >= 20:
            warnings.warn("Large L: Exact diagonalization might take a long time!")
        # get single site operaors
        sx = sparse.csr_matrix(np.array([[0., 1.], [1., 0.]]))
        sz = sparse.csr_matrix(np.array([[1., 0.], [0., -1.]]))
        id = sparse.csr_matrix(np.eye(2))
        sx_list = []  # sx_list[i] = kron([id, id, ..., id, sx, id, .... id])
        sz_list = []
        for i_site in range(L):
            x_ops = [id] * L
            z_ops = [id] * L
            x_ops[i_site] = sx
            z_ops[i_site] = sz
            X = x_ops[0]
            Z = z_ops[0]
            for j in range(1, L):
                X = sparse.kron(X, x_ops[j], 'csr')
                Z = sparse.kron(Z, z_ops[j], 'csr')
            sx_list.append(X)
            sz_list.append(Z)
        H_xx = sparse.csr_matrix((2**L, 2**L))
        H_z = sparse.csr_matrix((2**L, 2**L))
        for i in range(L - 1):
            H_xx = H_xx + sx_list[i] * sx_list[(i + 1) % L]
        for i in range(L):
            H_z = H_z + sz_list[i]
        H = -J * H_xx - g * H_z
        E, V = scipy.sparse.linalg.eigsh(H, k=1, which='SA', return_eigenvectors=True)
        return E[0]
```

Appendix C: TEBD and DMRG

Modified Implementation of the DMRG and TEBD algorithm for its implementation with the correct number of iterations and returning the energy errors at each step. Includes the envolope DMRG method used to input parameters and obtain the relative errors array.

```
    """Toy code implementing the time evolving block decimation (TEBD)."""

    import numpy as np
    from scipy.linalg import expm
    from a_mps import split_truncate_theta
    import tfi_exact


    def calc_U_bonds(model, dt):
        """Given a model, calculate ``U_bonds[i] = expm(-dt*model.H_bonds[i])``.

        Each local operator has legs (i out, (i+1) out, i in, (i+1) in), in short ``i j i* j*``.
        Note that no imaginary 'i' is included, thus real `dt` means imaginary time evolution!
        """
```

```
        H_bonds = model.H_bonds
        d = H_bonds[0].shape[0]
        U_bonds = []
        for H in H_bonds:
            H = np.reshape(H, [d * d, d * d])
            U = expm(-dt * H)
            U_bonds.append(np.reshape(U, [d, d, d, d]))
        return U_bonds


    def run_TEBD(psi, U_bonds, model, E_exact, N_steps, chi_max, eps):
        """Evolve the state `psi` for `N_steps` time steps with (first order) TEBD.

        The state psi is modified in place."""
        Nbonds = psi.L - 1

        errors = []

        assert len(U_bonds) == Nbonds
        for n in range(N_steps): #this are the number of time steps perfomed for the dt that gave rise to U_bonds
            for k in [0, 1]:  # even, odd
                for i_bond in range(k, Nbonds, 2):
                    update_bond(psi, i_bond, U_bonds[i_bond], chi_max, eps)
            E = model.energy(psi)
            errors.append(abs((E - E_exact) / E_exact))

        assert len(errors) == N_steps
        return errors, E
        # done


    def update_bond(psi, i, U_bond, chi_max, eps):
        """Apply `U_bond` acting on i,j=(i+1) to `psi`."""
        j = i + 1
        # construct theta matrix
        theta = psi.get_theta2(i)  # vL i j vR
        # apply U
        Utheta = np.tensordot(U_bond, theta, axes=([2, 3], [1, 2]))  # i j [i*] [j*], vL [i] [j] vR
        Utheta = np.transpose(Utheta, [2, 0, 1, 3])  # vL i j vR
        # split and truncate
        Ai, Sj, Bj = split_truncate_theta(Utheta, chi_max, eps)
        # put back into MPS
        Gi = np.tensordot(np.diag(psi.Ss[i]**(-1)), Ai, axes=[1, 0])  # vL [vL*], [vL] i vC
        psi.Bs[i] = np.tensordot(Gi, np.diag(Sj), axes=[2, 0])  # vL i [vC], [vC] vC
        psi.Ss[j] = Sj  # vC
        psi.Bs[j] = Bj  # vC j vR


    def TEBD_gs_finite(L, J, g, E_exact='',iterations=100, dts_len = 5):

        """
        calculate the imaginary time evolution to find the gs of the hamiltonian given by
        L, J and g.
        dts_len: determines how many of the dt in dts list will be used
        iterations should be total number of iterations
        """
        assert 1<= dts_len <= 5, 'len of dts chosen should be between 1 and 5'

        if not E_exact:
            E_exact = tfi_exact.finite_gs_energy(L, J, g)
        print("finite TEBD, (imaginary time evolution)")
        print("L={L:d}, J={J:.1f}, g={g:.2f}".format(L=L, J=J, g=g))
        import a_mps
        import b_model
        model = b_model.TFIModel(L, J=J, g=g)
        psi = a_mps.init_spinup_MPS(L)
        errors = []
        dts = [0.1, 0.01, 0.001, 1.e-4, 1.e-5]
```

```
        N_steps = iterations//dts_len #how many steps for each of the dt will be used
        for dt in dts[:dts_len]:
            U_bonds = calc_U_bonds(model, dt)
            err, E = run_TEBD(psi, U_bonds, model, E_exact, N_steps=N_steps, chi_max=30, eps=1.e
    -10)
            errors += err
            print("dt = {dt:.5f}: E = {E:.13f}".format(dt=dt, E=E))
        print("final bond dimensions: ", psi.get_chi())
        # if L < 20:  # for small systems compare to exact diagonalization
        #     E_exact = tfi_exact.finite_gs_energy(L, 1., g)
        #     print("Exact diagonalization: E = {E:.13f}".format(E=E_exact))
        #     print("relative error: ", abs((E - E_exact) / E_exact))
        return errors, psi


    if __name__ == "__main__":
        TEBD_gs_finite(L=14, J=1., g=1.5)

    """Toy code implementing the density-matrix renormalization group (DMRG)."""

    import numpy as np
    from a_mps import split_truncate_theta
    import scipy.sparse
    import scipy.sparse.linalg.eigen.arpack as arp


    class HEffective(scipy.sparse.linalg.LinearOperator):
    """Class for the effective Hamiltonian.

    To be diagonalized in `DMRGEngine.update_bond`. Looks like this::

        .--vL*           vR*--.
        |       i*    j*      |
        |       |     |       |
        (LP)---(W1)--(W2)----(RP)
        |       |     |       |
        |       i     j       |
        .--vL             vR--.
    """

    def __init__(self, LP, RP, W1, W2):
        self.LP = LP  # vL wL* vL*
        self.RP = RP  # vR* wR* vR
        self.W1 = W1  # wL wC i i*
        self.W2 = W2  # wC wR j j*
        chi1, chi2 = LP.shape[0], RP.shape[2]
        d1, d2 = W1.shape[2], W2.shape[2]
        self.theta_shape = (chi1, d1, d2, chi2)  # vL i j vR
        self.shape = (chi1 * d1 * d2 * chi2, chi1 * d1 * d2 * chi2)
        self.dtype = W1.dtype

    def _matvec(self, theta):
        """calculate |theta'> = H_eff |theta>"""
        x = np.reshape(theta, self.theta_shape)  # vL i j vR
        x = np.tensordot(self.LP, x, axes=(2, 0))  # vL wL* [vL*], [vL] i j vR
        x = np.tensordot(x, self.W1, axes=([1, 2], [0, 3]))  # vL [wL*] [i] j vR, [wL] wC i [i*]
        x = np.tensordot(x, self.W2, axes=([3, 1], [0, 3]))  # vL [j] vR [wC] i, [wC] wR j [j*]
        x = np.tensordot(x, self.RP, axes=([1, 3], [0, 1]))  # vL [vR] i [wR] j, [vR*] [wR*] vR
        x = np.reshape(x, self.shape[0])
        return x


    class DMRGEngine(object):
        """DMRG algorithm, implemented as class holding the necessary data.

        Parameters
        ----------
        psi, model, chi_max, eps:
            See attributes

```

```
        Attributes
        ----------
        psi : MPS
            The current ground-state (approximation).
        model :
            The model of which the groundstate is to be calculated.
        chi_max, eps:
            Truncation parameters, see :func:`a_mps.split_truncate_theta`.
        LPs, RPs : list of np.Array[ndim=3]
            Left and right parts ("environments") of the effective Hamiltonian.
            ``LPs[i]`` is the contraction of all parts left of site `i` in the network ``<psi|H|psi>``,
            and similar ``RPs[i]`` for all parts right of site `i`.
            Each ``LPs[i]`` has legs ``vL wL* vL*``, ``RPS[i]`` has legs ``vR* wR* vR``
        """

        def __init__(self, psi, model, E_exact,chi_max=100, eps=1.e-12):
            assert psi.L == model.L  # ensure compatibility
            self.model = model
            self.E_exact =  E_exact
            self.H_mpo = model.H_mpo
            self.psi = psi
            self.LPs = [None] * psi.L
            self.RPs = [None] * psi.L
            self.chi_max = chi_max
            self.eps = eps
            # initialize left and right environment
            D = self.H_mpo[0].shape[0]
            chi = psi.Bs[0].shape[0]
            LP = np.zeros([chi, D, chi], dtype="float")  # vL wL* vL*
            RP = np.zeros([chi, D, chi], dtype="float")  # vR* wR* vR
            LP[:, 0, :] = np.eye(chi)
            RP[:, D - 1, :] = np.eye(chi)
            self.LPs[0] = LP
            self.RPs[-1] = RP
            # initialize necessary RPs
            for i in range(psi.L - 1, 1, -1):
                self.update_RP(i)

        def sweep(self):
            errors = []
            # sweep from left to right
            for i in range(self.psi.L - 2):
                self.update_bond(i)
            E = self.model.energy(self.psi)
            errors.append(abs((E - self.E_exact) / self.E_exact))

            # sweep from right to left
            for i in range(self.psi.L - 2, 0, -1):
                self.update_bond(i)

            E = self.model.energy(self.psi)
            errors.append(abs((E - self.E_exact) / self.E_exact))

            assert len(errors) == 2
            return errors, E

        def update_bond(self, i):
            j = i + 1
            # get effective Hamiltonian
            Heff = HEffective(self.LPs[i], self.RPs[j], self.H_mpo[i], self.H_mpo[j])
            # Diagonalize Heff, find ground state `theta`
            theta0 = np.reshape(self.psi.get_theta2(i), [Heff.shape[0]])  # initial guess
            e, v = arp.eigsh(Heff, k=1, which='SA', return_eigenvectors=True, v0=theta0)
            theta = np.reshape(v[:, 0], Heff.theta_shape)
            # split and truncate
            Ai, Sj, Bj = split_truncate_theta(theta, self.chi_max, self.eps)
            # put back into MPS
            Gi = np.tensordot(np.diag(self.psi.Ss[i]**(-1)), Ai, axes=[1, 0])  # vL [vL*], [vL] i
vC
```

```
            self.psi.Bs[i] = np.tensordot(Gi, np.diag(Sj), axes=[2, 0])  # vL i [vC], [vC*] vC
            self.psi.Ss[j] = Sj  # vC
            self.psi.Bs[j] = Bj  # vC j vR
            self.update_LP(i)
            self.update_RP(j)

        def update_RP(self, i):
            """Calculate RP right of site `i-1` from RP right of site `i`."""
            j = i - 1
            RP = self.RPs[i]  # vR* wR* vR
            B = self.psi.Bs[i]  # vL i vR
            Bc = B.conj()  # vL* i* vR*
            W = self.H_mpo[i]  # wL wR i i*
            RP = np.tensordot(B, RP, axes=[2, 0])  # vL i [vR], [vR*] wR* vR
            RP = np.tensordot(RP, W, axes=[[1, 2], [3, 1]])  # vL [i] [wR*] vR, wL [wR] i [i*]
            RP = np.tensordot(RP, Bc, axes=[[1, 3], [2, 1]])  # vL [vR] wL [i], vL* [i*] [vR*]
            self.RPs[j] = RP  # vL wL vL* (== vR* wR* vR on site i-1)

        def update_LP(self, i):
            """Calculate LP left of site `i+1` from LP left of site `i`."""
            j = i + 1
            LP = self.LPs[i]  # vL wL vL*
            B = self.psi.Bs[i]  # vL i vR
            G = np.tensordot(B, np.diag(self.psi.Ss[j]**-1), axes=[2, 0])  # vL i [vR], [vR*] vR
            A = np.tensordot(np.diag(self.psi.Ss[i]), G, axes=[1, 0])  # vL [vL*], [vL] i vR
            Ac = A.conj()  # vL* i* vR*
            W = self.H_mpo[i]  # wL wR i i*
            LP = np.tensordot(LP, A, axes=[2, 0])  # vL wL* [vL*], [vL] i vR
            LP = np.tensordot(W, LP, axes=[[0, 3], [1, 2]])  # [wL] wR i [i*], vL [wL*] [i] vR
            LP = np.tensordot(Ac, LP, axes=[[0, 1], [2, 1]])  # [vL*] [i*] vR*, wR [i] [vL] vR
            self.LPs[j] = LP  # vR* wR vR (== vL wL* vL* on site i+1)

# DMRG

def DMRG(L,J,g,E_exact='',iterations=10):

    """
    DMRG algorithm to find the ground state of the the hamiltonian given by
    L, J and g.

    the number of iterations required is iterations = no_sweeps*2
    since a sweep goes from left to right
    """

    if not E_exact:
        E_exact = finite_gs_energy(L, J, g)

    model = TFIModel(L, J, g)
    # compare to exact result

    psi = init_spinup_MPS(model.L)
    eng = DMRGEngine(psi, model, E_exact, chi_max=30, eps=1.e-13)
    errors = []
    no_sweeps = iterations//2 #divided by 2
    for i in range(no_sweeps):
        err, E = eng.sweep()
        errors += err
        print("sweep {i:2d}: E = {E:.13f}, rel. error {err:.4e}".format(i=i + 1, E=E, err=err
[-1]))
    print("final bond dimensions: ", psi.get_chi())

    return errors, psi
```

Appendix D: Helping functions: Left-Orthonormalization

Helping functions used to change a general MPS into its left-orthonormal form. The function will return the norm which would ideally be close to 1.

```
    def single_mode_product(A, T, j):
        """
        Compute the j-mode product between the matrix `A` and tensor `T`.
        """
        T = np.tensordot(A, T, axes=(1, j))
        # original j-th dimension is now 0-th dimension; move back to j-th place
        T = np.transpose(T, list(range(1, j + 1)) + [0] + list(range(j + 1, T.ndim)))
        return T

    def mps_orthonormalize_left(Alist):
        """
        Left-orthonormalize a MPS using QR decompositions.
        The list of tensors in `Alist` are updated in-place.

        Returns the overall norm of the original MPS. (The updated MPS has norm 1.)

        we assume order of each Alist[i] is vl i vR
        """
        L = len(Alist) #number of sites

        Atemp = np.ones(shape=(1,1,1))
        Alist.append(Atemp)

        for l in range(L):
            Dlm1 = Alist[l].shape[0] #vL
            nl = Alist[l].shape[1] # i
            Dl = Alist[l].shape[2] # vR

            Alist[l] = np.reshape(Alist[l], newshape=(Dlm1*nl, Dl))

            q,r = np.linalg.qr(Alist[l], mode='reduced')
            Dl = q.shape[1]

            Alist[l] = np.reshape(q, newshape=(Dlm1, nl,Dl))
            Alist[l+1] = single_mode_product(r, Alist[l+1], 0) # contract with the first dimension
    Dl_0 of A[l+1]


        if Alist[-1] < 0: #multiply by -1 to make sure the norm is always positive
            Alist[-1] *= -1
            Alist[-2] *= -1

        norm = Alist.pop() #picks out the last element (norm) and returns it
        return norm
```

Appendix E: Helping functions: Ansatz initialization

Helping functions used for the different ansatz discussed in this paper, including "all-spins-right", "random", and "right-large". The option "all-spins-up" is included in the MPS module, appendix A.

```
    def init_spinright_MPS(L):
        """Return a product state with all spins up as an MPS"""
        B = np.zeros([1, 2, 1], dtype=np.float64)
        B[0, 0, 0] = 1./np.sqrt(2)
        B[0, 1, 0] = 1./np.sqrt(2)
        S = np.ones([1], dtype=np.float64)
        Bs = [B.copy() for i in range(L)]
        Ss = [S.copy() for i in range(L)]
        return MPS(Bs, Ss)

    def crandn(size):
        """
        Draw random samples from the standard complex normal (Gaussian) distribution.
        Use to generate random tensor filled with complex numbers.
        """
        # 1/sqrt(2) is a normalization factor
        return (np.random.normal(size=size) + 1j*np.random.normal(size=size)) / np.sqrt(2)

```

```
    def min_bonds(L):
        """
        Calculates the bond dimensions necessary to represent a state with length L exactly

        The bonds start from 1 andgrow up to 2**(L/2) for L even and 2**((L-1)/2) for L odd. (both bonds in the middle will have this dimension)
        Then they descend again back to 1: [1,2,...,2**(L//2),...,2,1]

        """
        bonds = [None for _ in range(L+1)]
        for i in range(L+1):
            bonds[i] = min(2**i,2**(L-i))
        return bonds

    def init_random_MPS(L):
        """Return a product state with random entries as an MPS"""

        d = 2 #local dimension for ising model

        D = min_bonds(L) #calculate the list of bond dimensions. len: L+1 (since we have an extra one at the right end)
        assert D[0] == D[-1] == 1

        Bs = [np.random.normal(size=(D[i],d,D[i+1])).astype(np.float64) / np.sqrt(d*D[i]*D[i+1]) for i in range(L)]

        S = np.ones([1], dtype=np.float64)
        Ss = [S.copy() for i in range(L)] #not really the real S values but just because it is needed. Should Erase it on the main class
        return MPS(Bs, Ss)

    def init_rightlarge_MPS(L):
        """Return a state as an MPS equivalent to the all right spins but with the right bond dimensions"""

        d = 2 #local dimension for ising model

        D = min_bonds(L) #calculate the list of bond dimensions. len: L+1 (since we have an extra one at the right end)
        assert D[0] == D[-1] == 1

        Bs = [np.zeros(shape=(D[i],d,D[i+1]),dtype=np.float64) for i in range(L)]

        for i in range(len(Bs)):
                Bs[i][0, 0, 0] = 1./np.sqrt(2)
                Bs[i][0, 1, 0] = 1./np.sqrt(2)

        S = np.ones([1], dtype=np.float64)
        Ss = [S.copy() for i in range(L)] #not really the real S values but just because it is needed. Should Erase it on the main class
        return MPS(Bs, Ss)
```

Appendix F: Helping functions: Gradient computation

Helping functions including all the sub-functions and contraction methods necessary to compute the total gradient as given in equation 8. This includes the computation of the derivatives $\partial_i g(\psi)$ and $\partial_i h(\psi)$, as well as functions to evaluate the expectation value and inner product, $g(\psi)$ and $h(\psi)$, repectively.

```

    def contract_left_block(A, W, L):
        """
        Contraction step from left to right, with a matrix product operator (MPO)
        sandwiched in between.

        To-be contracted tensor network::

        ---------           -----
        /         \         /     \
```

```
        |         0|---    ---|0 A 2|---
        |          |          \__1__/
        |          |             |
        |          |
        |          |           __|__
        |          |          /  2  \
        |    L    1|---    ---|0 W 1|---
        |          |          \__3__/
        |          |             |
        |          |
        |          |           __|__
        |          |          /  1  \
        |         2|---    ---|0 A*2|---
        \__________/          \_____/
        """
        assert A.ndim == 3
        assert W.ndim == 4
        assert L.ndim == 3
        # multiply with conjugated A tensor
        T = np.tensordot(L, A.conj(), axes=(2, 0)) # L0 L1 (L2), (A0*) A1* A2* -> L0 L1 A1* A2*

        # multiply with W tensor
        T = np.tensordot(W, T, axes=((0, 3), (1, 2))) # (W0) W1 W2 (W3), L0 (L1) (A1*) A2* -> W1 W2 L0 A2*

        # multiply with A tensor
        Lnext = np.tensordot(A, T, axes=((0, 1), (2, 1))) # (A0) (A1) A2, W1 (W2) (L0) A2* -> A2 W1 A2*

        return Lnext

    def contract_right_block(A, W, R):
        """
        Contraction step from right to left, with a matrix product operator
        sandwiched in between.

        To-be contracted tensor network::

              _____             _________
             /     \           /         \
           ---|0 A 2|---    ---|0        |
              \__1__/           |         |
                 |              |         |
                                |         |
              __|__             |         |
             /  2  \            |         |
           ---|0 W 1|---    ---|1   R    |
              \__3__/           |         |
                 |              |         |
                                |         |
              __|__             |         |
             /  1  \            |         |
           ---|0 A*2|---    ---|2        |
              \_____/           \_________/
        """

        assert A.ndim == 3
        assert W.ndim == 4
        assert R.ndim == 3


        # multiply with conjugated A tensor
        T = np.tensordot(R, A.conj(), axes=(2, 2)) # R0 R1 (R2), A0* A1* (A2*) -> R0 R1 A0* A1*
        # multiply with W tensor
        T = np.tensordot(W, T, axes=((1, 3), (1, 3))) # W0 (W1) W2 (W3), R0 (R1) A0* (A1*) -> W0 W2 R0 A0*
         # multiply with A tensor
        Rnext = np.tensordot(A, T, axes=((1, 2), (1, 2))) # A0 (A1) (A2), W0 (W2) (R0) A0* -> A0 W0 A0*

```

```
        return Rnext

        def contract_center_block(A, W):
        """
        Contraction step from right to left, with a matrix product operator
        sandwiched in between.

        To-be contracted tensor network:
              __|__
             /  2  \
          ---|0 W 1|---
             \__3__/
                |


              __|__
             /  1  \
          ---|0 A*2|---
             \_____/
        """

        assert A.ndim == 3
        assert W.ndim == 4
        # multiply W with conjugated A tensor
        C =  np.tensordot(W, A.conj(), axes=(3, 1)) # W0 W1 W2 (W3), A0* (A1*) A2* -> W0 W1 W2 A0* A2*
        return np.transpose(C, axes=(2,0,3,1,4))

    def construct_gradient_tensor_g(L,C,R):
        """
        construct the tensor of degree three

                          0       1        2
          _________       |       |        |     _________
         /         \      |       |        |    /         \
         |        0|---/   |       |        \---|0        |
         |         |       |       |            |         |
         |         |       |       |            |         |
         |         |       |       |            |         |
         |         |       |     __|__          |         |
         |         |       |    /  0  \         |         |
         |    L   1|---    ---|1   3|---    ---|1   R    |
         |         |       |     |          |    |         |
         |         |       |     |          |    |         |
         |         |       |  C  |          |    |         |
         |         |       |     |          |    |         |
         |         |       |     |          |    |         |
         |        2|---    ---|2   4|---    ---|2        |
         \_________/         \_____/         \_________/

        """
        return np.einsum(L,(0,1,2),C,(3,1,2,4,5),R,(6,4,5),(0,3,6)) # L0, C0, R0 = A0, A1, A2

    def construct_gradient_tensor_h(I,A,R):
        """
        construct the tensor of degree three

                          0       1        2
          _________       |       |        |     _________
         /         \      |       |        |    /         \
         |        0|---/   |       |        \---|0        |
         |         |       |       |            |         |
         |         |       |       |            |         |
         |     I   |       |       |            |    R    |
         |         |       |     __|__          |         |
         |         |       |    /  1  \         |         |
         |         |       |    |  A* |         |         |
         |        1|---    ---|0   2|---    ---|1        |
         \_________/         \_____/         \_________/

        I: identity resulting from our state being left normalized
```

```
        I0 = I1 = A*0
        """
        assert I.shape[0] == I.shape[1] == A.shape[0]

        return np.einsum(I,(0,1),A,(1,2,3),R,(4,3),(0,2,4)) # L0 A*0 R0 = A0 A1 A2

    def compute_right_operator_blocks(psi:MPS, H:TFIModel=None):
        """
        Compute all partial contractions starting from the right
        output is the list of rightblocks BR.

        input:
        -psi: MPS tensor form
        -H: MPO tensor form of the same length as psi

        if H is not given, the calculations are done for the gradient of h instead of g.
        Meaning that we are calculating only the inner product part, without any hamiltonian

        BR[-1] = BR[L-1] = identity but with dimension three/two in case H is not given
        BR[0] = block containing all contractions from A_(N-1) to A(1)

        in general: BR[i] contains all contractions up to A(i+1) (inclusive of it)
        """
        L = psi.L
        BR = [None for _ in range(L)]

        if H:
            assert L == H.L
            # initialize rightmost dummy block
            BR[-1] = np.array([[[1]]], dtype=psi.Bs[0].dtype) #shape: (1,1,1): A(N-1)[2] H(N-1)[1] A*(N-1)[2]
            for i in reversed(range(L-1)):
                BR[i] = contract_right_block(psi.Bs[i+1], H.H_mpo[i+1], BR[i+1])
            return BR
        else:
            # initialize rightmost dummy block
            BR[-1] = np.array([[1]], dtype=psi.Bs[0].dtype) #shape (1,1): A(N-1)[2] A*(N-1)[2]
            for i in reversed(range(L-1)):
                contraction = np.tensordot(psi.Bs[i+1], psi.Bs[i+1].conj(),axes=((1,1))) # A0 (A1) A2, A0* (A1)* A2* -> A0 A2 A0* A2*
                BR[i] = np.tensordot(contraction, BR[i+1],axes=((1,3),(0,1))) # A0 (A2) A0* (A2)*, (R0) (R1) -> A0 A0*
            return BR

    def compute_energy(psi:MPS, H:TFIModel) -> float:
        """
        computes the energy (expectation value) <psi|H|psi>
        input:
        - psi: MPS
        - H: TFI
        output:
        - energy: <psi|H|psi> (scalar)
        """

        BR = compute_right_operator_blocks(psi, H)
        BL = np.array([[[1.0]]], dtype=BR[0].dtype) #initialize with identity to close all open legs on left
        BL = contract_left_block(psi.Bs[0],H.H_mpo[0],BL)#include the A0 and M0 tensors

        return np.tensordot(BL,BR[0],axes=((0,1,2),(0,1,2)))

    def inner_product(psi:MPS) -> float:
        "Calculates the inner product <psi|psi> := norm(psi): scalar"
        Bs =  psi.Bs
        L = len(Bs)
        contr = np.ones((1,1)) # has indices (alpha_n*, alpha_n)
        for n in range(L):
            M_ket = Bs[n]  # has indices (alpha_n, j_n, alpha_{n+1})
            M_bra = Bs[n].conj()  # has indices (alpha_n*, j_n, alpha_{n+1}*)
            contr = np.tensordot(contr, M_ket , axes=(1, 0))
```

```
            # now contr has indices alpha_n*, j_n, alpha_{n+1}
            contr = np.tensordot(M_bra, contr, axes=([0, 1], [0, 1]))
        assert contr.shape == (1, 1)
        norm = contr.item()
        return norm

    def cost_function(psi:MPS,H:TFIModel) -> float:
        " definition of cost function used in this problem. Not really employed anywhere. Just as reference"
        return compute_energy(psi,H)/inner_product(psi)


    def compute_gradient(psi:MPS, H:TFIModel):
        """
        computes gradient of cost function evaluated at the current psi
        Works only for the cost function defined above, i.e. rayleigh quotient

        -cost function f(psi) = <psi|H|psi>/<psi|psi>: adding this to make sure that the gradient is calcualated wrt to normalized states

        input:
        -psi: MPS state from which to calculate the gradient.
        -H: MPO state of ising hamiltonian

        output:
        - grad: list of same size as psi.Bs with each tensor having the same dimensions as each Bs containing the d(f)/d(Bi) with Bi each of the tensors in the MPS

        recall:
        Each ‘B[i]‘ has legs (virtual left, physical, virtual right), in short ‘‘vL i vR‘‘
        Each  M[i] has legs  (vritual left, virtual right, up, down), in short  ’vL, vR, i, j’
        (virtual left, virtual right, local ket, local bra)

        L: will contain all the contractions to the left of the Bs[i] wrt which we are calculating the derivative
        R: same but to the right
        """

        L = psi.L #number of sites
        assert L == H.L

       # blocks for the gradient of g(x)
        BR_g = compute_right_operator_blocks(psi, H)
        BL_g = [None for _ in range(L)]
        BL_g[0] = np.array([[[1.0]]], dtype=BR_g[0].dtype) #identity of the first left block

        # blocks for the gradient of h(x)
        BR_h = compute_right_operator_blocks(psi)

        gradients = [None for _ in range(L)]

        for i in range(L):
            BC = contract_center_block(psi.Bs[i], H.H_mpo[i])
            gprime = construct_gradient_tensor_g(BL_g[i],BC,BR_g[i]) # A0 A1 A2

            I = np.eye(psi.Bs[i].shape[0])
            hprime = construct_gradient_tensor_h(I,psi.Bs[i],BR_h[i])

            h = inner_product(psi)
            g = compute_energy(psi, H)

            #print(’gprime: ’, gprime)
            #print(’hprime: ’, hprime)
            gradients[i] = ((gprime*h) - (hprime*g))/(h**2) #note: multiplying scalar to a tensor is equivalent to element wise multiplication

            if i!= L-1:
                BL_g[i+1] = contract_left_block(psi.Bs[i],H.H_mpo[i],BL_g[i])

        return gradients
```

Appendix G: Gradient Descent Algorithm

Main functions of the Gradient Descent algorithm including the update step and the envelop function where the ansatz, rate, iterations and Hamiltonian parameters are chosen.

```
    def update(psi_current:MPS, gradients:list, rate=1):

        """ Compute the updated psi based on gradient descendent method

        The Hamiltonian reads
        .. math ::
            H = - J \\sum_{i} \\sigma^x_i \\sigma^x_{i+1} - g \\sum_{i} \\sigma^z_i

        input:
        - rate: learning rate $\gamma$
        - psi_current: current values of variables w(t)
        - gradient: gradient of cost function f evaluated at current w(t), $\ nabla f(w(t))$

        output:
        - psi_next := w(t+1) =  w(t) - gamma * nabla f(w(t))

        I will be basically updating each of the tensors individual as psi.Bs[i] - rate*gradients[i
    ]

        """
        L = len(gradients)
        assert L == psi_current.L

        Bs_updated = [None for _ in range(L)]

        for i in range(L):
            Bs_updated[i] = psi_current.Bs[i] - rate*gradients[i]
        #print('Updated one: ',Bs_updated[0])
        mps_orthonormalize_left(Bs_updated)
        return MPS(Bs_updated, psi_current.Ss)

    # GDS optimization algorithm
    def GDS_optimization(L,J,g, rate=0.01, E_exact='', ansatz='right-large',iterations=100, tol='',
     convergence='series'):

        """
        Envelope function to run the GDS optimization

        inputs:

        - L, J, g: determine the system Hamiltonian
        - rate: learning rate for the GDS optimization
        - E_exact: defaults to None. Exact energy to be used for the error calculation
        - ansatz: initial state to be used. defaults to 'right-large'.
        - iterations: number of iterations to be performed if tol is not defined
        - tol: tolerance to be checked for the convergence methods
        - convergence:
            'series' (default) convergence means that two consecutive values in the energy series
    are close to each other
            'value', convergence means that the relative error is smaller than tolerance

        ouptuts:
        - errors: list of relative errors after each iteration
        - psi_next: last psi: MPS after the algorithm has ended
        - tot_iter: output only if tol is given. Shows the number of iterations the algorithm
    needed for convergence
        """

        if not E_exact:
            E_exact = finite_gs_energy(L, J, g)

        #initial gues
        if ansatz=='right-large':
            psi_current = init_rightlarge_MPS(L=L) #this state would usually be normalized
```

```
    elif ansatz=='random':
        psi_current = init_random_MPS(L=L) #need not be normalized
    elif ansatz=='right':
        psi_current = init_spinright_MPS(L=L) #should be normalized
    elif ansatz=='up':
        psi_current = init_spinup_MPS(L=L) #should be normalized

    print('initial norm:', inner_product(psi_current))
    mps_orthonormalize_left(psi_current.Bs) #ensures the norm is as close as possible to zero and leaves it in left-orthonormal form

    #model to solve: hamiltonian
    H = TFIModel(L=L,J=J,g=g)

    errors = []
    #checking the initial energy:
    print('initial energy:', compute_energy(psi_current,H))


    tot_iter = 0 #number of iterations needed to converge
    if tol: #do a while loop only if a tolerance is specified

        if convergence == 'series':
            Eold = 1 # this is the E_(i-1)here we are assuming Energy will not be close to zero, otherwise the algorithm will stop in the first iteration
            Enew = 0 #this is the E_i
            while abs(Enew - Eold) > tol:
                gradients = compute_gradient(psi=psi_current, H=H)

                psi_next = update(psi_current, gradients, rate=rate)

                Eold = Enew
                Enew = compute_energy(psi_next,H)
                errors.append(abs((Enew - E_exact) / E_exact))
                psi_current = psi_next
                tot_iter += 1
                if tot_iter == 300:
                    print('algorithm did not converge, reached number of iterations: 300')
                    break
        elif convergence == 'value':
            err = 1
            while err > tol:
                gradients = compute_gradient(psi=psi_current, H=H)

                psi_next = update(psi_current, gradients, rate=rate)
                E = compute_energy(psi_next,H)
                err = abs((E - E_exact) / E_exact)
                errors.append(err)
                psi_current = psi_next
                tot_iter += 1
                if tot_iter == 600:
                    print('algorithm did not converge, reached number of iterations: 600')
                    break
        else:
            raise 'not a valid convergence mode'
    else:
        for i in tqdm(range(iterations)):
            gradients = compute_gradient(psi=psi_current, H=H)
            #print('gradients:',gradients)
            #print('--------------------')
            psi_next = update(psi_current, gradients, rate=rate)
            #print('Bs 0:', psi_next.Bs[0])
            #print('Bs 1:', psi_next.Bs[1])
            #print('----------------------')
            #print(psi_next.Bs)
            #print('current norm is: ', inner_product(psi_next))
            E = compute_energy(psi_next,H)
            errors.append(abs((E - E_exact) / E_exact))
            psi_current = psi_next

```

```
130         print('final bond dimensions: ', [B.shape[0] for B in psi_next.Bs]+[1])
131         print('final energy:', compute_energy(psi_next,H))
132 
133         if tol:
134             return errors, psi_next, tot_iter
135         else:
136             return errors, psi_next
```