Gradient descent optimization of MPS for Ground state

Estêvão Gomes
Physik-Department E28, Technische Universität München, 85747 Garching, Germany
(Dated: July 16, 2025)

In this article, gradient descent is applied to optimize Matrix Product States for finding the ground state of the transverse-field Ising model, leveraging JAX for automatic differentiation. The results are compared with TEBD and DMRG, emphasizing convergence behavior and accuracy across various regimes. The study highlights both the strengths and limitations of gradient-based approaches for quantum many-body systems.

I. INTRODUCTION

The study of quantum many-body systems constitutes a fundamental pillar of modern condensed matter physics, where complex quantum phenomena emerge from the collective behavior of interacting particles. The primary obstacle in studying these systems computationally is the exponential scaling of the Hilbert space with system size, which results in inaccessibly large computational requirements. Tensor networks have become an essential tool for circumventing this exponential bottleneck, enabling efficient large-scale numerical simulations of quantum many-body systems, at the cost of using truncations that limit the entanglement content captured. [1]

To obtain the ground state of a large quantum many-body system, the usage of exact diagonalization methods is often impratical, thus iterative methods such as the Time-Evolving Block Decimation (TEBD) [2] and Density Matrix Renormalization Group (DMRG) [3] are preferred. In this article, direct gradient descent optimization is applied to the Matrix Product State (MPS) [4] representation of the ground state of a transverse field Ising model (TFI), making use of the automatic differentiation capabilities of JAX [5]. This approach is then compared with the results obtained from the TEBD and DMRG methods.

This article is structured as follows: In section II, we provide the theoretical background of the methods used. In section III, we present and discuss the results obtained from our simulations. Finally, we conclude in section IV.

II. THEORETICAL BACKGROUND

A. Matrix Product States

Matrix Product States provide a powerful representation of quantum many-body states, particularly efficient for systems that obey the area law for entanglement entropy, such as the 1D transverse field Ising model, discussed in detail in section II B.

An MPS is expressed as a product of tensor, where each tensor corresponds to a local degree of freedom (spin j_n). For open boundary conditions (OBC), the MPS is given by

$$|\Psi\rangle = \sum_{j_1,\cdots,j_N} M^{[1]j_1} M^{[2]j_2} \cdots M^{[N]j_N} |j_1 j_2 \dots j_N\rangle, \quad (1)$$

where each $M^{[n]j_n}$ tensor has shape (χ_{n-1}, d, χ_n), with d being the local dimension (e.g., $d = 2$ for spin-1/2 systems), and χ_n being the bond dimension at site n. The indices j_n run over the local degrees of freedom, and $\chi_0 = \chi_N = 1$ for OBC.

For periodic boundary conditions (PBC), the MPS is defined similarly, but with the additional condition that the first and last tensors are connected,

$$|\Psi\rangle = \sum_{j_1,\cdots,j_N} \mathrm{Tr}\left\{ M^{[1]j_1} M^{[2]j_2} \cdots M^{[N]j_N} \right\} |j_1 j_2 \dots j_N\rangle, \quad (2)$$

where $M^{[1]j_1}$ is connected to $M^{[N]j_N}$ through a cyclic permutation of the indices.

B. Transverse Field Ising model

The transverse field Ising model (TFI) is a fundamental model in quantum many-body physics, described for a spin-1/2 system by the Hamiltonian

$$H = -J \sum_{i=1}^{L} \sigma_i^x \sigma_{i+1}^x - g \sum_{i=1}^{L} \sigma_i^z, \quad (3)$$

where J is the coupling constant, g is the transverse field strength, and σ_i^x and σ_i^z are the Pauli matrices acting on the i-th spin. The model exhibits a quantum phase transition at a critical value of the transverse field $g_c = J$, where the system transitions from an ordered phase (for $g < g_c$) to a disordered phase (for $g > g_c$).

When using tensor networks, energy expectation values can be computed more efficiently by representing the Hamiltonian in the form of a Matrix Product Operator (MPO), which can be seen schematically in fig. 1.

In this representation, the tensor at each site, $W^{[n]}$, is defined as

$$W^{[n]} = \begin{bmatrix} \mathbb{I} & \sigma^x & -g\sigma^z \\ 0 & 0 & -J\sigma^z \\ 0 & 0 & \mathbb{I} \end{bmatrix}, \quad (4)$$

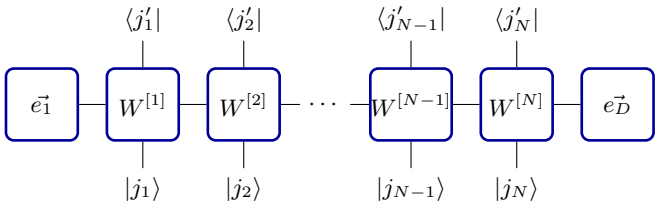

FIG. 1. Schematic representation of an MPO.

with $\mathbb{I}$ being the identity matrix, σ^x, σ^z the Pauli matrices, and the vectors $\vec{e_1}$ and $\vec{e_D}$ are defined as

$$\vec{e_1} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \tag{5}$$

$$\vec{e_D} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T. \tag{6}$$

C. Gradient Descent on MPS

To perform gradient descent on an MPS, one first needs to define the loss function to minimize. In this article, we aim to find the ground state of the TFI model, thus a suitable loss to minimize is the normalized energy expectation value,

$$\mathcal{L}(\psi) = \frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle}, \tag{7}$$

where the energy expectation value is calculated by contracting the MPS with the MPO representation of the Hamiltonian, as shown in fig. 2, and the normalization factor is computed by contracting the MPS with itself, as shown in fig. 3.

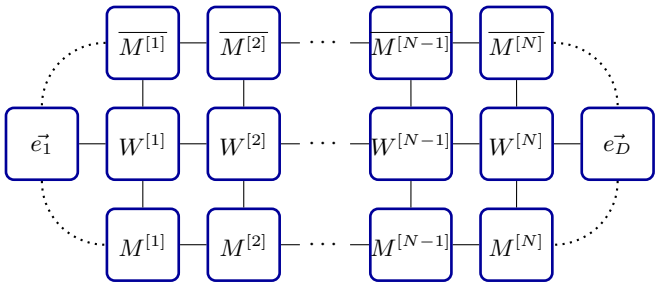

FIG. 2. Schematic representation of the energy expectation value, $\langle\psi|H|\psi\rangle$, of an MPS.

To update the MPS tensors $M^{[n]}$ during the optimization process, we compute the gradient of the loss function with respect to each tensor. This can be computed using JAX's automatic differentiation, or using the analytical expression. In section III A, we will compare the results obtained from both methods.

Analytically, the derivative of the loss function with respect to the tensor $M^{[n]}$ is given by

$$\partial_{M^{[n]}}\mathcal{L}(\psi) = \frac{\partial_{M^{[n]}}\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle} - \frac{\langle\psi|H|\psi\rangle\partial_{M^{[n]}}\langle\psi|\psi\rangle}{\langle\psi|\psi\rangle^2} \tag{8}$$

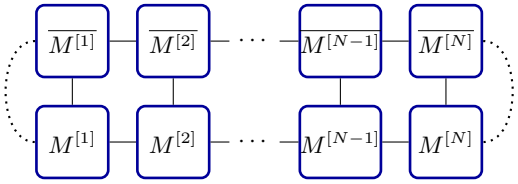

FIG. 3. Schematic representation of the squared norm, $\langle\psi|\psi\rangle$, of an MPS.

with $\partial_{M^{[n]}}$ denoting the derivative with respect to the tensor $M^{[n]}$. This expression can be further simplified by using

$$\partial_{M^{[n]}}\langle\psi|H|\psi\rangle = \langle\partial_{M^{[n]}}\psi|H|\psi\rangle + \langle\psi|H|\partial_{M^{[n]}}\psi\rangle \tag{9}$$

$$= 2\Re\left(\langle\psi|H|\partial_{M^{[n]}}\psi\rangle\right), \tag{10}$$

and

$$\partial_{M^{[n]}}\langle\psi|\psi\rangle = \langle\partial_{M^{[n]}}\psi|\psi\rangle + \langle\psi|\partial_{M^{[n]}}\psi\rangle \tag{11}$$

$$= 2\Re\left(\langle\psi|\partial_{M^{[n]}}\psi\rangle\right). \tag{12}$$

Equations (10) and (12) can be computed analytically using contractions similar to those used for the energy expectation value, in fig. 2, and normalization factor, in fig. 3, by ommiting the contraction with $M^{[n]}$. These can be visualized schematically in figs. 4 and 5, respectively.

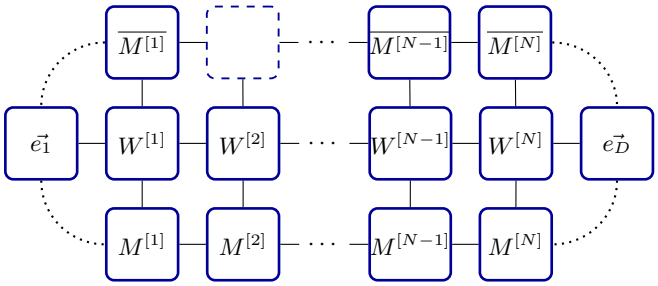

FIG. 4. Schematic representation of the derivative of the energy expectation value, $\partial_{M^{[2]}}\langle\psi|H|\psi\rangle/2$, of an MPS.

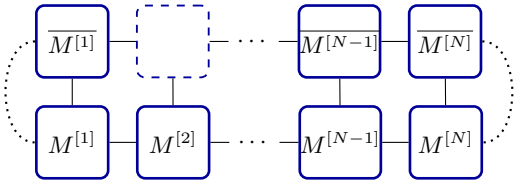

FIG. 5. Schematic representation of the derivative of the squared norm, $\partial_{M^{[2]}}\langle\psi|\psi\rangle/2$, of an MPS.

During the optimization process, we update the tensors $M^{[n]}$ using the ADAM optimizer [6], implemented in the OPTAX library [7], as it offers faster convergence compared to standard gradient descent by leveraging both the first and second moments of the gradients.

D. Time Evolving Block Decimation

The TEBD algorithm is used to obtain the time evolution of a quantum state,

$$|\psi(t)\rangle = U(t)|\psi(0)\rangle, \tag{13}$$

where $U(t) = e^{-itH}$ is the real time evolution operator and $U(\tau) = e^{-\tau H}$ the imaginary time evolution operator. The latter can be used to find the ground state of a quantum system via the relation

$$|\psi_{GS}\rangle = \lim_{\tau\to\infty} \frac{e^{-\tau H}|\psi_0\rangle}{||e^{-\tau H}|\psi_0\rangle||}. \tag{14}$$

Put simply, the TEBD algorithm starts by applying a Suzuki-Trotter decomposition, given in first and second order by

$$e^{(X+Y)\delta} = e^{X\delta}e^{Y\delta} + \mathcal{O}(\delta^2), \tag{15}$$
$$e^{(X+Y)\delta} = e^{X\delta/2}e^{Y\delta}e^{X\delta/2} + \mathcal{O}(\delta^3), \tag{16}$$

with δ a small parameter, to a Hamiltonian decomposed as a sum of two-site operators,

$$H = \sum_{n \text{ odd}} H_{n,n+1} + \sum_{n \text{ even}} H_{n,n+1}, \tag{17}$$

where the local Hamiltonian terms acting on the n-th and $(n+1)$-th sites, $H_{n,n+1}$, commute for either odd or even n.

This allows us to write the time evolution operator for a small timestep $\delta t \ll 1$ as a product of local operators acting on pairs of neighboring sites, given in first order by

$$e^{-\delta t H} = \prod_{n \text{ odd}} e^{-\delta t H_{n,n+1}} \prod_{n \text{ even}} e^{-\delta t H_{n,n+1}}. \tag{18}$$

The MPS is then updated by applying the local operators to the corresponding tensors, followed by a singular value decomposition to truncate the bond dimension and maintain computational efficiency.

E. Density Matrix Renormalization Group

DMRG is a variational algorithm that iteratively optimizes the MPS representation of a quantum state to minimize the energy expectation value.

A DMRG update consists of minimizing the energy by optimizing the tensors $M^{[n]}$ and $M^{[n+1]}$ in the MPS while keeping the remaining chain fixed. This is done by projecting the Hamiltonian into the reduced Hilbert space spanned by the basis $\{|\alpha_n\rangle \otimes |j_n\rangle \otimes |j_{n+1}\rangle \otimes |\alpha_{n+1}\rangle\}$, with $|\alpha_n\rangle$ and $|\alpha_{n+1}\rangle$ representing the left and right virtual bond states of the MPS, and $|j_n\rangle$, $|j_{n+1}\rangle$ are the physical site indices. The effective Hamiltonian in this subspace is then diagonalized using an iterative eigensolver such as the Lanczos algorithm [8] to obtain the lowest-energy state.

This two-site update is repeated sequentially for all pairs of neighboring sites in the MPS, starting from the leftmost pair and sweeping rightward to the end, then reversing direction and sweeping back to the left.

On that account, the DMRG algorithm can be seen as a "smart" version of gradient descent, as it performs exact diagonalization in a variational subspace, optimizing one (or two) tensors at a time. Unlike gradient descent, which updates all tensors simultaneously based on local gradient information, this local optimization guarantees that the energy never increases after each step.

III. RESULTS

In this section, we present the results obtained from the simulations of the TFI model using the methods described in section II. We then compare the results obtained from the gradient descent optimization of the MPS with those obtained from the TEBD and DMRG methods.

A. Analytical vs Automatic Differentiation

As a baseline consistency check, a comparison between the gradients computed using JAX's automatic differentiation and the analytical expression derived in section II C is performed. The mean error, its standard deviation and the maximum error are shown in fig. 6, as well as the average runtime of each gradient computation for the whole optimization.

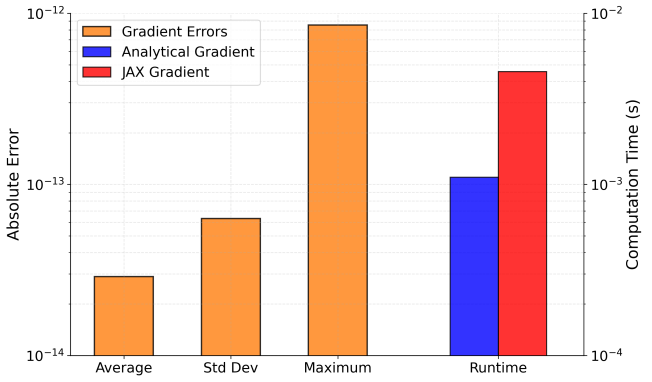

FIG. 6. Error and runtime comparison between analytical and automatic differentiation for the loss in eq. (7).

As expected, the analytical gradients differ from the automatic differentiation results by a value close to machine precision, demonstrating the accuracy of both computations. Moreover, the runtime of the analytical gradients is lower than that of the automatic differentiation,

making it a more efficient choice for large-scale optimizations due to its easy implementation. Nevertheless, the automatic differentiation approach is more flexible and easier to implement for complex models, giving it an edge in terms of usability.

For this project, automatic differentiation could also be used to compute the hessian of the loss, allowing for a faster convergence with the use of Newton's method.

B. Gradient Descent on MPS

To perform gradient descent, the MPS cannot be initialized as a low bond dimension product state such as the all-up state, $|\uparrow \cdots \uparrow\rangle$, as gradient descent cannot increase the bond dimension during optimization. However, simply starting with a high bond dimension product state is not sufficient either: if all tensors are identical, the gradient vanishes and no updates occur.

For this reason, in the remainder of the analysis of the transverse-field Ising model (TFI) with parameters $g > J$, the MPS is initialized as a high bond dimension all-up state with small Gaussian noise added to each tensor.

For the TFI in the regime $g < J$, a more appropriate starting point is the all-right state, $|\rightarrow \cdots \rightarrow\rangle$, with small noise added, as it better approximates the ground state. Furthermore, in the critical case $g = J$, this state remains preferable for initialization. Although the ground state is highly entangled and not well-approximated by any product state, the symmetry of the all-right state is closer to that of the true critical ground state than the all-up state.

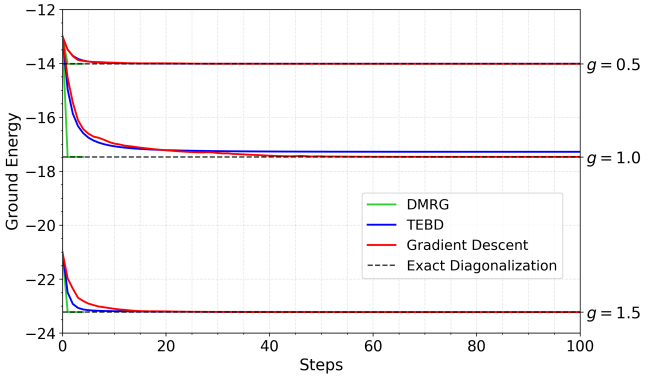

FIG. 7. Comparison of gradient descent with TEBD and DMRG.

Figure 7 shows the results of the optimization of the ground state of a $L = 14$, $J = 1$ TFI for different values of g using gradient descent, compared to the results obtained from the TEBD and DMRG methods. The ground state energy is plotted as a function of the number of the algorithm steps taken to arrive at such state.

For the gradient descent, an exponential learning rate decay was used, starting at 2×10^{-2} and decaying by a factor of 0.5 every 30 steps, as well as a bond dimension of $\chi = 30$.

For the TEBD, a second-order Suzuki-Trotter decomposition was used with an exponential time step decay starting at 10^{-1} and decaying by a factor of 0.1 every 30 steps. This decay was found to give the best results for the $g = 1.5$ case, meaning the TEBD could converge faster for the $g = 1.0$ or $g = 0.5$ by tuning the time step decay accordingly.

For the DMRG, the updates were performed with a maximum bond dimension of 100.

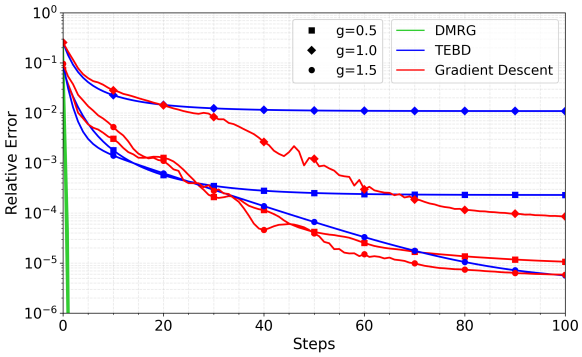

FIG. 8. Comparison of the relative error of the ground state energy obtained from gradient descent, TEBD and DMRG.

Figure 8 shows the relative error of the ground state energy obtained from the 3 methods. From it, it is possible to see the extreme speed at which DMRG converges, taking just one sweep to reach the ground state energy up to machine precision.

It is also possible to notice the worse performance of both TEBD and gradient descent for the critical case $g = 1.0$, where the ground state is highly entangled, leading to a slower and less accurate convergence. It is also possible to see that TEBD starts with a faster convergence than gradient descent, being overtaken by the latter after a few steps, though with an accuracy very close to that of gradient descent for the case $g = 1.5$, where the time step decay was optimized.

IV. CONCLUSION

Overall, the results obtained from simulations of the TFI model using gradient descent on MPS show that this method can be used to obtain accurate ground energies, even for highly entangled states. Nevertheless, DMRG remains the most efficient method for obtaining the ground state of a quantum system, converging to the ground energy in just one sweep, while gradient descent and TEBD require more iterations and careful parameter tuning. Still, gradient-based methods may still be used in systems where DMRG is less suitable, such as those non-Hermitian, with constraints, or when differentiability is needed.

[1] J. Hauschild and F. Pollmann, Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy), SciPost Phys. Lect. Notes , 5 (2018).
[2] G. Vidal, Efficient simulation of one-dimensional quantum many-body systems, Phys. Rev. Lett. **93**, 040502 (2004).
[3] S. R. White, Density matrix formulation for quantum renormalization groups, Phys. Rev. Lett. **69**, 2863 (1992).
[4] M. Fannes, B. Nachtergaele, and R. F. Werner, Finitely correlated states on quantum spin chains, Communications in Mathematical Physics **144**, 443 (1992).
[5] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, JAX: composable transformations of Python+NumPy programs, `http://github.com/jax-ml/jax` (2018), version 0.3.13.
[6] D. P. Kingma and J. L. Ba, Adam: A method for stochastic optimization, 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings (2014).
[7] DeepMind and Contributors, Optax: A gradient processing and optimization library for JAX, `https://github.com/deepmind/optax` (2020).
[8] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, Journal of Research of the National Bureau of Standards **45**, 255 (1950).

Appendix A: Code listing

1. optimize.py

```python
import os
from time import perf_counter as timer

import jax.numpy as jnp
from jax import jit, grad, config, block_until_ready
config.update("jax_enable_x64", True)
import optax

from src.MPS import MPS, init_spinup_MPS, init_spinright_MPS
from src.TFI import TFIModel
from src.TEBD import TEBD_engine
from src.DMRG import DMRGEngine
from src.ExactDiag import finite_gs_energy

import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
plt.rcParams.update({'font.size': 16})

images_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), "images")
if not os.path.exists(images_dir):
    os.makedirs(images_dir)

L = 14
J = 1.0
g = 1.5

theoretical_energies = {
    0.5: -14.01899646121673726,  # ground state energy for g=0.5
    1.0: -17.47100405473177176,  # ground state energy for g=1.0
    1.5: -23.22295943411735664   # ground state energy for g=1.5
}

energies_DMRG = []
energies_TEBD = []
energies_GD = []

theoretical_energies = {}

g_list = [0.5, 1.0, 1.5]
for g in g_list:
    theoretical_energies[g] = finite_gs_energy(L, J, g)

    if g <= J:
```

```
        init_MPS = init_spinright_MPS
    else:
        init_MPS = init_spinup_MPS

    print(f"Transverse-field Ising model with L={L}, J={J}, g={g}")

    model = TFIModel(L, J, g)

    steps = 100
    psi = init_MPS(L, 30, noise=True, eps=1e-5)

    print("Chi:", psi.get_chi())
    print(f"<psi|sigma_z|psi> =  {psi.site_expectation_value(model.sigmaz).sum():.8f}")
    print(f"<psi|sigma_x|psi> =   {psi.site_expectation_value(model.sigmax).sum():.8f}")
    print(f"<psi|psi>         =   {psi.norm_squared():.8f}")
    print(f"<psi|H|psi>       = {model.energy(psi):.8f}")
    print(f"<psi|H|psi>       = {model.energy_mpo(psi):.8f}")

    ##############################################################################
    #                                 DMRG                                       #
    ##############################################################################

    print("\nRunning DMRG...")

    energy_DMRG = [model.energy(psi)]
    dmrg = DMRGEngine(psi.copy(), model, chi_max=30, eps=1e-12)

    for sweep in range(4):
        dmrg.sweep()
        energy = model.energy(dmrg.psi)
        energy_DMRG.append(energy)

    print(f"DMRG ground state energy: {energy_DMRG[-1]:.5f}")

    energy_DMRG = jnp.array(energy_DMRG)
    energies_DMRG.append(energy_DMRG)

    ##############################################################################
    #                                 TEBD                                       #
    ##############################################################################

    print("\nRunning TEBD...")

    psi_TEBD = init_MPS(L, 2, noise=False)
    energy_TEBD = [model.energy(psi_TEBD)]
    scheduler = optax.schedules.exponential_decay(1.e-1, 30, 0.1, staircase=False)

    for step in range(steps):
        tebd = TEBD_engine(psi_TEBD, model, chi_max=30, eps=1e-10, dt=scheduler(step))
        energy = tebd.run(1, order=2)
        energy_TEBD += energy

        if step % 5 == 0:
            print(f"TEBD ground state energy: {energy[-1]:.5f}")

    energy_TEBD = jnp.array(energy_TEBD)
    energies_TEBD.append(energy_TEBD)

    ##############################################################################
    #                                  GD                                        #
    ##############################################################################

    @jit
    def loss(psi):
        """Loss function to minimize, which is the energy expectation value."""
        return model.energy_mpo(psi) / psi.norm_squared()

    @jit
    def loss_grad(psi):
        """Gradient of the loss function."""
```

```
        energy = model.energy_mpo(psi)
        norm_squared = psi.norm_squared()

        energy_grad = model.energy_mpo_grad(psi)
        norm_grad = psi.norm_squared_grad()

        Bs_grad = [energy_grad[i]/ norm_squared - energy * norm_grad[i] / norm_squared**2  for i in range(psi.L)]

        return MPS(Bs_grad, psi.Ss)

    scheduler = optax.schedules.exponential_decay(2.e-2, 30, 0.5, staircase=False)
    optimizer = optax.adam(learning_rate=scheduler)
    opt_state = optimizer.init(psi)

    energy_GD = [model.energy(psi)]

    print("\nRunning Gradient Descent...")

    t_grad_analytical = 0

    if g == 1.5:
        grad_error_avg = 0
        grad_error_std = 0
        grad_error_max = 0

        t_grad_jax = 0

        # First run for compilation
        grad_analytical = block_until_ready(loss_grad(psi))
        grads = block_until_ready(grad(loss)(psi))

    start = timer()
    for step in range(steps):
        t_start_analytical = timer()
        grad_analytical = block_until_ready(loss_grad(psi))
        t_grad_analytical += timer() - t_start_analytical

        if g == 1.5:
            t_start_jax = timer()
            grads = block_until_ready(grad(loss)(psi))
            t_grad_jax += timer() - t_start_jax

            for i in range(psi.L):
                errors = grad_analytical.Bs[i] - grads.Bs[i]
                grad_error_avg += jnp.mean(jnp.abs(errors))
                grad_error_std += jnp.std(errors)
                grad_error_max = max(grad_error_max, jnp.max(jnp.abs(errors)))

        # Apply the optimizer to update the MPS
        updates, opt_state = optimizer.update(grad_analytical, opt_state, psi)
        psi = optax.apply_updates(psi, updates)

        # Normalize the MPS (does not change the energy)
        psi = psi.normalize()

        energy = model.energy_mpo(psi)
        energy_GD.append(energy)

        if step % 5 == 0:
            print(f"Step {step:>4}, Loss: {energy:>9.5f}, Learning rate: {scheduler(step):>4.2e}")

    if g == 1.5:
        grad_error_avg /= steps
        grad_error_std /= steps

        t_grad_analytical /= steps
        t_grad_jax /= steps

    energy_GD = jnp.array(energy_GD)
```

```
    energies_GD.append(energy_GD)

    print(f"Optimization completed in {timer() - start:.3f} seconds\n")

##############################################################################
#                                Plotting                                    #
##############################################################################

# Plotting energy decay

fig, ax1 = plt.subplots(figsize=(10, 6))

for i, energy_DMRG in enumerate(energies_DMRG):
    ax1.plot(jnp.arange(5), energy_DMRG, label='DMRG' if i == 0 else "", linewidth=2, color="
    limegreen")
for i, energy_TEBD in enumerate(energies_TEBD):
    ax1.plot(jnp.arange(steps + 1), energy_TEBD, label='TEBD' if i == 0 else "", linewidth=2, color
    ="blue")
for i, energy_GD in enumerate(energies_GD):
    ax1.plot(jnp.arange(steps + 1), energy_GD, label='Gradient Descent' if i == 0 else "",
    linewidth=2, color="red")

ax1.set_xlabel("Steps", fontsize=17)
ax1.set_ylabel("Ground Energy", fontsize=17)
ax1.set_xlim(0, steps)
ax1.set_ylim(-24, -12)

ax2 = ax1.twinx()
ax2.set_ylim(-24, -12)

# Store ticks and labels
yticks = []
yticklabels = []

for i, (g, theoretical_energy) in enumerate(theoretical_energies.items()):
    ax1.axhline(y=theoretical_energy, color='black', linestyle='--', linewidth=1.5, alpha=0.8,
    label = "Exact Diagonalization" if i == 0 else "")
    yticks.append(theoretical_energy)
    yticklabels.append(fr'$g={g}$')

# Set the custom ticks and labels
ax2.set_yticks(yticks)
ax2.set_yticklabels(yticklabels, fontsize=17)

ax1.minorticks_on()
ax1.grid(which='major', axis='y', alpha=0.3, linestyle='--')
ax1.grid(which='both', axis='x', alpha=0.3, linestyle='--')
ax1.legend(loc='center right', bbox_to_anchor=(0.98, 0.30), fontsize=15)

# Remove right spine for cleaner look
ax1.spines['right'].set_visible(False)

# Adjust layout
plt.tight_layout()

# Save with high quality
plt.savefig(os.path.join(images_dir, "ground_state_optimization.png"), dpi=300, bbox_inches='tight'
    , facecolor='white')


# Plotting relative error
plt.figure(figsize=(10, 6))

markers = ["s", "D", "o"]
for i, (g, theoretical_energy) in enumerate(theoretical_energies.items()):
    relative_error_DMRG = jnp.abs((theoretical_energy - energies_DMRG[i]) / theoretical_energy)
    relative_error_TEBD = jnp.abs((theoretical_energy - energies_TEBD[i]) / theoretical_energy)
    relative_error_GD = jnp.abs((theoretical_energy - energies_GD[i]) / theoretical_energy)

```

```
    plt.plot(jnp.arange(5), relative_error_DMRG, marker=markers[i], markevery=10, color="limegreen", linewidth=2)
    plt.plot(jnp.arange(steps + 1), relative_error_TEBD, marker=markers[i], markevery=10, color="blue", linewidth=2)
    plt.plot(jnp.arange(steps + 1), relative_error_GD, marker=markers[i], markevery=10, color="red", linewidth=2)

plt.xlim(0, steps)
plt.ylim(1.e-6, 1.e-0)
plt.xlabel("Steps", fontsize=17)
plt.ylabel("Relative Error", fontsize=17)
plt.yscale('log')

# Custom legend (just measurements)
custom_lines = [
    Line2D([0], [0], color='limegreen'),
    Line2D([0], [0], color='blue'),
    Line2D([0], [0], color='red')
]
legend1 = plt.legend(custom_lines, ['DMRG', 'TEBD', 'Gradient Descent'], loc='upper right', fontsize=15)
plt.gca().add_artist(legend1)  # Add first legend manually

# Second legend
custom_markers = [
    Line2D([0], [0], color='black', marker='s', linestyle=''),
    Line2D([0], [0], color='black', marker='D', linestyle=''),
    Line2D([0], [0], color='black', marker='o', linestyle='')
]
legend2 = plt.legend(custom_markers, [f'g={g_list[0]}', f'g={g_list[1]}', f'g={g_list[2]}'], loc='upper right', fontsize=15, bbox_to_anchor=(0.65, 1.0))
plt.gca().add_artist(legend2)  # Add second legend manually

plt.grid()
plt.minorticks_on()
plt.grid(which='both', alpha=0.3, linestyle='--')
plt.savefig(os.path.join(images_dir, "relative_error_ground_state.png"), dpi=300)

# Plotting gradient comparison
fig, ax1 = plt.subplots(figsize=(10, 6))

# Define colors
colors = {
    'errors': "#FF7F0E",      # Modern orange
    'analytical': "#0000FF",  # Deep blue
    'jax': '#FF0000',         # Deep red
    'edge': "#000000"         # Black for edges
}

# Data preparation
error_data = [
    ("Average", jnp.abs(grad_error_avg)),
    ("Std Dev", jnp.abs(grad_error_std)),
    ("Maximum", jnp.abs(grad_error_max)),
]

error_labels = [item[0] for item in error_data]
error_vals = [item[1] for item in error_data]

# Create main error bars
X_axis = jnp.arange(len(error_labels))
bar_width = 0.55

bars1 = ax1.bar(X_axis, error_vals, bar_width,
                color=colors['errors'], edgecolor=colors['edge'],
                linewidth=1.5, alpha=0.8, label="Gradient Errors")

# Formatting for left y-axis
ax1.set_xticks(X_axis)
ax1.set_xticklabels(error_labels, fontsize=15)
```

```
ax1.set_ylabel("Absolute Error", fontsize=17)
ax1.set_yscale("log")
ax1.set_ylim(1e-14, 1e-12)
ax1.grid(True, which='both', alpha=0.3, linestyle='--')
ax1.tick_params(axis='y', labelsize=15)

# Create second y-axis for runtime
ax2 = ax1.twinx()
ax2.set_ylabel('Computation Time (s)', fontsize=17)
ax2.set_yscale('log')
ax2.set_ylim(1e-4, 1e-2)
ax2.tick_params(axis='y', labelsize=15)

# Add runtime bars
runtime_x = len(error_labels) + 0.5
bar_width_runtime = 0.45

# Add space for runtime bars
ax1.set_xlim(-0.5, runtime_x + 0.8)
ax2.set_xlim(-0.5, runtime_x + 0.8)

bars2 = ax2.bar(runtime_x - bar_width_runtime/2, t_grad_analytical, bar_width_runtime,
                label="Analytical Gradient", color=colors['analytical'],
                edgecolor=colors['edge'], linewidth=1.2, alpha=0.8)

bars3 = ax2.bar(runtime_x + bar_width_runtime/2, t_grad_jax, bar_width_runtime,
                label="JAX Gradient", color=colors['jax'],
                edgecolor=colors['edge'], linewidth=1.2, alpha=0.8)

# Update x-axis labels to include runtime
all_labels = error_labels + ["Runtime"]
ax1.set_xticks(list(X_axis) + [runtime_x])
ax1.set_xticklabels(all_labels, fontsize=15)

# Create a unified legend in the original position
handles1, labels1 = ax1.get_legend_handles_labels()
handles2, labels2 = ax2.get_legend_handles_labels()
ax1.legend(handles1 + handles2, labels1 + labels2, loc='upper left', fontsize=15)

# Remove top and right spines for cleaner look
ax1.spines['top'].set_visible(False)
ax2.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)

# Adjust layout
plt.tight_layout()

# Save with high quality
plt.savefig(os.path.join(images_dir, "gradient_comparison.png"), dpi=300, bbox_inches='tight',
    facecolor='white')

plt.show()
```

2. src/MPS.py

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

import jax
jax.config.update("jax_enable_x64", True)
import jax.numpy as jnp
from jax import tree_util, jit
from functools import partial

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 jnp.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 jnp.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 #stop_gradient(s) for s in Ss]
        self.L = len(Bs)

    @jit
    def copy(self):
        # return tree_util.tree_map(jnp.array, self)
        return MPS([jnp.array(B) for B in self.Bs], [jnp.array(S) for S in self.Ss])

    @jit
    def norm_squared(self):
        """Calculate the norm squared of the MPS, which is the overlap with itself."""
        return overlap(self, self).real

    @jit
    def norm_squared_grad(self):
        """
        Compute the gradient of the norm squared with respect to the MPS tensors.
        Args:
            psi: MPS object
        Returns:
            Gradient of the norm squared with respect to the MPS tensors.
        """
        grad = []

        contr_left = jnp.ones((1, 1), dtype=self.Bs[0].dtype)
        contr_right = jnp.ones((1, 1), dtype=self.Bs[0].dtype)

        left_blocks = [contr_left]
        right_blocks = [contr_right]

        for n in range(self.L - 1):
            M_bra = self.Bs[n].conj() # vL* i* vR*
            M_ket = self.Bs[n]        # vL i vR

            contr_left = jnp.tensordot(contr_left, M_bra, [0, 0])                 # [vR*] vR, [vL*] i* vR*
            contr_left = jnp.tensordot(contr_left, M_ket, axes=([0, 1], [0, 1]))  # [vR] [i*] vR*, [vL] [i] vR

            left_blocks.append(contr_left)

        for n in reversed(range(1, self.L)):
            M_bra = self.Bs[n].conj() # vL* i* vR*
            M_ket = self.Bs[n]        # vL i vR

            contr_right = jnp.tensordot(M_bra, contr_right, [2, 0])                 # vL* i* [vR*], [vL*] vL
            contr_right = jnp.tensordot(contr_right, M_ket, axes=([1, 2], [1, 2]))  # vL* [i*] [vL], vL [i] [vR]

```

```
            right_blocks.append(contr_right)

        for n in range(self.L):
            M_ket = self.Bs[n] # vL i vR
            contr_left = left_blocks[n]                # vR* vR
            contr_right = right_blocks[self.L - n - 1] # vL* vL

            grad_n = jnp.tensordot(contr_left, M_ket, [1, 0])   # vR* [vR], [vL] i vR
            grad_n = jnp.tensordot(grad_n, contr_right, [2, 1]) # vR* i [vR], vL* [vL]

            grad.append(2*grad_n)

        assert all(grad[i].shape == self.Bs[i].shape for i in range(self.L))
        return grad

    @jit
    def normalize(self):
        center = self.L // 2
        psi = self.copy()
        psi.Bs[center] = psi.Bs[center] / jnp.sqrt(psi.norm_squared())
        return psi

    @jit
    def canonicalize(self):
        """
        Return a new MPS in right-canonical form by a left-to-right SVD sweep.
        Optionally truncate to chi_max and discard singular values < eps.
        Does not modify self.
        """
        psi = self.copy() # Create a copy to avoid modifying self
        chis = psi.get_chi()
        for i in range(psi.L - 1):
            chivC = chis[i]
            j = i + 1
            theta = psi.get_theta2(i)  # vL i j vR

            Ai, Sj, Bj = split_theta(theta, chivC)

            # put back into MPS
            Gi = jnp.tensordot(jnp.diag(psi.Ss[i]**(-1)), Ai, axes=[1, 0])  # vL [vL*], [vL] i vC
            psi.Bs[i] = jnp.tensordot(Gi, jnp.diag(Sj), axes=[2, 0])  # vL i [vC], [vC] vC
            psi.Ss[j] = Sj #jax.lax.stop_gradient(Sj)  # vC
            psi.Bs[j] = Bj  # vC j vR
        return psi

    @partial(jit, static_argnames=["i"])
    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 jnp.tensordot(jnp.diag(self.Ss[i]), self.Bs[i], [1, 0])  # vL [vL'], [vL] i vR

    @partial(jit, static_argnames=["i"])
    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 jnp.tensordot(self.get_theta1(i), self.Bs[j], [2, 0])  # vL i [vR], [vL] j vR

    def get_chi(self):
        """Return bond dimensions."""
        return [self.Bs[i].shape[2] for i in range(self.L - 1)]

    @jit
    def site_expectation_value(self, op):
        """Calculate expectation values of a local operator at each site."""
        result = []
        for i in range(self.L):
            theta = self.get_theta1(i)  # vL i vR
```

```
            op_theta = jnp.tensordot(op, theta, axes=[1, 1])  # i [i*], vL [i] vR
            result.append(jnp.tensordot(theta.conj(), op_theta, [[0, 1, 2], [1, 0, 2]]))
            # [vL*] [i*] [vR*], [i] [vL] [vR]
        return jnp.real(jnp.array(result))

    @jit
    def bond_expectation_value(self, op):
        """Calculate expectation values of a local operator at each bond."""
        result = []
        for i in range(self.L - 1):
            theta = self.get_theta2(i)  # vL i j vR
            op_theta = jnp.tensordot(op[i], theta, axes=[[2, 3], [1, 2]])
            # i j [i*] [j*], vL [i] [j] vR
            result.append(jnp.tensordot(theta.conj(), op_theta, [[0, 1, 2, 3], [2, 0, 1, 3]]))
            # [vL*] [i*] [j*] [vR*], [i] [j] [vL] [vR]
        return jnp.real(jnp.array(result))

    @jit
    def entanglement_entropy(self):
        """Return the (von-Neumann) entanglement entropy for a bipartition at any of the bonds."""
        result = []
        for i in range(1, self.L):
            S = jnp.array(self.Ss[i])
            S = S[S > 1e-30]  # 0*log(0) should give 0; avoid warnings or NaN by discarding small S
            S2 = S * S
            assert abs(jnp.linalg.norm(S) - 1.) < 1.e-14
            result.append(-jnp.sum(S2 * jnp.log(S2)))
        return jnp.array(result)

    def _tree_flatten(self):
        children = (self.Bs, self.Ss)  # arrays / dynamic values
        aux_data = {}  # static values
        return (children, aux_data)

    @classmethod
    def _tree_unflatten(cls, aux_data, children):
        Bs, Ss = children
        return cls(Bs, Ss)

tree_util.register_pytree_node(MPS,
                               MPS._tree_flatten,
                               MPS._tree_unflatten)

@jit
def overlap(mps_bra, mps_ket):
    """
    Compute the overlap <mps_bra|mps_ket> for two MPS in right-canonical form.
    Both should be lists of tensors of the same length.
    """
    L = len(mps_bra.Bs)
    contr = jnp.ones((1, 1), dtype=mps_bra.Bs[0].dtype)
    for n in range(L):
        M_bra = mps_bra.Bs[n].conj() # vL* i* vR*
        M_ket = mps_ket.Bs[n]        # vL i vR

        contr = jnp.tensordot(contr, M_ket, axes=(1, 0)) # vR* [vR], [vL] j vR
        contr = jnp.tensordot(M_bra, contr, axes=([0, 1], [0, 1])) # [vL*] [j*] vR*, [vR*] [j] vR
    assert contr.shape == (1, 1)
    return contr[0, 0]

def init_spinup_MPS(L: int, chi_max: int, noise: bool = False, eps: float = 1e-4, key=None) -> MPS:
    """
    Create an all-up spin MPS with maximum bond dimension chi_max.
    Optionally add small noise to each tensor.

    Args:
        L: int, number of sites
        chi_max: int, maximum bond dimension (at the center)
        key: jax.random.PRNGKey or None
        noise: bool, whether to add noise
```

```
        eps: float, noise amplitude (if noise=True)
    Returns:
        mps: list of jnp.ndarray, each of shape (left, 2, right)
    """
    # Compute the staircase bond dimensions
    chi = [min(2**min(i, L-i), chi_max) for i in range(L+1)]
    shapes = [(chi[i], 2, chi[i+1]) for i in range(L)]
    Bs = []
    for left, phys, right in shapes:
        tensor = jnp.zeros((left, phys, right), dtype=jnp.float64)
        tensor = tensor.at[0, 0, 0].set(1.0)
        if noise:
            import jax.random as jr
            if key is None:
                key = jr.PRNGKey(42)
            subkey, key = jr.split(key)
            tensor += eps * jr.normal(subkey, shape=(left, phys, right), dtype=jnp.float64)
        Bs.append(tensor)
    Ss = [jnp.pad(jnp.ones([1], jnp.float64), (0, chi[i]-1)) for i in range(L)]
    mps = MPS(Bs, Ss)
    mps = mps.canonicalize() # Canonicalize to ensure the noise is properly incorporated
    # mps = mps.normalize()  # Not needed, as canonicalization already normalizes the MPS
    return mps

def init_spinright_MPS(L: int, chi_max: int, noise: bool = False, eps: float = 1e-4, key=None) ->
    MPS:
    """
    Create an all-right spin MPS with maximum bond dimension chi_max.
    Optionally add small noise to each tensor.

    Args:
        L: int, number of sites
        chi_max: int, maximum bond dimension (at the center)
        key: jax.random.PRNGKey or None
        noise: bool, whether to add noise
        eps: float, noise amplitude (if noise=True)
    Returns:
        mps: list of jnp.ndarray, each of shape (left, 2, right)
    """
    # Compute the staircase bond dimensions
    chi = [min(2**min(i, L-i), chi_max) for i in range(L+1)]
    shapes = [(chi[i], 2, chi[i+1]) for i in range(L)]
    Bs = []
    for left, phys, right in shapes:
        tensor = jnp.zeros((left, phys, right), dtype=jnp.float64)
        tensor = tensor.at[0, 0, 0].set(0.5**0.5)
        tensor = tensor.at[0, 1, 0].set(0.5**0.5)
        if noise:
            import jax.random as jr
            if key is None:
                key = jr.PRNGKey(42)
            subkey, key = jr.split(key)
            tensor += eps * jr.normal(subkey, shape=(left, phys, right), dtype=jnp.float64)
        Bs.append(tensor)
    Ss = [jnp.pad(jnp.ones([1], jnp.float64), (0, chi[i]-1)) for i in range(L)]
    mps = MPS(Bs, Ss)
    mps = mps.canonicalize() # Canonicalize to ensure the noise is properly incorporated
    # mps = mps.normalize()  # Not needed, as canonicalization already normalizes the MPS
    return mps

@partial(jit, static_argnames=["chivC"])
def split_theta(theta, chivC):
    """Split a two-site wave function in mixed canonical form.

    Split a two-site wave function as follows::
          vL --(theta)-- vR     =>    vL --(A)--diag(S)--(B)-- vR
                |   |                       |             |
                i   j                       i             j

    Parameters
```

```
    ----------
    theta : jnp.Array[ndim=4]
        Two-site wave function in mixed canonical form, with legs ``vL, i, j, vR``.
    chivC : int
        Maximum number of singular values to keep

    Returns
    -------
    A : jnp.Array[ndim=3]
        Left-canonical matrix on site i, with legs ``vL, i, vC``
    S : jnp.Array[ndim=1]
        Singular/Schmidt values.
    B : jnp.Array[ndim=3]
        Right-canonical matrix on site j, with legs ``vC, j, vR``
    """
    chivL, dL, dR, chivR = theta.shape
    theta = jnp.reshape(theta, [chivL * dL, dR * chivR])

    X, Y, Z = jnp.linalg.svd(theta, full_matrices=False) # returns Y sorted in descending order

    # truncate
    X, Y, Z = X[:, :chivC], Y[:chivC], Z[:chivC, :]

    Y = jnp.maximum(Y, 1e-12)  # avoid division by zero

    # renormalize
    S = Y / jnp.linalg.norm(Y)  # == Y/sqrt(sum(Y**2))

    # split legs of X and Z
    A = jnp.reshape(X, [chivL, dL, chivC])
    B = jnp.reshape(Z, [chivC, dR, chivR])
    return A, S, B

def split_truncate_theta(theta, chi_max, eps):
    """Split and truncate a two-site wave function in mixed canonical form.

    Split a two-site wave function as follows::
          vL --(theta)-- vR     =>    vL --(A)--diag(S)--(B)-- vR
                |   |                       |             |
                i   j                       i             j

    Afterwards, truncate in the new leg (labeled ``vC``).

    Parameters
    ----------
    theta : jnp.Array[ndim=4]
        Two-site wave function in mixed canonical form, with legs ``vL, i, j, vR``.
    chi_max : int
        Maximum number of singular values to keep
    eps : float
        Discard any singular values smaller than that.

    Returns
    -------
    A : jnp.Array[ndim=3]
        Left-canonical matrix on site i, with legs ``vL, i, vC``
    S : jnp.Array[ndim=1]
        Singular/Schmidt values.
    B : jnp.Array[ndim=3]
        Right-canonical matrix on site j, with legs ``vC, j, vR``
    """
    chivL, dL, dR, chivR = theta.shape
    theta = jnp.reshape(theta, [chivL * dL, dR * chivR])

    X, Y, Z = jnp.linalg.svd(theta, full_matrices=False) # returns Y sorted in descending order

    # truncate
    chivC = min(chi_max, jnp.sum(Y > eps))
    X, Y, Z = X[:, :chivC], Y[:chivC], Z[:chivC, :]

```

```
    # renormalize
    S = Y / jnp.linalg.norm(Y)  # == Y/sqrt(sum(Y**2))

    # split legs of X and Z
    A = jnp.reshape(X, [chivL, dL, chivC])
    B = jnp.reshape(Z, [chivC, dR, chivR])
    return A, S, B
```

3. src/TFI.py

```
"""Toy code implementing the transverse-field ising model."""

import jax
jax.config.update("jax_enable_x64", True)
import jax.numpy as jnp
from jax import tree_util, jit

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 jnp.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 = jnp.array([[0., 1.], [1., 0.]])
        self.sigmay = jnp.array([[0., -1j], [1j, 0.]])
        self.sigmaz = jnp.array([[1., 0.], [0., -1.]])
        self.id = jnp.eye(2)
        self._H_bonds = None
        self._H_mpo = None

    def _init_H_bonds(self):
        """Initialize `H_bonds` hamiltonian. Called by H_bonds."""
        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 * jnp.kron(sx, sx) - gL * jnp.kron(sz, id) - gR * jnp.kron(id, sz)
```

```
58            # H_bond has legs ``i, j, i*, j*``
59            H_list.append(jnp.reshape(H_bond, [d, d, d, d]))
60        self._H_bonds = H_list
61
62    def _init_H_mpo(self):
63        """Initialize the MPO representation of the Hamiltonian."""
64        W = jnp.zeros((3, 3, self.d, self.d))
65
66        W = W.at[0, 0].set(self.id)
67        W = W.at[0, 1].set(self.sigmax)
68        W = W.at[0, 2].set(-self.g * self.sigmaz)
69        W = W.at[1, 2].set(-self.J * self.sigmax)
70        W = W.at[2, 2].set(self.id)
71
72        self._H_mpo = [W.copy() for _ in range(self.L)]
73
74    @property
75    def H_bonds(self):
76        """Return the Hamiltonian bonds."""
77        if self._H_bonds is None:
78            self._init_H_bonds()
79        return self._H_bonds
80
81    @property
82    def H_mpo(self):
83        """Return the MPO representation of the Hamiltonian."""
84        if self._H_mpo is None:
85            self._init_H_mpo()
86        return self._H_mpo
87
88    @jit
89    def energy(self, psi):
90        """Evaluate energy E = <psi|H|psi> for the given MPS."""
91        assert psi.L == self.L
92        return jnp.sum(psi.bond_expectation_value(self.H_bonds))
93
94    @jit
95    def energy_mpo(self, psi):
96        """
97        Compute the expectation value <mps_bra|MPO|mps_ket> for two MPS and an MPO.
98        All should be lists of tensors of the same length.
99        Args:
100           mps_bra: MPS object (bra, conjugated)
101           mps_ket: MPS object (ket)
102           mpo: list of MPO tensors (one per site)
103       Returns:
104           Scalar energy expectation value
105       """
106       assert psi.L == self.L
107       left_vec = jnp.array([1, 0, 0], dtype=psi.Bs[0].dtype)
108       contr = left_vec.reshape(1, 3, 1)
109
110       for n in range(self.L):
111           M_bra = psi.Bs[n].conj() # vL* i* vR*
112           M_ket = psi.Bs[n]        # vL i vR
113           W = self.H_mpo[n]        # wL wR i* i
114
115           contr = jnp.tensordot(contr, M_bra, [0, 0])                 # [vR*] wR vR, [vL*] i* vR*
116           contr = jnp.tensordot(contr, W, axes=([0, 2], [0, 2]))      # [wR] vR [i*] vR*, [wL] wR
      [i*] i
117           contr = jnp.tensordot(contr, M_ket, axes=([0, 3], [0, 1]))  # [vR] vR* wR [i], [vL] [i]
      vR
118
119       assert contr.shape == (1, 3, 1)
120       return contr[0, 2, 0] # right_vec = jnp.array([0, 0, 1], dtype=psi.Bs[0].dtype)
121
122   @jit
123   def energy_mpo_grad(self, psi):
124       """
125       Compute the gradient of the energy expectation value with respect to the MPS tensors.
```

```
        Args:
            psi: MPS object
        Returns:
            Gradient of the energy expectation value with respect to the MPS tensors.
        """
        assert psi.L == self.L

        grad = []

        left_vec = jnp.array([1, 0, 0], dtype=psi.Bs[0].dtype)
        right_vec = jnp.array([0, 0, 1], dtype=psi.Bs[0].dtype)
        contr_left = left_vec.reshape(1, 3, 1)
        contr_right = right_vec.reshape(1, 3, 1)

        left_blocks = [contr_left]
        right_blocks = [contr_right]

        for n in range(self.L - 1):
            M_bra = psi.Bs[n].conj() # vL* i* vR*
            M_ket = psi.Bs[n]        # vL i vR
            W = self.H_mpo[n]        # wL wR i* i

            contr_left = jnp.tensordot(contr_left, M_bra, [0, 0])                 # [vR*] wR vR, [vL*] i* vR*
            contr_left = jnp.tensordot(contr_left, W, axes=([0, 2], [0, 2]))      # [wR] vR [i*] vR*, [wL] wR [i*] i
            contr_left = jnp.tensordot(contr_left, M_ket, axes=([0, 3], [0, 1]))  # [vR] vR* wR [i], [vL] [i] vR

            left_blocks.append(contr_left)

        for n in reversed(range(1, self.L)):
            M_bra = psi.Bs[n].conj() # vL* i* vR*
            M_ket = psi.Bs[n]        # vL i vR
            W = self.H_mpo[n]        # wL wR i* i

            contr_right = jnp.tensordot(M_bra, contr_right, [2, 0])                 # vL* i* [vR*], [vL*] wL vL
            contr_right = jnp.tensordot(contr_right, W, axes=([1, 2], [2, 1]))      # vL* [i*] [wL] vL, wL [wR] [i*] i
            contr_right = jnp.tensordot(contr_right, M_ket, axes=([1, 3], [2, 1]))  # vL* [vL] wL [i], vL [i] [vR]

            right_blocks.append(contr_right)

        for n in range(self.L):
            M_ket = psi.Bs[n] # vL i vR
            W = self.H_mpo[n] # wL wR i* i
            contr_left = left_blocks[n]                # vR* wR vR
            contr_right = right_blocks[self.L - n - 1] # vL* wL vL

            grad_n = jnp.tensordot(contr_left, M_ket, [2, 0])                   # vR* wR [vR], [vL] i vR
            grad_n = jnp.tensordot(grad_n, W, axes=([1, 2], [0, 3]))            # vR* [wR] [i] vR, [wL] wR i* [i]
            grad_n = jnp.tensordot(grad_n, contr_right, axes=([1, 2], [2, 1])) # vR* [vR] [wR] i*, vL* [wL] [vL]

            grad.append(2*grad_n)

        assert all(grad[i].shape == psi.Bs[i].shape for i in range(self.L))
        return grad

    def _tree_flatten(self):
        children = (self._H_bonds,)  # arrays / dynamic values
        aux_data = {
            "L": self.L,
            "J": self.J,
            "g": self.g,
        }  # static values
```

```
        return (children, aux_data)

    @classmethod
    def _tree_unflatten(cls, aux_data, children):
        obj = cls(aux_data["L"], aux_data["J"], aux_data["g"])
        obj._H_bonds, = children
        return obj

tree_util.register_pytree_node(TFIModel,
                               TFIModel._tree_flatten,
                               TFIModel._tree_unflatten)
```

4. src/ExactDiag.py

```
"""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 warnings

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 = sparse.linalg.eigsh(H, k=1, which='SA', return_eigenvectors=True)
    return E[0]
```

5. src/TEBD.py

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

import numpy as np
```

```
from scipy.linalg import expm
from src.MPS import split_truncate_theta

class TEBD_engine:
    def __init__(self, psi, model, chi_max, eps, dt):
        self.psi = psi
        self.model = model
        self.chi_max = chi_max
        self.eps = eps
        self.dt = dt
        self._U_bonds_dt = None
        self._U_bonds_half_dt = None

    def _init_U_bonds(self, 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 = self.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

    @property
    def U_bonds_dt(self):
        """Return the U_bonds for the full time step."""
        if self._U_bonds_dt is None:
            self._U_bonds_dt = self._init_U_bonds(self.dt)
        return self._U_bonds_dt

    @property
    def U_bonds_half_dt(self):
        """Return the U_bonds for the half time step."""
        if self._U_bonds_half_dt is None:
            self._U_bonds_half_dt = self._init_U_bonds(self.dt / 2)
        return self._U_bonds_half_dt

    def update_bond(self, i, bond="dt"):
        """Apply `U_bond` acting on i,j=(i+1) to `psi`."""
        if bond == "dt":
            U_bond = self.U_bonds_dt[i]
        elif bond == "half_dt" or bond == "dt/2":
            U_bond = self.U_bonds_half_dt[i]
        else:
            raise ValueError("bond must be 'dt', 'half_dt' or 'dt/2'")
        j = i + 1
        # construct theta matrix
        theta = self.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, 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

    def run(self, steps, order=2):
        """Evolve the state `psi` for `N_steps` time steps with TEBD.
        The state psi is modified in place."""
        energy_TEBD = []
        Nbonds = self.psi.L - 1

```

```
        if order == 1:
            for n in range(steps):
                for k in [0, 1]:  # even, odd
                    for i_bond in range(k, Nbonds, 2):
                        self.update_bond(i_bond, "dt")
                energy_TEBD.append(self.model.energy(self.psi))
            return energy_TEBD

        elif order == 2:
            for i_bond in range(0, Nbonds, 2): # even bonds
                self.update_bond(i_bond, "half_dt")
            for i_bond in range(1, Nbonds, 2): # odd bonds
                self.update_bond(i_bond, "dt")
            energy_TEBD.append(self.model.energy(self.psi))
            for n in range(steps - 1):
                for k in [0, 1]:  # even and odd
                    for i_bond in range(k, Nbonds, 2):
                        self.update_bond(i_bond, "dt")
                energy_TEBD.append(self.model.energy(self.psi))
            for i_bond in range(0, Nbonds, 2): # even bonds
                self.update_bond(i_bond, "half_dt")
            energy_TEBD.pop()
            energy_TEBD.append(self.model.energy(self.psi))
            return energy_TEBD

        else:
            raise ValueError("order must be 1 or 2")
```

6. src/DMRG.py

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

import numpy as np
from src.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, chi_max=100, eps=1.e-12):
        assert psi.L == model.L  # ensure compatibility
        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):
        # sweep from left to right
        for i in range(self.psi.L - 2):
            self.update_bond(i)
        # sweep from right to left
        for i in range(self.psi.L - 2, 0, -1):
            self.update_bond(i)

    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)
```