Recurrent Neural Network for Solving the Extreme Eigenvalue Problem

Heine Aabø and Stian Bilek

May 2019

Abstract

In this project we present a recurrent neural network (RNN) proposed by Feng et al. to solve the extreme eigenvalue problem of a simple pairing model. As this just turns out to be an iterative method solving an ordinary differential equation with an ODE-solver, we instead propose minimizing the given cost function with an adaptive gradient descent method (Rayleigh Quotient minimization). The results are presented with calculations by scipy.linalg.eigh, serving as a benchmark, along with the ODE-solver to show the improvement using gradient descent. The RNN outperformes scipy and the ODE-solver in terms of time with an increasing time gap as the matrix grows in size, as well as getting a relative energy difference in the order of 10^{-11} against the benchmark. When comparing the gradient descent method with the ODE-solver, we also achieved a more stable convergence as the latter method tended to be sensitive to the initial conditions. The algorithm is implemented in python and we have included it in the appendix.

1 Introduction

From the postulates of quantum mechanics, well introduced to all students of the field, we know that to every observable there is a linear, Hermitian operator $\hat{A}$ with a corresponding set of eigenvalues $\{a_i\}$ that satisfy

$$\hat{A}\Psi_i = a_i\Psi_i$$

Any measurement of the observable will result in one of these eigenvalues. When studying some quantum system we are usually interested in its total energy, described by the Hamiltonian $\hat{H}$. Particularly we want to find the lowest allowed energy level being the lowest eigenvalue. A commonly used method for this is configuration interaction (CI) which is based on the variational principle. CI requires diagonalization of a matrix which dimensionality increases rapidly with the number of particles and basis states. Conventional methods for diagonalizing this matrix can for large systems become both memory and time consuming. An interesting topic which we will look into in this project, is whether or not a neural network

can be used to more efficiently find the lowest eigenvalue of such matrices.

In this project the eigenvalue problem of the Hamiltonian of a simple pairing model will be solved using a recurrent neural network (RNN) proposed by Fuye Feng, Quanju Zhang, and Hailin Liu [3]. In reality this turns out to be an iterative scheme solving a set of differential equations. The authors solved these equations with a ODE-solver, but we propose a gradient descent minimization of the cost function with an adaptive learning rate. The Hamiltonian for increasing number of basis states will be found using CI and then the method of finding the lowest eigenvalue with an RNN will be benchmarked against scipy.linalg.eigh.

In section 2 (Theory), we will start by introducing the pairing model and find the corresponding Hamiltonian matrix. Then we will derive the recurrent neural network which we will use to solve for its eigenvalues. We then move to section 3 (Results) where we benchmark our RNN against numpy's eigenvalue solver. In section 4 (Discussion) we will discuss our results and conclude them in section 5 (Conclusion). Our RNN-class is included in the appendix in section 6.

Our github repository contains all relevant files and also a class for our RNN. It can be found here https://github.com/stiandb/CompPhysics2.

2 Theory

2.1 Configuration Interaction[1]

We want to solve the time-independent Schröedinger Equation

$$\hat{H} \left|\Phi_k\right\rangle = (\hat{H}_0 + \hat{H}_1) \left|\Phi_k\right\rangle = \epsilon_k \left|\Phi_k\right\rangle \tag{1}$$

The problem is that we do not know the solution when the particle-particle interaction is present. We do know however, that an arbitrary antisymmetric N-particle wave function can be written as a linear combination of Slater determinants. If we choose the 2K eigenfunctions ψ_k of $\hat{H}_0$ as our basis, the solutions to equation 1 are then given by

$$\left|\Phi_k\right\rangle = c_0^{(k)} \left|\Psi_0\right\rangle + \sum_{ia} c_i^{a(k)} \left|\Psi_i^a\right\rangle + \sum_{i<j,a<b} c_{ij}^{ab(k)} \left|\Psi_{ij}^{ab}\right\rangle + \cdots \tag{2}$$

We can write the Hamiltonian in the $\left|\Psi_i\right\rangle$ basis by utilizing the fact that $\sum_i \left|\Psi_i\right\rangle \left\langle\Psi_i\right| = I$:

$$\hat{H} = \sum_{ij} \left|\Psi_i\right\rangle \left\langle\Psi_i\right| \hat{H} \left|\Psi_j\right\rangle \left\langle\Psi_j\right| \tag{3}$$

By multiplying both sides with $\left\langle\Psi_l\right|$ and using equation 1 and 2, we see that

[1] this section is copy-pasted from an earlier project, found at https://github.com/stiandb/ManyBodyFys4480/tree/master/PDF

$$\langle \Psi_l | \hat{H} | \Phi_k \rangle = \sum_j \langle \Psi_l | \hat{H} | \Psi_j \rangle c_j = \epsilon_k c_l \tag{4}$$

This can be written in matrix notation as

$$\begin{bmatrix} \langle \Psi_0 | \hat{H} | \Psi_0 \rangle & \langle \Psi_0 | \hat{H} | \Psi_1 \rangle & \dots & \langle \Psi_0 | \hat{H} \left| \Psi_{\binom{2K}{N}} \right\rangle \\ \langle \Psi_1 | \hat{H} | \Psi_0 \rangle & \langle \Psi_1 | \hat{H} | \Psi_1 \rangle & \dots & \langle \Psi_1 | \hat{H} \left| \Psi_{\binom{2K}{N}} \right\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \left\langle \Psi_{\binom{2K}{N}} \right| \hat{H} | \Psi_0 \rangle & \left\langle \Psi_{\binom{2K}{N}} \right| \hat{H} | \Psi_1 \rangle & \dots & \left\langle \Psi_{\binom{2K}{N}} \right| \hat{H} \left| \Psi_{\binom{2K}{N}} \right\rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{\binom{2K}{N}} \end{bmatrix} = \epsilon_k \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{\binom{2K}{N}} \end{bmatrix} \tag{5}$$

Thus we can in principle solve equation 1 exactly by finding the eigenfunctions and eigenvalues of the above matrix. In practice however, we require an infinite number of single-particle wave functions to form a complete set, which means that the $\binom{2K}{N}$ Slater determinants will not form a basis for our N-particle functions. Nevertheless, the solutions will be exact within the N-particle subspace spanned by our Slater determinants.

2.2 Paring model

A common way to model the nuclear structure is to assume that nucleons form pairs. This is supported by the fact that even-even nuclei are slightly more bound than odd-even and odd-odd nuclei[1]. More energy is needed to excite the nuclei since a pair has to be broken.

Furthermore, the BCS theory of superconductivity assumes simultaneous creation and condensation of electron pairs, or Cooper pairs, due to electron-phonon interactions[2]. These realizations motivates the introduction of pairing interactions when studying certain many-body systems and their properties.

One such system is the simple pairing model of an ideal fermionic system where we assume no broken pairs, that is seniority quantum number $\mathcal{S} = 0$, and spin degeneracy 2.

2.2.1 Hamiltonian

The pairing model Hamiltonian can be written on the form

$$\begin{aligned} \hat{H} =& \hat{H}_0 + \hat{H}_1 \\ =& \sum_p \epsilon_p a_p^\dagger a_p - \frac{1}{2} g \sum_{pq} a_{p+}^\dagger a_{p-}^\dagger a_{q-} a_{q+} \end{aligned} \tag{6}$$

where ϵ_p is the single-particle energy of level $p = 1, 2, \dots$ and g is the pairing strength. Ideally the pairing strength can be treated as a constant, however for a realistic atomic nucleus this is not the case. Still it serves as a useful tool to study the pairing interaction. Assuming equally spaced single-particle orbitals by a constant ξ we can write $\epsilon_p = (p-1)\xi$, and introduce $\sigma = \pm$ representing the possible spin values. Next we recognize the number operator $\hat{n}_p = a_p^\dagger a_p$. The interaction part $\hat{H}_1$ can only excite a

pair of particles at a time, so it is convenient to introduce the pair creation and annihilation operators $P_p^+ = a_{p+}^\dagger a_{p-}^\dagger$ and $P_p^- = a_{p-} a_{p+}$. This yields the simple expression:

$$\hat{H} = \xi \sum_{p\sigma} (p-1)\hat{n}_p - \frac{1}{2} g \sum_{pq} P_p^+ P_q^- \tag{7}$$

Since we are only dealing with pairs of particles occupying different energy levels, we can turn to occupation representation to calculate the matrix with Slater determinants as basis states. The basis will be composed of all possible variations of particle-hole excitations. The total number of basis states then becomes the binomial coefficient

$$\binom{p}{n} = \frac{p!}{n!(p-n)!} \tag{8}$$

where p is the number of hole states and n is the number of particle pairs. Since the particle pairs can not be broken we can define a general basis states in terms of pair creation and annihilation operators

$$\left| \Phi_{ij\dots}^{ab\dots} \right\rangle = P_a^+ P_b^+ \dots P_k^- P_j^- P_i^- \left| \Phi_0 \right\rangle \tag{9}$$

with the ground state defined as

$$\left| \Phi_0 \right\rangle = \left(\prod_{i \leq F} P_i^\dagger \right) \left| 0 \right\rangle \tag{10}$$

acting on the vacuum state $|0\rangle$, where F denotes the Fermi level without degeneracy as we are exciting pairs. The matrix elements of eq. (7) is given by

$$H_{ij} = \langle \phi_i | \hat{H} | \phi_j \rangle = \langle \phi_i | \hat{H}_0 | \phi_j \rangle + \langle \phi_i | \hat{H}_1 | \phi_j \rangle \tag{11}$$

for the any basis state $|\phi_i\rangle$. Lets first deal with the one-body part, given as

$$\langle \phi_i | \hat{H}_0 | \phi_j \rangle = \xi \sum_{p\sigma} (p-1) \langle \phi_i | \hat{n}_{p\sigma} | \phi_j \rangle \tag{12}$$

where $\langle \phi_i | \hat{n}_{p\sigma} | \phi_j \rangle = \delta_{ij}$. Since only pairs will occupy a state we can substitute the sum over σ with 2. As the number operator acting on a state gives zero if no particle occupies that state ($a|0\rangle = 0$), we get

$$\langle \phi_i | \hat{H}_0 | \phi_i \rangle = 2\xi \sum_p (p-1) \delta_{p \in \phi_i} \tag{13}$$

where the Kronecker delta $\delta_{p \in \phi_i}$ is zero unless state p is occupied in ϕ_i.

For the two-body part, we only get a contribution if there is a maximum of one pair different in the two slater deteminants. First we cover

the diagonal elements:

$$\begin{aligned}\left\langle \Phi_{ij\dots}^{ab\dots} \right| \hat{H}_1 \left| \Phi_{ij\dots}^{ab\dots} \right\rangle &= -\frac{1}{2} g \sum_{pq} \left\langle \Phi_{ij\dots}^{ab\dots} \right| P_p^+ P_q^- \left| \Phi_{ij\dots}^{ab\dots} \right\rangle \\ &= -\frac{1}{2} g \left[\sum_{pq>F} \delta_{pq} \sum_{c>F} \delta_{qc} + \sum_{pq<F} \delta_{pq} \sum_{k<F}^{F} \delta_{kq} \right] \\ &= -\frac{1}{2} g \left[\sum_{a>F} 1 + \sum_{k<F} 1 \right] = -\frac{1}{4} g \cdot n_p \end{aligned} \tag{14}$$

where $c \in \Phi_{ij\dots}^{ab\dots}$ and $k \in \Phi_{ij\dots}^{ab\dots}$. The off-diagonal elements will be

$$\left\langle \Phi_{ij\dots}^{ab\dots} \right| \hat{H}_1 \left| \Phi_{ij\dots}^{ac\dots} \right\rangle = -\frac{1}{2} g \sum_{pq} \delta_{pb} \delta_{qc} \tag{15}$$

for one different excited pair, and

$$\left\langle \Phi_{ij\dots}^{ab\dots} \right| \hat{H}_1 \left| \Phi_{ik\dots}^{ab\dots} \right\rangle = -\frac{1}{2} g \sum_{pq} \delta_{pj} \delta_{qk} \tag{16}$$

for one pair different under the fermi-level.

We have included a python script to calculate the hamiltonian for arbitrary number of pairs, basis states, ξ and g on our github.

2.3 Recurrent Neural Network for finding eigenvalues

The most general recurrence formula for a recurrent neural network is

$$h_t = f(h_{t-1}, z_t) \tag{17}$$

where h_t is referred to as the hidden state at time t and z_t is some input represented as a time-sequence. The hidden state of the network is dependent on the previous hidden state and hence the network is called recurrent.

The eigenvalue problem is to find a vector $\boldsymbol{v}_k = [v_1^k, v_2^k, \cdots, v_n^k]^T$ with the corresponding eigenvalue λ_k which satisfies the following equation

$$(A - I\lambda_k)\boldsymbol{v}_k = 0. \tag{18}$$

where $A \in \mathbf{R}^{n \times n}$ and I is the identity matrix. A conventional method to solve this is to observe that the columns of $(A - I\lambda)$ has to be linearly dependent for this equation to have a solution. A matrix has linearly dependent columns if the determinant is zero

$$|A - I\lambda| = 0$$

This provides us with a polynomial equation of order n which yields us n eigenvalues. The eigenvalues can then be inserted one by one back into

equation 18 to solve n systems of linear equations with n variables. This method can be time consuming for large-scale problems.

In practical problems, we are often interested in finding only the largest or the smallest eigenvalue with the corresponding normalized eigenvector. Finding the smallest(largest) corresponds to minimizing(maximizing) the Rayleigh quotient:

$$R(\boldsymbol{x}) = \frac{\boldsymbol{x}^T A \boldsymbol{x}}{\boldsymbol{x}^T \boldsymbol{x}} \tag{19}$$

for a symmetric matrix A and a vector $\boldsymbol{x}$. To illustrate this we utilize the fact that any vector in $\mathbf{R}^n$ can be written as a linear combination of the n orthonormal eigenvectors $\boldsymbol{v}$ of A:

$$\boldsymbol{x} = \sum_{i=1}^{n} c_i \boldsymbol{v_i}$$

We choose that $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Plugging this into equation eq. (19) gives

$$\begin{aligned} R(\boldsymbol{x}) &= \frac{\sum_{i=1}^n c_i \boldsymbol{v}_i^T \sum_{j=1}^n c_j A \boldsymbol{v}_j}{\sum_{i=1}^n c_i \boldsymbol{v}_i^T \sum_{j=1}^n c_i \boldsymbol{v}_j} \\ &= \frac{\sum_{i=1}^n c_i \boldsymbol{v}_i^T \sum_{j=1}^n c_j \lambda_j \boldsymbol{v}_j}{\sum_{i=1}^n c_i \boldsymbol{v}_i^T \sum_{j=1}^n c_i \boldsymbol{v}_j} \\ &= \frac{\sum_{i=1}^n c_i^2 \lambda_i \boldsymbol{v}_i^T \boldsymbol{v}_i}{\sum_{i=1}^n c_i^2 \boldsymbol{v}_i^T \boldsymbol{v}_i} \\ &= \frac{\sum_{i=1}^n c_i^2 \lambda_i}{\sum_{i=1}^n c_i^2} \end{aligned}$$

Going from the second to the third line we utilized that the eigenvectors are linearly independent and from the third to the fourth line we utilized that they are normalized. Since $c_i^2 \geq 0$ and $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, we can substitute all λ_i with λ_1 to find a lower bound on $R(\boldsymbol{x})$

$$R(\boldsymbol{x}) \geq \frac{\sum_{i=1}^n c_i^2 \lambda_1}{\sum_{i=1}^n c_i^2} = \lambda_1 \frac{\sum_{i=1}^n c_i^2}{\sum_{i=1}^n c_i^2} = \lambda_1$$

Hence the minimization will yield the lowest possible eigenvalue. We find $R(\boldsymbol{x}) = \lambda_1$ when $c_1 = \pm 1$, so we will also have the corresponding eigenvector. Now over to the minimization. We have

$$\boldsymbol{\nabla} R(\boldsymbol{x}) = \frac{2A\boldsymbol{x}(\boldsymbol{x}^T\boldsymbol{x}) - 2(\boldsymbol{x}^T A \boldsymbol{x})\boldsymbol{x}}{(\boldsymbol{x}^T\boldsymbol{x})^2} \tag{20}$$

equating this to zero, multiplying by $(\boldsymbol{x}^T\boldsymbol{x})^2$ and dividing by 2 gives

$$0 = A\boldsymbol{x}(\boldsymbol{x}^T\boldsymbol{x}) - (\boldsymbol{x}^T A \boldsymbol{x})\boldsymbol{x} \tag{21}$$

A recurrent neural network (RNN) can be proposed from this by setting the equilibrium points of the network to be the points where the gradient

of the Rayleigh quotient is zero. It is governed by the following system of differential equations

$$\frac{d\boldsymbol{x}}{dt} = -A\boldsymbol{x}(\boldsymbol{x}^T\boldsymbol{x}) + (\boldsymbol{x}^T A\boldsymbol{x})\boldsymbol{x} \tag{22}$$

The largest eigenvalue can similarly be found by multiplying the right hand side of the equality with -1.

2.4 Gradient Descent - Optimal Step Length

The original article [3] proposed setting up the ODE as in eq. (22) and solve it with an ODE-solver, but we will instead minimize the cost function eq. (19) using gradient descent with an adaptive learning rate. We can introduce an optimal step length η that will lead us towards a minima by computing

$$\frac{dR(\boldsymbol{x} - \eta\nabla R)}{d\eta} = 0 \tag{23}$$

We start by defining

$$R(\boldsymbol{x} - \eta\nabla R) = \frac{[\boldsymbol{x} - \eta\nabla R]^T A[\boldsymbol{x} - \eta\nabla R]}{[\boldsymbol{x} - \eta\nabla R]^T[\boldsymbol{x} - \eta\nabla R]} = \frac{\alpha}{\beta} \tag{24}$$

which gives the the left side of eq. (23)

$$\frac{dR(\boldsymbol{x} - \eta\nabla R)}{d\eta} = \left(\frac{d\alpha}{d\eta}\beta - \alpha\frac{d\beta}{d\eta}\right)\frac{1}{\beta^2} \tag{25}$$

Next we introduce some variables, to simplify the notation. As we have

$$\begin{aligned}\alpha &= [\boldsymbol{x} - \eta\nabla R]^T A[\boldsymbol{x} - \eta\nabla R] \\ &= \boldsymbol{x}^T A\boldsymbol{x} - \eta\boldsymbol{x}^T A\nabla R - \eta\nabla R^T A\boldsymbol{x} + \eta^2\nabla R^T A\nabla R\end{aligned} \tag{26}$$

where $\boldsymbol{x}$ and ∇R are vectors and A is a matrix, clearly all values here are scalars. Also since A is hermitian we have that $\boldsymbol{x}^T A\nabla R = \nabla R^T A\boldsymbol{x}$. We can then rewrite eq. (26)

$$\alpha = a - 2\eta b + \eta^2 c \tag{27}$$

where

$$\frac{d\alpha}{d\eta} = -2b + 2\eta c \tag{28}$$

In the same notion we find that

$$\begin{aligned}\beta &= [\boldsymbol{x} - \eta\nabla R]^T[\boldsymbol{x} - \eta\nabla R] \\ &= \boldsymbol{x}^T\boldsymbol{x} - \eta\boldsymbol{x}^T\nabla R - \eta\nabla R^T\boldsymbol{x} + \eta^2\nabla R^T\nabla R\end{aligned} \tag{29}$$

gives

$$\beta = e - 2\eta f + \eta^2 g \tag{30}$$

with $\boldsymbol{x}^T\nabla R = \nabla R^T\boldsymbol{x}$, and where

$$\frac{d\beta}{d\eta} = -2f + 2\eta g \tag{31}$$

Setting eqs. (27) and (30) into eq. (25) gives

$$\frac{dR(\boldsymbol{x}-\eta\nabla R)}{d\eta} = [(2\eta c - 2b)(\eta^2 g - 2\eta f + e) - (\eta^2 c - 2\eta b + a)(2\eta g - 2f)]/\beta^2 \tag{32}$$

From eq. (23) with some calculations this becomes

$$0 = \eta^2(gb - cf) + \eta(ec - ag) - eb + af \tag{33}$$

and we get an expression for the step length η

$$\eta_\pm = \frac{(ag - ec) \pm \sqrt{(ec - ag)^2 - 4(af - eb)(bg - cf)}}{2(bg - cf)} \tag{34}$$

where we expect the solution η_+ to take us towards the minimum of $R(\boldsymbol{x})$ and η_- to take us towards the maximum. We can then update $\boldsymbol{x}_t$ by using

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta_+(\boldsymbol{x}_t)\nabla R(\boldsymbol{x}_t) \tag{35}$$

where $\eta_+(\boldsymbol{x})$ is given by eq. (34) and $\nabla R(\boldsymbol{x})$ is given by eq. (20).

2.5 scipy.linalg.eigh

Our results will be compared to the speed and eigenvalues computed with `scipy.linalg.eigh`. It uses the `LAPACK` routine `CHEEVR` computing eigenvalues with the dqds algorithm[6] and eigenvectors with Relatively Robust Representations[7]. Compared to numpy's similar function it takes two arguments: *eigvals* making it possible to only return the smallest eigenvalue, and *check_finite* which speeds it up.

3 Results

To test the performance of the RNN we apply it on the Hamiltonian given by eq. (7). We do this for two particle pairs and an increasing number of basis states and we save the time usage of `scipy.linalg.eigh` and our RNN. The result can be seen in figure 1. The results of the RNN is shown for two different methods minimizing eq. (19): gradient descent with convergence limit

$$\epsilon = \sum_i |\nabla R_i|$$

and with scipy's ODE-solver (originally proposed in [3]). The ODE-solver uses an initial guess for the eigenvector sampled from a uniform distribution, this gave the best results for it. With gradient descent the results are shown for two constant values of ϵ and two values increasing logarithmically with the size of the Hamiltonian. This is due to the fact that the performance of the gradient descent increased with the size of the Hamiltonian.

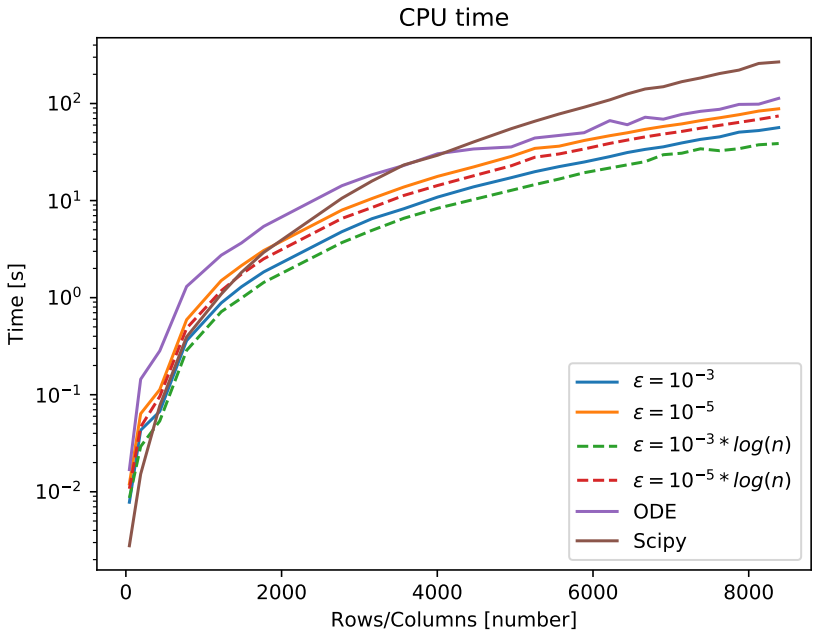

Figure 1: The time usage in seconds against size of the Hamiltonian matrix for different time steps against `scipy.linalg.eigh`.

We see that scipy was quicker than the RNN when the dimensions of the matrix was small, but as the dimensionality increased the RNN did better time-wise. We also see that the gradient descent method outperformed the ODE-method with respect to time usage. Figure 2 below shows the fractional time t_{scipy}/t_{RNN} against the number of rows of the matrix:

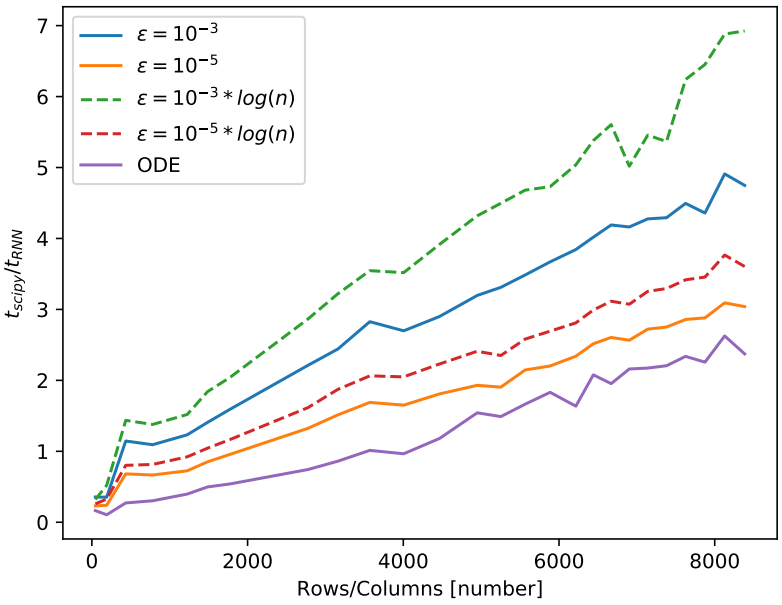

Figure 2: Fractional time between `scipy.linalg.eigh` and RNN against the size of the Hamiltonian. ODE uses scipy's ODE-solver, while ϵ is the convergence limit for the gradient descent.

We see that the trend appears linear in the number of columns/rows in favour of the RNN. To evaluate the found eigenvalues, the relative difference of the eigenvalues found with RNN and scipy, abs($\frac{E_{RNN}-E_{scipy}}{E_{scipy}}$), is plotted against number of rows/columns of our matrix and can be seen below in figure 3:

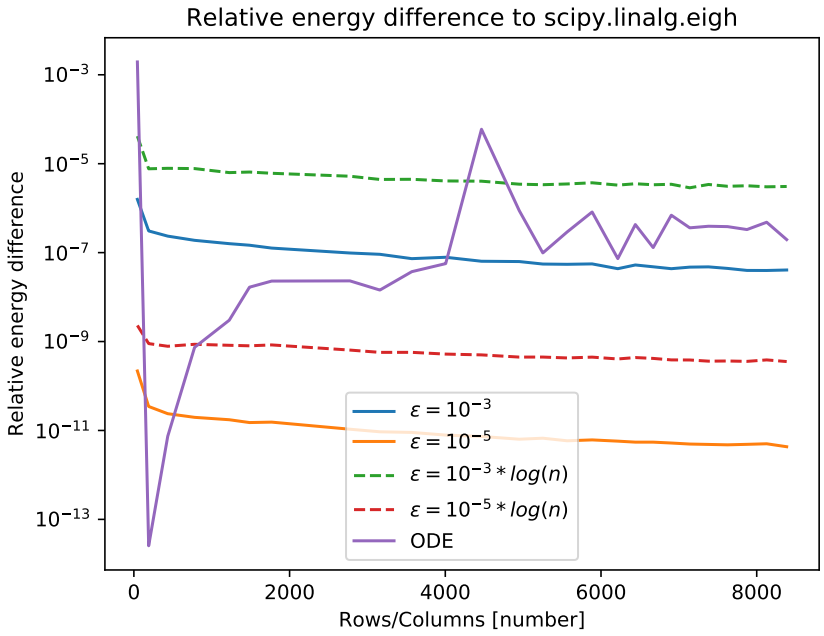

Figure 3: The relative difference between the energy calculated with `scipy.linalg.eigh` and the RNN against the size of the Hamiltonian.

We see that we found eigenvalues close to scipy's for all the matrix dimensions.

4 Discussion

First we discuss why we compared our method with scipy's eigenvalue solver. We looked at other methods for finding only the smallest or largest eigenvalue, like the iterative Lanczos algorithm[5]. We learned that this method, to the best of our knowledge, is restricted to positive definite matrices and therefore did not yield the correct eigenvalues for all our matrices. We also looked into Power iteration [4], but this method only yields the eigenvalue with the largest absolute value and hence did not fit our purpose either. `scipy.linalg.eigh` solves for symmetric, not necessarily positive definite matrices and hence we chose this method as our benchmark. We checked that for the eigenvalues found with scipy, we had a sufficiently small L2 norm for $A\boldsymbol{v} - \lambda\boldsymbol{v}$.

From fig. 1, comparing the CPU time of the different methods, we see that the RNN with gradient descent outperformed both `scipy.linalg.eigh` and the ODE-method as the dimensionality of the input matrix increased, while still achieving a low relative error in the energy estimate (see fig. 3).

Figure 2 shows that as we reached a dimensionality of 8385 rows, $t_{scipy} \approx 7t_{RNN}$ with a relative energy difference of only $|\frac{E_{RNN}-E_{scipy}}{E_{scipy}}| \approx 10^{-5}$, and it also suggests that the factor $\frac{t_{scipy}}{t_{RNN}}$ increases linearly with the dimension size. This is very welcome when dealing with high-dimensional matrices.

In fig. 3 we see that the eigenvalue computed with the RNN with gradient descent gets closer to scipy's eigenvalue as the Hamiltonian increases in size. The ODE-solver has a more irregular performance with random peaks, probably as a result of the random initialization, but with a clear trend as the Matrix size increase. From figs. 2 and 3 it is also apparent that having a convergence limit dependent of the matrix size decreases the time usage, keeping a respectable precision in the eigenvalue. When dealing with bigger systems, that is more particle pairs and basis states, the size of the Hamiltonian grows very large and so this decrease in time can become very useful.

All the matrices we apply the neural network on are block diagonal so it would be interesting to expand the result to other hamiltonians and also get a theoretical insight on the convergence properties of the algorithm. Different gradient descent methods could also serve as an interesting topic, as we are not sure if any other method will converge faster for this particular problem.

As a closing remark we would like to point out whether or not this actually is a recurrent neural network. In reality our algorithm spends all its time on the gradient descent and even though it is possible to argue that eqs. (17) and (35) are equivalent, it seems more fair to just call this a minimization problem of eq. (19).

5 Conclusion

We learned that the RNN provided a significant time decrease when compared with `scipy.linalg.eigh()`. Figure 2 shows that $t_{RNN} \approx \frac{1}{7}t_{scipy}$ for the largest matrix we tested (8385 rows and columns) with a relative difference in energy of only $|\frac{E_{RNN}-E_{scipy}}{E_{scipy}}| \approx 10^{-5}$ (fig. 3). Figure 2 also seems to suggest that the factor t_{scipy}/t_{RNN} increases linearly with matrix dimensions. We saw from comparing figs. 2 and 3 that the accuracy of the method depends on our convergence limit and that the gradient descent method with adaptive learning rate achieves higher (and more stable) accuracy than the ODE-method while also reaching convergence faster. We also see that introducing a convergence limit dependent on the matrix size, $\epsilon = a \cdot log(n)$, for some small a enables us to achieve a close to constant relative energy error while also providing a decrease in time usage. We only applied the model to a block-symmetric simple pairing hamiltonian and it would be interesting to test it with other systems and also research the theoretical convergence properties of the method.

Appendix

Here we present the RNN-class that we made to solve for the smallest eigenvalue of a symmetric matrix:

```
import numpy as np
import time
import sys
import matplotlib.pyplot as plt
from hamiltonian import *

class EigRNN:
    """
    Finds the lowest eigenvalue of a symmetric matrix
    by minimizing the Rayleigh quotient with gradient
    descent
    """
    def __init__(self,A,eps=1e-4,maxiter=1000):
        """
        A - input matrix
        eps - stops iterating when eigenvalue is not changing more
 ↪  than eps
        maxiter - maximum number of iterations
        """
        self.A = A
        self.maxiter = maxiter
        self.eps = eps
        self.x = np.random.rand(self.A.shape[0])

    def dR(self):
        """
        Returns the gradient of the Rayleigh quotient
        """
        x = self.x
        self.Ax = self.A@x
        self.xTx = x.T@x
        self.eig = x.T@self.Ax/self.xTx
        self.grad = 2*((self.xTx)*self.Ax -
         ↪  (x.T@self.Ax)*x)/((self.xTx)*(self.xTx))
        return(self.grad)

    def optStep(self):
        """
        Returns the optimal step length
        """
        dR = self.grad
        x = self.x
        xTA = self.Ax.T
        a = xTA@x
        b = xTA@dR
```

```
        c = dR.T@self.A@dR
        e = self.xTx
        f = x.T@dR
        g = dR.T@dR
        return( ( (a*g - e*c ) + np.sqrt( (e*c - a*g)**2 - 4*(a*f - e*b)*(b*g - c*f) ) )/(2*(b*g - c*f)))

    def solve(self):
        """
        Returns the lowest eigenvalue of A.
        To be called after initialization.
        """
        convergence = False
        val = 0
        for i in range(self.maxiter):
            grad = self.dR()
            self.x = self.x - self.optStep()*grad
            if np.sum(np.abs(grad)) < self.eps*(np.log(self.x.shape[0])):
                convergence = True
                break

        if not convergence:
            print('WARNING: Did not converge. Try increasing maxiter or a smaller eps.')

        return(self.eig,self.x)



if __name__ == '__main__':
    n_pairs = int(sys.argv[1])
    n_basis = int(sys.argv[2])
    print('System with {} pairs and {} basis states'.format(n_pairs,n_basis))
    delta = float(sys.argv[3])
    g = float(sys.argv[4])
    epsilon = float(sys.argv[5])
    H,Eref = hamiltonian(n_pairs,n_basis,delta,g)
    RNN = EigRNN(H,eps=epsilon)
    eigval,eigvec = RNN.solve()
    print('Energy: {}'.format(eigval))
```

References

[1] A. Bohr, B. R. Mottelson, and D. Pines. Possible analogy between the excitation spectra of nuclei and those of the superconducting metallic state. *Phys. Rev.*, 110:936–938, May 1958.

[2] Leon N. Cooper. Bound electron pairs in a degenerate fermi gas. *Phys. Rev.*, 104:1189–1190, Nov 1956.

[3] Fuye Feng, Quanju Zhang, and Hailin Liu. A recurrent neural network for extreme eigenvalue problem. 2005.

[4] Lambers, Jim. Lecture: The University of Southern Mississippi.

[5] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Natl. Bur. Stand. B*, 45:255–282, 1950.

[6] Shengguo Li, Ming Gu, and Beresford Parlett. An improved dqds algorithm. *SIAM Journal on Scientific Computing*, 36:C290–C308, 06 2014.

[7] Beresford Parlett and Inderjit S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Linear Algebra and its Applications*, 309:121–151, 04 2000.