UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2019:14

Linear-scaling recursive expansion of the Fermi-Dirac operator

Linnéa Andersson

Examensarbete i matematik, 15 hp
Handledare: Emanuel Rubensson
Ämnesgranskare: Denis Gaidashev
Examinator: Martin Herschend
Maj 2019

Department of Mathematics
Uppsala University

Linear-scaling recursive expansion of the Fermi-Dirac operator

Linnéa Andersson

Abstract

To assess the electronic structure of a large system of molecules calls for a method to efficiently evaluate the Fermi-Dirac matrix function. A recursive expansion resulting in an implicit expression which can be solved as a linear system of equations is explored and is demonstrated for the first time to achieve linear time complexity with the use of sparse matrix algebra. Two methods, linear conjugate gradient and Newton-Schulz iteration, are investigated with regards to their performance in solving the linear system at every step in the recursive expansion at finite electronic temperature. At zero temperature the recursive expansion method is compared to the second-order spectral projection method, which is known to be one of the most efficient methods.

Acknowledgements

I would like to thank my supervisor Emanuel Rubensson as well as Anders Niklasson for introducing me to this topic, aiding me throughout the project and making it fun to work with.

Contents

1 Introduction

With modern computing it is today possible to simulate the properties of materials on a quantum mechanical level, which has useful applications in areas such as materials science, chemistry and biology. To determine the electronic structure of a molecular system consisting of millions of atoms requires means of calculation that scale well in terms of memory usage and computation time. The current search is for algorithms which scale linearly, $O(n)$, with problem size n. A vital computational task in electronic structure calculations concerns the occupancy of energy states or orbitals given the temperature of a system. Electrons belong to the particle group fermions, with the attribute that no two fermions with the same spin within a system can occupy the same energy level. The probability that a certain energy ϵ in a molecular system is occupied is given by the Fermi-Dirac distribution:

$$f_{FD}(\epsilon) = \frac{1}{e^{\beta(\epsilon-\mu)}+1} \tag{1}$$

where μ is the chemical potential and $\beta = 1/(k_B T)$ where k_B is the Boltzmann constant and T the electronic temperature.

In quantum physics the interactions and kinetic energy of particles within a system is described by the Hamiltonian. However calculations on the exact Hamiltonian are practically impossible. To make computations practically feasible the Hamiltonian is usually represented using density functional theory in which case the matrix is referred to as the Kohn-Sham matrix, or using the Hartree-Fock approximation. The eigenvalues of the Kohn-Sham or Hartree-Fock Hamiltonian, henceforth referred to as F, then correspond to orbital energy. From the Hamiltonian the density matrix D can be constructed by applying the Fermi-Dirac operator to F. The eigenvalues of D then give the occupation level for the eigenvalues of F, i.e. the occupation of the electronic states.

Application of the Fermi-Dirac function to a matrix F can easily be performed after an eigendecomposition of F. Since F always has the property of being real and symmetric it can be decomposed as

$$F = V\Lambda V^T \tag{2}$$

where Λ is a diagonal matrix with the eigenvalues of F on the diagonal and the columns of V consist of the corresponding eigenvectors. The Fermi-Dirac function applied to F is then defined as

$$f_{FD}(F) = V[e^{\beta(\Lambda-\mu)}+1]^{-1}V^T \tag{3}$$

At zero temperature the density matrix is given by the Heaviside step function

$$\theta(F) = V\theta(\mu I - \Lambda)V^T$$

were

$$\theta(x) = \begin{cases} 1, & \text{if } x < \mu. \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

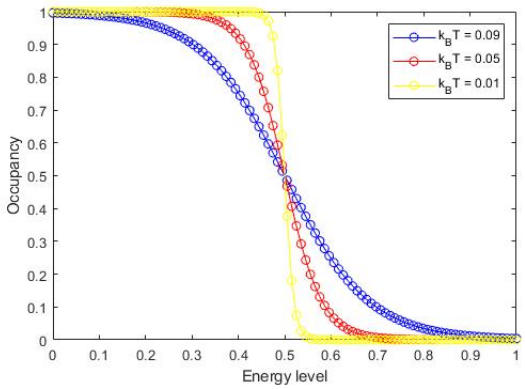


Figure 1: The Fermi-Dirac distribution for different temperatures. As temperature goes to zero it approaches a step function centered at $\mu = 0.5$.

However, for large atomic systems this is not an efficient way to calculate the density matrix, as the computational cost of standard diagonalization scale cubically, $O(n^3)$, with system size n. Therefore a lot of effort in recent years has been put in to develop alternative means of evaluating the density matrix. Examples of methods include constrained minimization schemes, building the density matrix by Chebyshev expansion, recursive density matrix purification or sign matrix methods[1, 4]. It is also possible to use divide-and-conquer or graph partitioning algorithms. In this work, the focus is on a recursive expansion of the Fermi-Dirac operator developed by Niklasson [2]. It results in an implicit iteration process in which a linear system of equations has to be solved at every iteration. In theory it should scale linearly with system size with the help of sparse matrix algebra, at least for sufficiently large problems with non-metallic elements or systems at high enough temperatures. However linear-scaling complexity has yet not been shown in practise. A key purpose with this study is to demonstrate linear-scaling complexity of the recursive expansion of the f_{FD}-operator for the first time. Two methods for solution of the linear system are analyzed for effectiveness in relation to the density matrix problem - the conjugate gradient method and using Newton-Schulz iteration for finding the matrix inverse. At zero temperature the efficiency of the recursive expansion technique is compared to the second-order spectral projection (SP2) technique.

2 Approximation of the Fermi-Dirac operator

Two polynomial expansion methods are here examined, the commonly used Chebyshev expansion and the recursive expansion method which is later implemented and numerically analyzed.

2.1 Chebyshev expansion

The Fermi-Dirac matrix function can be expanded in terms of polynomials. A common choice are the Chebyshev polynomials T_n where T_n has degree n.

$$f_{FD}(X) \approx P(X) = \sum_{i=1}^{n} c_i T_i(X) \tag{5}$$

The Chebyshev polynomials arise when trying to find the polynomial which best approximates a given continuous function $f(x)$ on the interval $[-1, 1]$. Given a set of data points $x_1, x_2, ...x_n$ and a function f there exists an interpolation polynomial $P_{n-1}(x)$ of degree $n-1$ such that $P_{n-1}(x_i) = f(x_i)$ at each point x_i. The interpolation error at some point x is given by [11]

$$f(x) - P_{n-1}(x) = \frac{f^{(n)}(\xi)}{n!} \prod_{i=1}^{n} (x - x_i) \tag{6}$$

for some $\xi \in [-1, 1]$. To find an upper bound for the error, Chebyshev showed that the polynomial which minimizes

$$\max_{x, x_i \in [-1,1]} \left| \prod_{i=1}^{n} (x - x_i) \right| \tag{7}$$

is $\frac{T_n}{2^{n-1}}$ where $T_n = \cos(n \arccos(x))$ [11] came to be known as the Chebyshev polynomials. They satisfy the recurrence relation

$$\begin{aligned} &T_0(x) = 1 \\ &T_1(x) = x \\ &T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \end{aligned} \tag{8}$$

which can be used for evaluating the series. T_n has n roots located at $x_k = \cos \frac{(2k-1)\pi}{2n}$. Note that these can be regarded as interpolation points with the property that they are unequally spaced. This avoids the Runge phenomenon of high oscillations near the end points for high polynomial orders which may occur when using equidistant points. With the weight function $w(x) = \frac{1}{\sqrt{1-x^2}}$ the Chebyshev polynomials form an orthogonal sequence

$$\int_{-1}^{1} w(x) T_m(x) T_n(x) dx = \begin{cases} \pi, & n = m = 0 \\ \frac{\pi}{2} & n = m \neq 0 \\ 0 & n \neq m \end{cases} \tag{9}$$

and this orthogonality relation can be used to determine the coefficients c_n. Let

$$f(x) = \sum_{i=0}^{N} c_i T_i(x) \tag{10}$$

and multiply by $T_n(x)w(x)$ and integrate over $[-1,1]$ which gives

$$c_0 = \frac{1}{\pi}\int_{-1}^{1} f(x)w(x)T_0(x)dx \tag{11}$$

$$c_n = \frac{2}{\pi}\int_{-1}^{1} f(x)w(x)T_n(x)dx \tag{12}$$

The coefficient integrals can be evaluated using the Fast Fourier Transform. Before applying a Chebyshev expansion to $f_{FD}(X)$ the eigenvalues of the Hamiltonian need to be rescaled to the $[-1,1]$-interval which can be accomplished with a linear transformation since its eigenvalues are positive and finite. To achieve a polynomial order n requires $O(n)$ matrix multiplications using the recurrence relation (8). However by a direct expansion only $O(\sqrt{n})$ matrix multiplications are required [14], which is optimal for any polynomial. Next we shall explore a simpler, recursive polynomial expansion method where the number of matrix multiplications scale only as $O(log(n))$ in relation to the expansion order n.

2.2 Recursive expansion

One way to approximate a large class of functions is with a a Padé approximation, using rational functions. Niklasson [2] showed that for large n on the [0,1] interval

$$f_n = \frac{(1-x)^n}{x^n + (1-x)^n} \approx \frac{1}{e^{4n(x-0.5)}+1} \tag{13}$$

which corresponds to the Fermi-Dirac function with $\beta = 4n$ and $\mu = 0.5$. The expression is based on an approximation of the exponential function which can be written as

$$e^x = (e^{x/n})^n = \left(\frac{e^{x/2n}}{e^{-x/2n}}\right)^n \tag{14}$$

Using Taylor series we get that

$$\lim_{n\to\infty}\frac{e^{x/2n}}{e^{-x/2n}} = \lim_{n\to\infty}\frac{1+\frac{1}{2n}x+\frac{1}{(2n)^2}\frac{x^2}{2!}+....}{1-\frac{1}{2n}x+\frac{1}{(2n)^2}\frac{x^2}{2!}-....} = \frac{2n+x}{2n-x} \tag{15}$$

So

$$e^x = \lim_{n\to\infty}\left(\frac{2n+x}{2n-x}\right)^n \tag{16}$$

This in turn gives that

$$\Phi(x) = [e^x+1]^{-1} = \lim_{n\to\infty}\frac{(2n-x)^n}{(2n+x)^n+(2n-x)^n} \tag{17}$$

If $x = 2n - 4nx$, then for large n

$$\Phi(2n-4nx) \approx \frac{x^n}{x^n+(1-x)^n} \tag{18}$$

The function $f_n = \frac{x^n}{x^n+(1-x)^n}$ has the important recursive property that

$$f_{n=m\times k}(x) = f_m(f_k(x)), \tag{19}$$

which enables an efficient recursive expansion of the function for large values of n. For example, if $n = 1024$ and we were to expand $f_{1024}(x)$ it requires only 10 matrix multiplications plus the cost of solving a linear system 10 times. It is possible to evaluate

$$X_{i+1} = [X_i^2 + (I - X_i)^2]^{-1} X_i^2 \tag{20}$$

in the same complexity range as doing a matrix multiplication, so calculating the function in a recursive manner is advantageous - to achieve expansion order n, only $O(log(n))$ iterations are required.

To be able to evaluate the Fermi-Dirac function with this scheme at a desired temperature and chemical potential, an initial transformation is required. It can be chosen as

$$X_0 = f_0(F) = a_0(\mu I - F) + 0.5I \tag{21}$$

with

$$a_0 = \frac{\beta}{4n} \tag{22}$$

at finite temperature where n is the full expansion order. At zero temperature a_0 can be chosen as

$$a_0 = \frac{1}{2} min\left(\frac{1}{\mu - \lambda_{min}}, \frac{1}{\lambda_{max} - \mu}\right) \tag{23}$$

to rescale the eigenvalues of F to the $[0, 1]$-interval. At zero temperature the expansion is convergent for any initial transformation, however this rescaling will improve the rate of convergence. The minimum and maximum eigenvalues can be approximated for instance using the Gersgorin circle theorem.

In the resulting iterative relation (20) all matrices are of size $N \times N$, so there are N separate linear systems to be solved at every iteration. The matrix $[X_i^2 + (I - X_i)^2]$ is symmetric positive definite and X_i is expected to be fairly close to X_{i+1}, making it a good starting guess for X_{i+1}. The system (20) is also well-conditioned. Consequently the system is well-suited for solution with the conjugate gradient method as presented in the next section. Another approach is Newton-Schulz iteration for finding the inverse of $[X_i^2 + (I - X_i)^2]$, where $A_i^{-1} = [X_i^2 + (I - X_i)^2]^{-1}$ is expected to be a good starting guess for A_{i+1}^{-1}. For the first iteration of Newton-Schulz a conjugate gradient solution of $AX = I$ can provide a good starting guess. The two iterative processes for evaluating the recursive expansion are thus

Algorithm 1 Recursive expansion iteration with conjugate gradient

$X_0 = f_0(F)$
for $i = 1, 2, ..., log_2$n **do**
  Calculate X_i^2
  Calculate $A = 2X_i^2 - 2X_i + I$
  for $j = 1, 2, ..., N$ **do**
    $x_0 =$ jth column of X_i # Starting guess
    $b =$ jth column of X_i^2
    Solve $Ax = b$ with conjugate gradient
  end for
end for

Algorithm 2 Recursive expansion iteration with Newton-Schulz

$X_0 = f_0(F)$
for $i = 1, 2, ..., log_2$n **do**
  Calculate X_i^2
  Calculate $A_i = 2X_i^2 - 2X_i + I$
  if i = 1 **then**
    Do conjugate gradient on $A_1 X = I$ to find good starting guess A_1^{-1}
  else
    Use A_{i-1}^{-1} as a starting guess
  end if
  Newton-Schulz iteration to find A_i^{-1}
  $X_{i+1} = A_i^{-1} X_i^2$
end for

3 Conjugate gradient method

3.1 A geometric approach

Intuitively, the conjugate gradient method can be understood as converting the problem of solving a linear system $Ax = b$ to the problem of minimizing the function

$$q(x) = \frac{1}{2}x^T Ax - x^T b \tag{24}$$

which has the gradient

$$\nabla q(x) = \frac{1}{2}x^T A + \frac{1}{2}Ax - b \tag{25}$$

For symmetric A we have that $x^T A = A^T x$, so

$$\nabla q(x) = Ax - b \tag{26}$$

Also note that the Hessian of $q(x)$ is $\nabla^2 q(x) = A$, so the Hessian is positive definite if A is. In this case $q(x)$ will form a hyperbolic paraboloid in the hyperplane with a global minimum at $\nabla q(x) = Ax - b = 0$. Locating this minimum is equivalent to solving $Ax = b$.

A reasonable approach to locating the minimum would be to start with an initial guess for the minimum x_0 and from there follow the plane downwards. The method of steepest descent urges us to search in the direction that $q(x)$ decreases fastest, which is the opposite direction of $\nabla q(x)$. Define the residual in the $k : th$ iteration as $r_k = -\nabla q(x) = b - Ax_k$ and we can set up steepest descent iteration as

$$x_k = x_{k-1} + \alpha_{k-1} r_{k-1} \tag{27}$$

where α_k is the step size at every iteration. Note that we are moving in the direction of $-\nabla q(x_{k-1})$, that is, the locally steepest direction. The step size is found by minimizing $q(x)$ in the movement direction, hence minimizing $f(\alpha) = q(x_{k-1} + \alpha_{k-1} r_{k-1})$.

$$f'(\alpha) = r_{k-1}^T q'(x_{k-1} + \alpha r_{k-1}) = 0 \quad \Rightarrow \quad r_{k-1}^T q'(x_k) = 0 \quad \Rightarrow \quad r_{k-1}^T r_k = 0$$

It follows that α must be chosen so that $r_{k-1}^T r_k = 0$. By multiplying (27) with A and subtracting b one can derive the expression $r_k = r_{k-1} - a_{k-1} A r_{k-1}$ for the residuals. Multiplying by r_{k-1}^T on both sides yields

$$\begin{aligned} r_{k-1}^T r_k = r_{k-1}^T r_{k-1} - \alpha_{k-1} r_{k-1}^T A r_{k-1} \quad \Rightarrow \\ 0 = r_{k-1}^T r_{k-1} - \alpha_{k-1} \quad \Rightarrow \quad \alpha_{k-1} = \frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T A r_{k-1}} \end{aligned} \tag{28}$$

Note that $r_{k-1}^T r_k = 0$ implies that the current search direction is orthogonal to the previous one. This means that the solution will have a zigzagging behavior towards convergence, where previous directions or components of previous directions could be searched multiple times. For optimal convergence rate, we only want to search each direction once.

3.2 Conjugacy

The solution is to have independent search directions. This is accomplished by requiring search directions to be A- conjugate. Two vectors u and v are A-conjugate if $u^T A v = 0$. This scalar product also defines the A- norm:

$$||v||_A = \sqrt{v^T A v} \tag{29}$$

Let $x = A^{-1} b$ be the exact solution of $Ax = b$ and x_k the approximation after k iterations. A- conjugate search directions is motivated by the fact that minimizing the error $x - x_k$ in the A- norm is equivalent with minimizing $q(x) = \frac{1}{2} x^T A x - x^T b$ and thus solving $Ax = b$. We have that [9]

$$
\begin{aligned}
q(x_k) &= \frac{1}{2}x_k^T A x_k - x_k^T b \\
q(x) &= \frac{1}{2}x^T A x - x^T b = \frac{1}{2}(A^{-1}b)^T A A^{-1}b - b^T A^{-1}b = -\frac{1}{2}b^T A^{-1}b \\
\frac{1}{2}||x - x_k||_A^2 &= \frac{1}{2}(x - x_k)^T A(x - x_k) = \frac{1}{2}x_k^T A x_k - x_k^T A x + \frac{1}{2}x^T A x \\
&= \frac{1}{2}x_k^T A x_k - x_k^T b + \frac{1}{2}b^T A^{-1}b
\end{aligned}
$$

It follows that

$$q(x) - q(x_k) = \frac{1}{2}||x - x_k||_A^2 \tag{30}$$

The A- conjugate search directions can be found iteratively. If p_k is the current search direction then the next search direction is found and x_k is updated by

$$p_{k+1} = r_{k+1} + \beta_k p_k \tag{31}$$

$$x_{k+1} = x_k + \alpha_k p_{k+1} \tag{32}$$

A new expression for α_k can be found in the same way as before to be

$$\alpha_k = \frac{r_k^T p_k}{p_k A p_k} \tag{33}$$

and where β_k is chosen so that $p_{k+1}^T A p_k = 0$. It can be shown by induction that this condition will guarantee $p_i^T A p_j = 0 \quad \forall \quad i \neq j$. Multiplying the equation by $p_k^T A$ and using A- conjugacy gives

$$\beta_k = \frac{p_k^T A r_{k+1}}{p_k^T A p_k} \tag{34}$$

A more efficient way to calculate β_k can be obtained by exploiting the relation $r_{k+1} = r_k - \alpha_k A p_k$. Through this one obtains $r_{k+1}^T r_{k+1} = -\alpha_k r_{k+1}^T A p_k$ and since $\alpha_k = \frac{r_k^T p_k}{p_k A p_k}$, β_k can be expressed as

$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \tag{35}$$

3.3 The Krylov subspace

The conjugate gradient method is one among many Krylov subspace methods. The idea is to minimize $x - x_k$ in every iteration over an expanded search space until the search space spans R^n. In conjugate gradient iteration, a sequence of nested subspaces $K_1 \subset K_2 \subset ... \subset K_n$ is built. The Krylov subspace sequence K_n generated by matrix $A \in R^{n \times n}$ and vector $b \in R^n$ is generated by

$$K_n(A, b) = span\{b, Ab, A^2 b ... A^{n-1} b\} \tag{36}$$

Since $r_0 = b - Ax_0$, the residuals $r_0, r_1 ... r_n$ span the same Krylov sequence

$$K_n(A, b) = K_{n-1}(A, r_0) = span\{r_0, Ar_0, A^2 r_0 ... A^{n-2} r_0\} \tag{37}$$

To see this we can use the relation $r_k = r_{k+1} - \alpha A r_{k-1}$. Since $r_1 = r_0 - \alpha A r_0$ and $r_2 = r_0 - 2\alpha A r_0 - \alpha^2 A^2 r_0$ means that $r_1 \in K_2(A, r_0)$ and $r_2 \in K_3(A, r_0)$ and so on. In every iteration the residual r_k is minimized over an expanded subspace K_{k+1}.

A Krylov sequence has the property that there exists an integer $1 \leq d = d(A, v) \leq n$ so that the vectors $v, Av, ...Av^{d-1}$ are linearly independent but the vectors $v, Av, ...Av^{d-1}, Av^d$ are linearly dependent [7]. This means that the sequence eventually will stop growing, when it becomes invariant under A and $K_d(A, v) = K_{d+1}(A, v)$. Then r_n can be written in terms of $r_n = p(A)r_0$ where p(A) is a polynomial of A. The condition for convergence $r_n = 0$ is then equivalent with at least two of the vectors building the polynomial p(A) being linearly dependent, as it otherwise cannot be zero.

3.4 Convergence

As A is Hermitian it is possible to find orthogonal eigenvectors of A which span R^n. Let λ_i and v_i be the eigenvalues and orthogonal eigenvectors of A. Then b can be expressed in terms of the eigenvectors of A [8, 10].

$$b = \sum_i^m \gamma_i v_i \tag{38}$$

If $x_k = P(A)_k b \in K_k(A, b)$ and $x = A^{-1}b$, the error after k iterations can be expressed as

$$||x - x_k||_A^2 = ||(A^{-1} - P_k(A))b||_A^2 = b^T (A^{-1} - P_k(A))^T A (A^{-1} - P_k(A))b$$

Inserting (38) into this expression gives

$$\sum_{i=1}^m (P_k(\lambda_i) - \frac{1}{\lambda_i})\gamma_i v_i \sum_{i=1}^m (P_k(\lambda_i) - \frac{1}{\lambda_i})\lambda_i \gamma_i v_i$$

since $P_k(A) \sum_{i=1}^m v_i = \sum_{i=1}^m P_k(\lambda_i) v_i$. Due to orthogonal vectors it simplifies to

$$\sum_{i=1}^m (P_k(\lambda_i) - \frac{1}{\lambda_i})^2 \lambda_i \gamma_i^2 = \sum_{i=1}^m q_k(\lambda_i) \tag{39}$$

So the error will depend on a polynomial applied to some of the eigenvalues of A. Note that only those eigenvalues whose eigenvectors are used to span b are important. Since the algorithm minimizes $||x - x_k||_A$, the polynomial q_k will be such that it minimizes the sum in (39). Thus as the iteration progresses it will have zeros at the relevant eigenvalues, and if A has $s < n$ distinct eigenvalues, the iteration will converge after s iterations. In floating point arithmetic the

above described convergence property is not that meaningful as it can be unclear at which point eigenvalues are distinct. It is not even guaranteed that the iteration converges after n iterations. An upper bound for the error can be derived from (39) and it can be shown that the rate of convergence depends on the condition number of the matrix A. The condition number of a matrix associated with solution of the system $Ax = b$ is defined as $\kappa(A) = ||A^{-1}|| \cdot ||A||$. An important result for convergence analysis is [9]

$$||x - x_k||_A \leq 2||x - x_0||_A \left(\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \right)^k \tag{40}$$

where $\kappa_2(A) = ||A^{-1}||_2 \cdot ||A||_2$. For a normal matrix (where $AA^T = A^T A$) it holds that $\kappa(A) = \frac{|\lambda_{max}|}{|\lambda_{min}|}$ where λ_{max} and λ_{min} are the maximum and minimum eigenvalues of A. The convergence is fast for low condition numbers and if the starting guess is good. The recursive expansion has the property that A will approach the identity matrix as the recursion steps are applied, which means that the condition number is approaching 1. Also the starting guess will improve for every recursion step. Therefore convergence should be very rapid especially after a few recursion steps.

3.5 The algorithm

The conjugate gradient method requires only one matrix-vector multiplication in every iteration, which can be made very fast for sparse matrices and vectors. The stopping criterion is based on the norm of the residual.

Algorithm 3 Conjugate gradient

$x = x_0$, $r = b - Ax$, $\rho = r^T r$, $k = 0$
while $sqrt(\rho) > tol$ **do**
  $k = k + 1$
  if k == 1 **then**
    $p = r$
  else
    $\beta = \rho/\rho_2$
    $p = r + \beta \cdot p$
  end if
  $w = Ap$;
  $\alpha = p^T r / p^T w$
  $x = x + \alpha \cdot p$
  $r = r - \alpha \cdot w$
  $\rho_2 = \rho, \rho = r^T r$
end while

4 Newton-Schulz iteration

A compelling way to calculate a matrix inverse is with Newton-Schulz iteration, as the error decreases quadratically per iteration and because of the fact that matrix multiplications, for which there are a multitude of fast, parallel implementations, constitutes a bulk of the work involved. Provided X_0 is a good starting guess for A^{-1}, the inverse can be approximated with the iteration

$$X_{i+1} = X_i(2 - AX_i) \tag{41}$$

Let $R_i = I - AX_i$ be the residual at iteration i such that $X_{i+1} = X_i(I + R_i)$, as can be seen from

$$X_{i+1} = X_i(2 - AX_i) = X_i(I + (I - AX_i)) \tag{42}$$

If $||R_0|| < 1$, where $||\cdot||$ is some matrix norm satisfying $||AB|| \leq ||A||||B||$, then X_i converges quadratically to A^{-1} [12, 13].

Proof For the residual we have the relation

$$R_m = R_0^{2m} \tag{43}$$

which can be proven by induction. At $m = 1$ we have

$$R_1 = I - AX_1 = I - AX_0(I + R_0) = I - (I - R_0)(I + R_0) = R_0^2$$

Using the assumption (43) it follows that

$$R_{m+1} = I - AX_{m+1} = I - AX_m(I + R_m) = I - (I - R_m)(I + R_m) = R_m^2 = R_0^{2(2m)}$$

As $||R_0^2|| \leq ||R_0||^2$, it holds that $||R_0^m|| \leq ||R_0||^m$ and so if $||R_0|| < 1$, X_i will converge to A^{-1} as i increases. Also, $||R_{m+1}|| = ||R_m^2|| \leq ||R_m||^2$, hence the rate of convergence is quadratic. The quadratic convergence can be used to find a suitable stopping criterion for the iteration. Errors from floating point arithmetic and for this application, truncation errors, will at some point dominate the error arising from doing a finite number of iterations with Newton-Schulz. At this point, doing more iterations will not result in increased accuracy. We can therefore stop iterating [3] when the error is no longer decreasing quadratically, that is when $||R_i||^2 < ||R_{i+1}||$. However, through numerical experimentation it was found that iterating until this condition is satisfied does not in fact give higher accuracy for $A_{approx}^{-1} b = X_i$ except for when using a very low truncation threshold. Therefore in the numerical tests the stopping criterion is set at $||R_i||_F < 0.01$, where F signifies the Frobenius norm, which provides increasing accuracy for lower truncation thresholds down to about 10^{-10} for the test matrices (see Figure 4b).

Algorithm 4 Newton-Schulz

```
Find starting guess X_0 with conjugate gradient
for i = 1,2... do
    Calculate residual R_i = I − AX_i
    Calculate X_{i+1} = X_i(I + R_i)
    Calculate e(i) = ||R_i||
    if e(i) < tol then
        break
    end if
end for
```

5 The second-order spectral projection method

At zero temperature the density matrix is given by the Heaviside step function with a step at the chemical potential μ.

$$D = \theta(\mu I - F) \tag{44}$$

All eigenvalues of the density matrix are then 0 or 1, so the density matrix D is idempotent i.e.

$$D = D^2 \tag{45}$$

The trace of the density matrix at zero temperature is equal to the number of occupied orbitals. The trace of a matrix is equal to the sum of its eigenvalues.

$$Tr[D] = \sum_{i=0}^{n} \lambda_i = n_{occ} \tag{46}$$

The second-order spectral projection method (SP2) [6] is a trace correcting algorithm, where low-order polynomials are applied to achieve correct occupation and satisfy the idempotency condition. An initial transformation is done on F, rescaling the eigenvalues to the [0,1]-interval. The polynomials $P_1(X) = X^2$ and $P_2(X) = 2X - X^2$ are then applied iteratively on X_0, pushing the eigenvalues towards 0 and 1. Let x_i be the orthogonal eigenvectors of X_0. Then X_0 can be expressed as

$$X_0 = \sum_{k=0}^{n} \lambda_k x_k x_k^T \tag{47}$$

After iteration i the approximate density matrix will look like

$$X_i = \sum_{k=0}^{n} P_i(\lambda_k) x_k x_k^T \tag{48}$$

where P_k is some combination of the polynomials P_1 and P_2. The occupation at step i is given by

$$Tr[X_i] = \sum_{k=0}^{n} P_i(\lambda_k) \tag{49}$$

Which polynomial to apply next is decided by the size of the trace. In the interval $(0, 1)$ P_1 decreases and P_2 increases the magnitude of the eigenvalues, thus correcting the trace. Convergence can be understood by the fact that P_1 and P_2 have fixed points at 0 and 1. The error is measured as the deviation from idempotency

$$e_i = ||X_i - X_i^2|| \tag{50}$$

During the first phase of iterations eigenvalues may jump around in the $[0, 1]$ interval, however when X_i is nearly idempotent the error e_i decreases at least quadratically. The stopping criteria used here is based on quadratic convergence [3].

Algorithm 5 SP2

C = 6.8872
$X_0 = \frac{\lambda_{max} I - F}{\lambda_{min} - \lambda_{max}}$
for i = 1,2,.. **do**
  if $Tr[X_i] > n_{occ}$ **then**
    $X_{i+1} = X_i^2$
    $p_i = 1$
  else
    $X_{i+1} = 2X_i - X_i^2$
    $p_i = 0$
  end if
  $e_i = ||X_i - X_i^2||$
  if $i >= 2$ and $p_i \neq p_{i-1}$ and $e_i > Ce_{i-2}$ **then**
    break
  end if
end for

6 Implementation

6.1 Linear scaling sparse matrix algebra

In each step of the recursive expansion $[X_{i-1}^2 + (I - X_{i-1})^2]X_i = X_{i-1}^2$, X_{i-1}^2 and $X_{i-1}^2 + (I - X_{i-1})^2$ need to be calculated. Achieving linear complexity of the total computation hence necessitates matrix addition and matrix multiplication to have linear complexity. Addition will be an $O(n)$ operation if the X_i matrices are consistently sparse throughout the iterations. Matrix multiplication is a bit more tricky. In the f_{FD} recursive expansion, we need to solve n equations n times with $O(n)$ complexity. This means that the conjugate gradient solution step must be done in constant time. For this reason all the operations vector addition and subtraction, vector scalar product and matrix-vector multiplication must be constant time operations. A way to meet the described complexity constraints it to utilize sparse matrix representation. In real world applications many of the matrix elements will be zero or close to zero, so that a threshold value can be used to remove small elements for increased performance without too much loss of accuracy. To achieve linear scaling however, the number of non-zeros on each row in the matrix must remain constant regardless of system size. There are many different forms of sparse matrix representation, each which have advantages depending on how the matrix is accessed. In this work, matrices were stored in the so called ELLPACK format [15].

6.2 The ELLPACK sparse matrix format

In the ELLPACK (ELL) format a matrix is represented as two smaller matrices, one for values and one for column number, as well as an array for how many non-zero elements each row contains. To determine the size of the smaller matrices requires knowledge of the maximum number of non-zero elements ($maxnnz$) in a row. If N is the system size, the size of the smaller matrices is then $N \times maxnnz$ and the size of the array is $N + 1$. For example, the matrix

$$\begin{bmatrix} 4 & 5 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

would be represented in the ELL format as

$$\text{val} = \begin{bmatrix} 4 & 5 \\ 6 & * \\ 7 & 8 \\ 9 & * \end{bmatrix} \qquad \text{col} = \begin{bmatrix} 0 & 1 \\ 1 & * \\ 1 & 2 \\ 3 & * \end{bmatrix} \qquad \text{row\_offset} = [0, 2, 3, 5, 6]$$

where * represents padding. Vectors are also stored in this format, the first row vector of the matrix is represented as

$$\text{val} = [4, 5] \qquad \text{col} = [0, 1] \qquad \text{row\_offset} = [0, 2]$$

The row_offset array helps with iterating through the matrix. A walkthrough of each element looks like

```
for (i = 0; i < n; i++)
    for (j = 0; j < row_offset[i+1]-row_offset[i]; j++)
        currentelement = i*maxnnz+j
```

So it is easy to access the matrix row by row. Column access is tricky. Luckily this can be ignored as all matrices are symmetric and rows can be read instead of columns. This format is most efficient when all rows have a similar number of non-zeros as it reduces excessive padding. There are formats without padding, such as the CSR-format where the values and columns are simply stored in arrays, which is more efficient space-wise. However, the ELL format has the advantage that the start of a row in the matrix never changes position as the matrix is manipulated and the number of non-zeros changes. This makes it easy to write to several rows at once in a multi-threaded program. To parallelize matrix-matrix multiplication for matrices in the CSR-format is a much more complex and demanding task.

6.3 Truncation

When doing matrix multiplications with sparse matrices non-zero elements tend to propagate and result in a denser matrix. To retain sparsity throughout calculations it is therefore necessary to remove small elements after every matrix multiplication and matrix-vector multiplication, according to some threshold value. This results in an error which is difficult to assess beforehand, although it is possible to find error bounds [1]. Here errors are examined with numerical tests.

6.4 Matrix multiplication

Consider the matrix-matrix multiplication $A \times B = C$. In conventional dense matrix-matrix multiplication, for each row in the A matrix, each row in the B matrix is visited one time, requiring $O(n^2)$ visitations, multiplied by $O(n)$ operations to calculate products for each visitation, resulting in an operational complexity of $O(n^3)$. Implementing matrix multiplication in the same fashion with sparse matrices may result in an $O(n^2)$ algorithm, as for each row-column multiplication one may only have to calculate a few products. Considering sparsity, visiting every column in the B matrix for every row in A is unnecessary, as for many instances there will not be overlapping elements to be multiplied. Instead for each row in A we visit only the rows in B for which there are non-zero elements on the row of A. Until the row is finished, the results are stored in their correct column positions in a temporary array of size n. To avoid having to reset the entire temporary array after each row, as a position in the array is written to, its index is saved in an indices array. To make sure the indices are written as the columns of matrix C in correct order, the indices array should be sorted. The indices array only needs to contain a maximum of $maxnnz$ elements so this does not contribute to the overall complexity. The matrix multiplication described in Algorithm (6) [15] assumes that A and B are symmetric and treats rows as columns since they are easier to access and to take advantage of caching.

Algorithm 6 Matrix-matrix multiplication, $A \times B = C$, ELLPACK

```
tempvals[n], indices[maxnnz]
for i = 0, i ¡ n do
  m = 0
  for All elements j in row i of A do
    Find the column of element j
    Go to row col[j] in B
    for For all elements k in row col[j] of B  do
      if tempvals[col[k] == 0 then
        indices[m] = col[k]
        m++
      end if
      tempvals[col[k]] += val[j]*val[k]
    end for
  end for
  Sort indices array
  for k = 0, k < m do
    C.val[i*maxnnz+k] = tempvals[indices[k]]
    C.col[i*maxnnz+k] = indices[k]
    tempvals[indices[k]] = 0
    indices[k] = 0
  end for
  Number of elements in row i of C = m
end for
```

6.5 Matrix-vector multiplication

For matrix-vector multiplication to occur in constant time, we cannot simply multiply matrix-vector like Ax, as that would require a walkthrough of all rows of A. Instead of doing the operation like $Ax = y$, we recognize that $x^T A = y^T$ for symmetric A. In this fashion only the rows of A for which x has non-zero columns are visited. So for each non-zero in x, a maximum of $maxnnz$ products are calculated, so the computational complexity is in the worst case $O(maxnnz^2)$, thus constant with regards to n.

Algorithm 7 Matrix-vector multiplication, $A \times x = y$, ELLPACK

```
tempvals[n], indices[maxnnz]
m = 0
for All elements j in x do
  Find the column of j
  Go to row col[j] in A
  for All elements k in row col[j] of A do
    if tempvals[col[k]] == 0 then
      indices[m] = col[k]
      m++
    end if
    tempvals[col[k]] += val[k]*val[j]
  end for
end for
Sort indices array
for i = 0, i ¡ m do
  y.val[i] = tempvals[indices[i]]
  y.col[i] = indices[i]
  tempvals[indices[i]] = 0
  indices[i] = 0
end for
Number of elements in y = m
```

6.6 Parallelization with OpenMP

For a program to be applicable to a demanding computational task, it needs to have sufficient parallelism to ensure utmost performance using all computer resources available. The substantial performance gain comes from parallelizing the most computationally intensive parts of the code. For this reason matrix-matrix multiplication and solution of the linear system in the conjugate gradient implementation was parallelized using OpenMP [18], a simple interface for shared-memory multiprocessing. It has a work sharing construct in which for-loop iterations are split up dynamically (*omp for schedule*{*dynamic*}) between threads, meaning that the work load can remain balanced between threads even if different loop iterations contain different amounts of work. This construct was used for the outer for-loop in matrix multiplication and in the conjugate gradient solution.

7 Performance at finite temperature

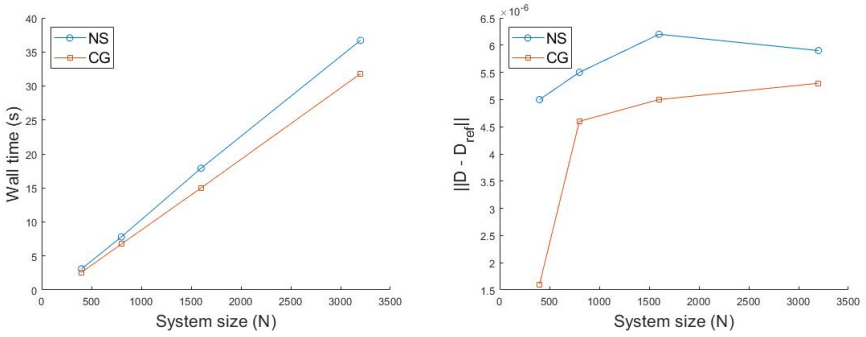
The two implementations of the recursive expansion algorithm, with Newton-Schulz and conjugate gradient respectively used to solve the linear system, are here examined. To compare the methods stopping criterias and truncation thresholds were chosen so that the resulting errors are of roughly the same

magnitude. The efficiency was measured using wall time and the number of necessary matrix multiplications they perform. As execution time may vary on different computer architectures, the number of matrix multiplications performed is often a more practical measure of how demanding a calculation is. However the difficulty of performing a matrix multiplication may vary between algorithms, which is why execution time is also a relevant measure. The Hamiltonian matrix used in the experiments was generated with random diagonal elements $10 \cdot rand$ (where $rand$ is the Matlab function generating random elements in (0,1)) and with off-diagonal elements $A(i,j) = e^{-a(i-j)^2}$, where $a = 0.01$ if not stated otherwise. This mimics the Hamiltonian of a non-metallic system or a metallic system at finite temperature [1, 6]. Tests were also performed on the Hamiltonian of a system of alkane chains which was generated by the quantum chemistry program Ergo [16], using the program *generate_alkane.cc* which is distributed with Ergo. It uses Hartree-Fock representation and a standard Gaussian basis set STO-3G. The temperature was set so that $k_bT = 0.25$ and the chemical potential at $\mu = 0.1$ in arbitrary units, which is in the HOMO-LUMO gap (the gap between the highest occupied and lowest unoccupied energy) of the alkane system.

7.1 Serial performance

7.1.1 Performance for a simulated Hamiltonian

Firstly, it can be established that the recursive expansion scheme using threshold sparse matrix algebra does indeed scale linearly with system size as can be seen in Figure 2. So it seems that intermediate matrices maintain sparsity throughout the iterations. Secondly, solution with conjugate gradient gives slightly higher accuracy in less processing time than solution with Newton-Schulz.



(a) (b)

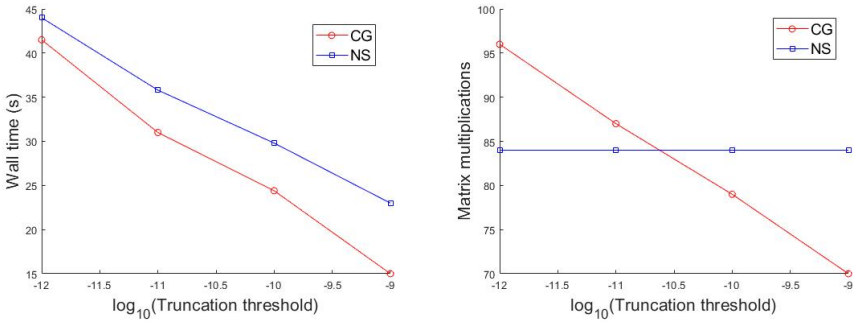
Figure 2: The recursive expansion scales linearly both for conjugate gradient and Newton-Schulz. The error plot corresponds to the execution in Figure a). The error was measured as $||D - D_{ref}||_2$ were D_{ref} was calculated with diagonalization of the F matrix. Truncation threshold for the conjugate gradient implementation was 10^{-9} and 10^{-8} for the Newton-Schulz implementation. The conjugate gradient solutions used tolerance $||r||_2 \leq 10^{-7}$, Newton-Schulz iteration used $||R||_F \leq 0.01$ as a convergence criteria.

Interesting to note from Figure 3 is the fact that the conjugate gradient implementation is faster than the Newton-Schulz implementation despite doing more matrix-matrix multiplications for lower truncation levels. This could be due the fact that the majority of the multiplications in conjugate gradient are matrix-vector multiplications were the vector is located at the same address throughout the conjugate gradient solution, allowing fast access if the address remains in cache. Using the valgrind cache profiling tool cachegrind [17] shows that matrix multiplication has a L2-cache miss rate of 10^{-3} while matrix multiplication only has a miss rate of 10^{-8} for a 1600×1600 matrix on this particular system (see Test specifications).

Figure 4a shows how many matrix multiplications are required per task in the two implementations. In the execution 10 recursions were applied (so $k = 2^{10}$ is the total expansion order), which is why calculation of X^2 and calculation of the next X in Newton-Schulz always require 10 multiplications. It can be noted that the number of multiplications for Newton-Schulz to calculate A^{-1} remains quite constant, converging after around 4 iterations regardless of truncation threshold. Conjugate gradient on the other hand requires to up to 10 iterations on average to solve for X_{i+1}. After a certain number of recursive steps the conjugate gradient convergence is faster as A is more and more well-conditioned.

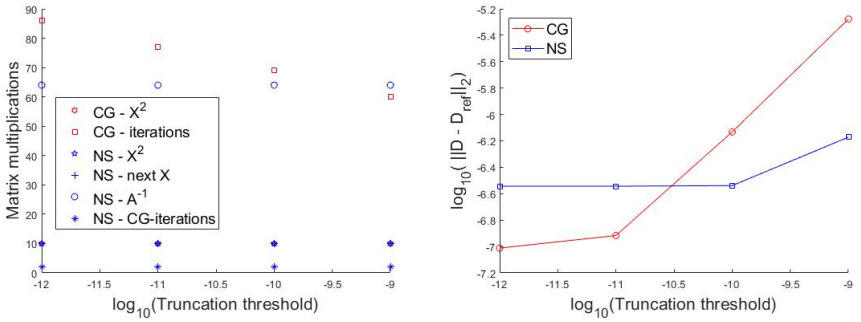
7.1.2 Performance for an alkane Hamiltonian

Tests were done on Hamiltonians with sizes $n = 674$, 1122, 1570 and 2242 corresponding to 290, 482, 674 and 962 atoms respectively. When run for a Hamiltonian of alkane chains, which is more dense than previous simulated Hamiltoni-



(a) (b)

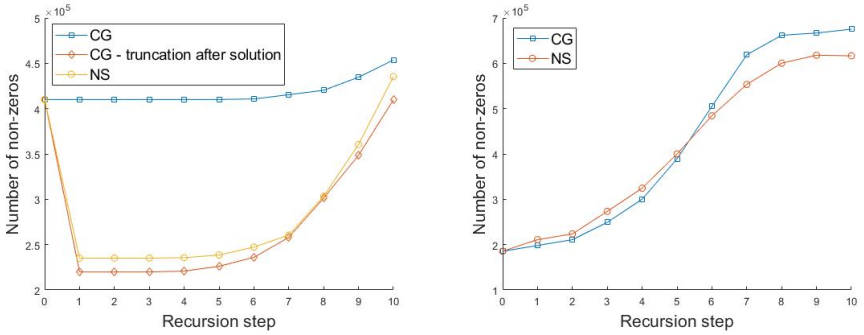
Figure 3: Comparison of execution time and number matrix multiplications in relation to truncation level for a 1600×1600-matrix. Newton-Schulz iteration used stopping criterion $||R||_F \leq 0.01$ and the conjugate gradient stopping criterion was set two magnitudes higher than the truncation threshold.



(a) (b)

Figure 4: Matrix multiplications per task and error plot corresponding to Figure 3

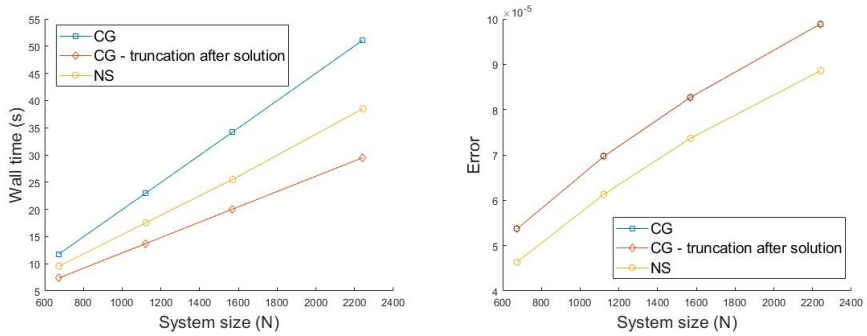
ans, the conjugate gradient implementation experiences a drop in performance. This appears to result from an inability to remove unimportant small elements in the X_i matrix when it is more dense. A remedy for this performance issue is to truncate after each conjugate gradient solution, and not only after each matrix-vector multiplication. Figure 5a displays the number of non-zeros at each step in the recursion before and after applying this truncation.



(a) Alkane Hamiltonian (b) Sparser Hamiltonian

Figure 5: Variation of the number of non-zeros in X_i after each recursion step n were $k = 2^n$ is total expansion order.

Figure 6a establishes that linear scaling of the recursive expansion holds also for a Hamiltonian arisen from a real molecular system. With the extra truncation applied in the conjugate gradient implementation, it is faster than the Newton-Schulz implementation for approximately the same level of accuracy. Figure 6b shows that the extra truncation does not seem to affect the accuracy of the conjugate gradient implementation.



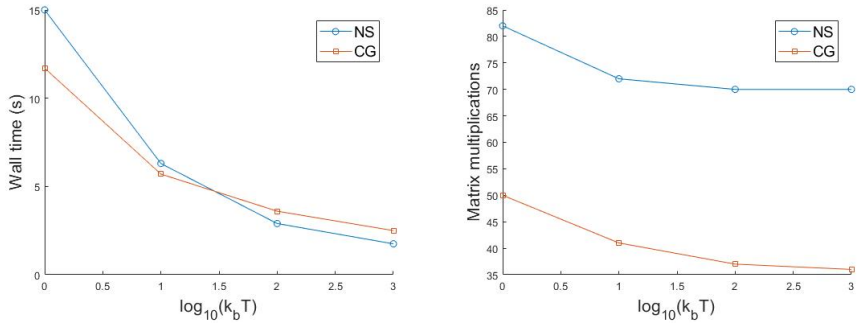
(a) (b)

Figure 6: Execution time and corresponding eigenvalue error for an alkane Hamiltonian of different sizes. Error calculated as $e = ||(\lambda - \lambda_{ref})||_2$. Threshold value for conjugate gradient was 10^{-9}, for Newton-Schulz 10^{-8}. The conjugate gradient solutions used tolerance $||r||_2 \leq 10^{-7}$, Newton-Schulz iteration used $||R||_F \leq 0.01$.

7.1.3 High temperature performance

At high temperatures the $X_0 = f_0(F) = a_0(\mu I - F) + 0.5I$ matrix approaches the matrix $0.5I$ as $a_0 = \frac{1}{4kTk_b}$ grows increasingly small. So at high temperatures

the X_i matrices become more well-conditioned. In Figure 7b the lines flatten out with increasing temperature due to the fact that both solvers converge after only one iteration. It is noteworthy that although Newton-Schulz does require more matrix multiplications, it is faster than conjugate gradient at high temperatures. This may be explained by the fact that the conjugate gradient implementation spends time reading and writing vectors from X_i, but does not do enough iterations to make up for this with its caching advantage. In the recursive expansion scheme we have that $\beta = \frac{1}{k_b T} = 4n$ where $n = 2^k$ is the total expansion order, meaning that for a higher temperature a higher expansion order is required to achieve accuracy. This fact is illustrated in 8 where the error is at first decreasing with temperature due to the matrix getting more well-conditioned but then stagnates from $k_B T = 100$ to $k_B T = 1000$ where errors from finite expansion order start to dominate. The cost of doing extra iterations to achieve a higher expansion order may be remedied by the fact that X_i is extremely well-conditioned at high temperatures.



(a) (b)

Figure 7: Execution time and number of matrix multiplications for different temperatures. Threshold was set at 10^{-10}, the conjugate gradient tolerance was set at $||r||_2 < 10^{-8}$ and Newton-Schulz tolerance at $||R||_F < 0.01$. 12 recursions were done. Tests were done for an alkane Hamiltonian of size 1122×1122 (482 atoms).

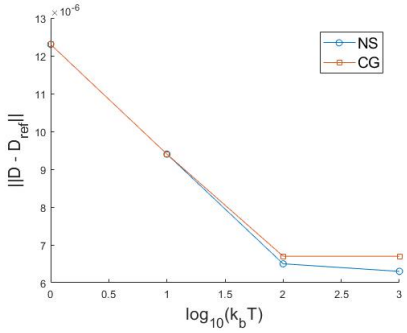
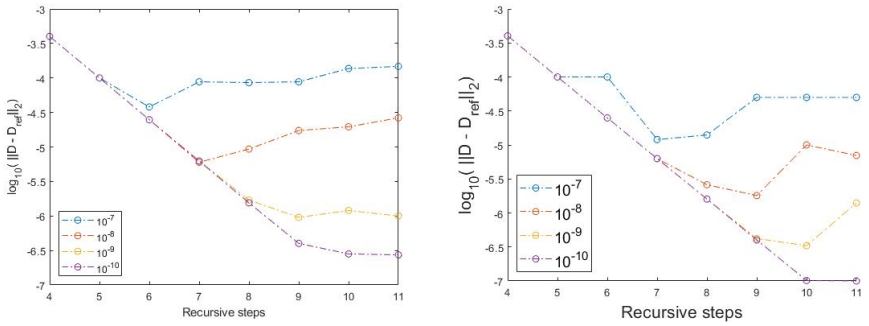


Figure 8: Errors corresponding to Figure 7.

7.2 Errors

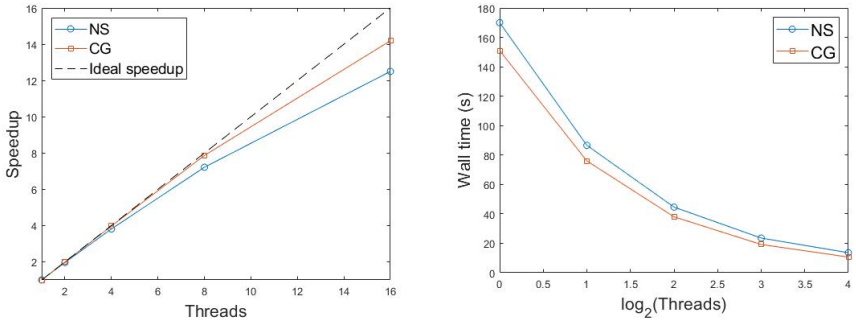
The Fermi-Dirac approximation will have two kinds of errors, namely those arising from doing truncation after every matrix multiplication and an error from a finite recursive expansion order. If the error from truncation is predominant, a higher expansion order will not give higher accuracy. To prevent doing superfluous iterations and too be able to select a suitable truncation threshold for the desired accuracy it is useful to know something about the relationship between the two errors. The following plots (Figure 9) show the norm of the error in the density matrix were the reference density matrix was calculated using diagonalization versus the number of recursions n (were $k = 2^n$ is the total expansion order) applied. It can be noted that the Newton-Schulz implementation can achieve higher accuracy than the conjugate gradient implementation for the same threshold value.



(a) Conjugate gradient (b) Newton-Schulz

Figure 9: Errors for different truncation thresholds for a simulated Hamiltonian of size $n = 400$, using $a = 0.2$ in the formula for off-diagonal elements.

7.3 Parallel performance



(a) Speedup comparison (b) Execution time comparison

Figure 10: Comparison of conjugate gradient and Newton-Schulz run in parallel for a 6400×6400-matrix.

Figure 10a demonstrates how speedup is near ideal for up to eight threads. The speedup decline for 16 threads may be due the percentage increase in execution time of serial parts of the code or to lack of enough work in relation to the overhead of thread creation. From Figure 10 it appears that the conjugate gradient implementation has better multi-threaded performance gain than the Newton-Schulz implementation. This is surprising as in the conjugate gradient implementation each thread requires storage for intermediate results from vector operations in the conjugate gradient iteration, whereas matrix-multiplication in parallel only requires extra storage for temporary values and indices. The Newton-Schulz implementation does involve more serial work, such as adding matrices and calculating matrix norms, than the conjugate gradient implementation. However, from Figure 11 it is made evident that a conjugate gradient solution does parallelize slightly better than a matrix multiplication.

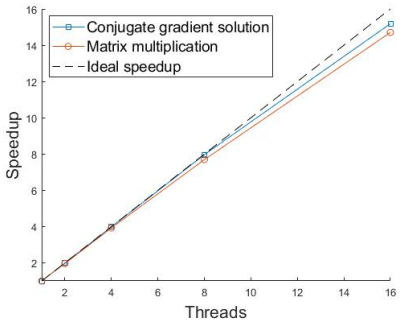
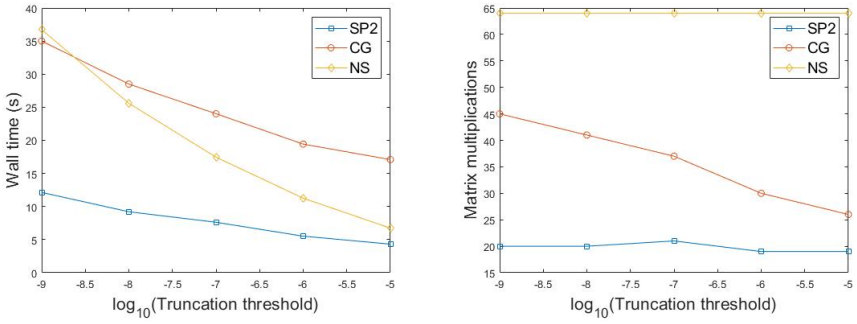


Figure 11: Speedup comparison of a conjugate gradient solution and a matrix multiplication.

8 Performance at zero temperature

8.1 Serial performance

The tests were done on a previously described alkane molecule Hamiltonian of size 1122×1122. The implementations were run for different truncation threshold values to compare for speed and accuracy. As can be seen in Figure 12, the recursive expansion seems very slow compared to SP2 while also being less accurate 13. It can be noted that Newton-Schulz is faster than conjugate gradient for high truncation thresholds. As the truncation threshold decreases, conjugate gradient requires more iterations on average to converge but this behavior cannot as clearly be seen for SP2.



(a) (b)

Figure 12: Comparison of the recursive expansion with conjugate gradient and Newton-Schulz respectively, and SP2 for execution time and number of matrix multiplications. Stopping criterion for conjugate gradient was set at two magnitudes above truncation threshold, for Newton-Schulz it was set at $||R||_F \leq 0.01$. The recursive expansion was applied 8 times, meaning $k = 2^8$ is total expansion order.

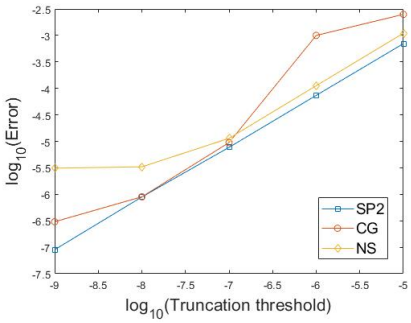


Figure 13: Errors corresponding to Figure 12, measured as $e = ||(\boldsymbol{\lambda} - \boldsymbol{\lambda_{ref}})||$, where $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda_{ref}}$ are vectors of approximate and reference eigenvalues.

8.2 Parallel performance

In Figure 14 can be seen that the conjugate gradient implementation receives a larger performance gain from parallelization than SP2, due to the conjugate gradient solution parallelizing better than matrix multiplication as seen in Figure 11.

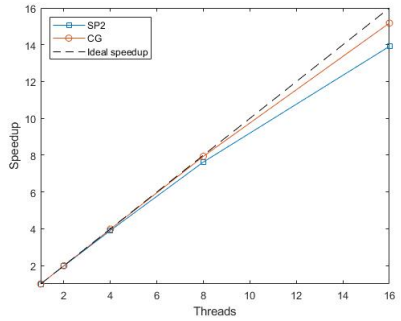


Figure 14: Speedup comparison of SP2 and the recursive expansion with conjugate gradient.

9 Comparison with diagonalization

From Figure 15 it is clear that diagonalization is faster than a serial recursive expansion for small matrices, but due to its linear scaling property it overtakes diagonalization in performance for larger matrices. A parallelized version of the recursive expansion outperforms diagonalization more quickly.

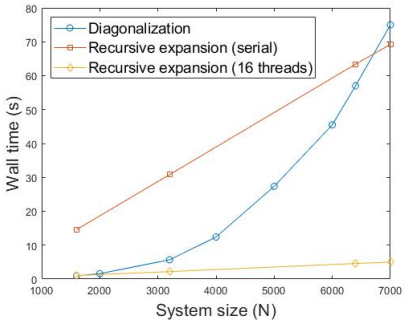


Figure 15: Performance comparison at finite temperature for a simulated Hamiltonian as specified in section 7, where the recursive expansion is implemented with conjugate gradient (truncation threshold at 10^{-9} and stopping criterion 10^{-7}). Diagonalization was performed serially in Matlab using the function *eig*, which uses Cholesky decomposition.

10 Discussion

The numerical experiments in this work were done on small systems compared to those one would like to solve in real applications. However the properties discovered by the tests should hold also for more extensive systems. The rate of convergence of the density matrix methods explored here depends on the temperature and the eigenvalue distribution of the Hamiltonian. Convergence in relation to temperature has been examined, however the eigenvalue distribution parameter has not been taken into account in this work.

11 Conclusion

The recursive expansion by Niklasson was demonstrated to scale linearly in computational cost with regards to system size. The intermediate matrices maintain sparsity throughout the calculation in the practical implementation of the method. Therefore it is potentially useful to evaluate the density matrix for large atomic systems in electronic structure calculations. For solution of the linear system arising in the recursive expansion the conjugate gradient method was found to be faster than Newton-Schulz iteration for low and at zero temperature. Importantly, it also gave better speedup when run on multiple threads. Using a vector-algebra intensive conjugate gradient implementation of the recursive expansion iteration to evaluate the density matrix at low temperature hence seems to be the best option. At high temperatures the Newton-Schulz solution was found to be faster than the conjugate gradient solution. At zero temperature the SP2 implementation performs much better than the recursive expansion, due to the low number of matrix multiplications required by SP2. However, the recursive expansion with conjugate gradient did have better speedup properties than SP2, possibly making it viable when run in parallel on a large number of cores.

12 Test specifications

All simulations were performed on a 16-core AMD Opteron Processor 6282 SE CPU with frequency 2.6 GHz. L2 and L3 cache sizes are 2048 Kb and 6144 Kb respectively.

References

[1] Rubensson, Emanuel H., *Controlling errors in recursive Fermi-Dirac operator expansions with applications in electronic structure theory*, SIAM Journal of Scientific Computing, Vol. 34, no. 1, B1-B23, 2012

[2] Niklasson, Anders M. N., *Implicit purification for temperature-dependent density matrices*, Physical Review, Vol. 68, 233104, 2003

[3] Kruchinina, Anastasia, Rudberg, Elias and Rubensson, Emanuel H., *Parameterless Stopping Criteria for Recursive Density Matrix Expansions*, Journal of Chemical Theory and Computation, Vol. 12, 5788-5802, 2016

[4] Rubensson,Emanuel H., Rudberg, Elias, Sałek, Paweł, *Chapter 12: Methods for Hartree-Fock and density functional theory electronic structure calculations with linearly scaling processor time and memory usage* from *Linear-Scaling techniques in Computational Chemistry and Physics*, Springer, 2011

[5] Niklasson, Anders M. N., *Chapter 16: Density matrix methods in linear scaling electronic structure theory*, from *Linear-Scaling techniques in Computational Chemistry and Physics*, Springer, 2011

[6] Niklasson, Anders M. N., *Expansion algorithm for the density matrix*, Physical Review, Vol. 66, 155115, 2002

[7] Liesen, Jörg and Strakos, Zdenek, *Krylov Subspace Methods: Principles and Analysis*, Oxford Scholarship Online, 2013

[8] Björk, Åke, *Numerical methods in Matrix Computations*, Springer, 2015

[9] Golub, Gene H., Van Loan, Charles F., *Matrix Computations*, The Johns Hopkins University Press, 4th edition, 2013

[10] Nocedal, Jorge and Wright, Stephen J., *Numerical Optimization*, Springer, Second Edition, 2006

[11] Mathews, John H. and Fink, Kurtis K., *Numerical Methods Using Matlab*, 4th edition, Pearson, 2004

[12] Schulz, Günther, *Iterative Berechung der reziproken Matrix*, Zeitschrift für Angewandte Mathematik und Mechanik, Vol. 13, 57-59, 1933

[13] Hotelling, Harold, *Some New Methods in Matrix Calculation*, The Annals of Mathematical Statistics, Vol. 14, p. 14, 1943

[14] Liang, WanZhen, Saravanan, Chandra, Shao, Yihan, Baer, Roi, Bell, Alexis T. and Head-Gordon, Martin, *Improved Fermi operator expansion methods for fast electronic structure calculations*, Journal of Chemical Physics, Vol. 119, 4117, 2003

[15] Mniszewski, S. M., Cawkwell, M. J., Wall, M. E., Mohd-Yusof, J., Bock, N., Germann, T. C., and Niklasson, A. M. N., *Efficient Parallel Linear Scaling Construction of the Density Matrix for Born–Oppenheimer Molecular Dynamics*, Journal of Chemical Theory and Computation, Vol. 11, 4644–4654, 2015

[16] Rudberg, E., Rubensson, E., Salek, P., Kruchinina, A., *Ergo: An open-source program for linear-scaling electronic structure calculations*, SoftwareX, Vol. 7, 107-111, 2018, `http://www.ergoscf.org/`

[17] `http://valgrind.org/docs/manual/cg-manual.html`

[18] `https://www.openmp.org/`