UPPSALA
UNIVERSITET

22031
Examensarbete 15 hp
Juni 2022

Recursion methods for solving the Schrödinger equation

Thor Lindberg
Anton Ljungar
Emy Engström

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Recursion methods for solving the Schrödinger equation

Thor Lindberg, Anton Ljungar, Emy Engström

The purpose of this study is to approximate the local density of states (LDOS) for a metal block by solving the Schrödinger equation in an efficient way. To make the code more effective different methods were implemented, for example trying to parallelize the process and to run the code solely on a GPU (Graphic Processing Unit). The conclusion that was drawn was that running the code in parallel over the different orbitals on a multicore central processing unit (CPU) is faster and thus more efficient than running it in sequential order. Running the calculations on a GPU was determined to be slower because of inefficient use of its bandwidth due to individual indexing in matrices and vectors. Further tests using block versions of the same algorithm on GPUs could be of interest because of better use of the available bandwidth. These tests were not done due to time constraints.

Handledare: Ramon Cardias Alves de Almeida, Anders Bergman
Ämnesgranskare: Teresa Zardán Gómez de la Torre
Examinator: Martin Sjödin
ISSN: 1401-5757, UPTEC F22 031

Contents

1 Introduction

Magnetism has been known to humanity since ancient times[1] and has since then had an enormous impact on humanity, with inventions such as electric motors, hard drives and magnetic resonance imaging (MRI). Magnetism continues to be a research subject at the very forefront in science[2] and as such it is a very important force to understand. The interactions between electrons in materials is what gives them their electric and magnetic properties[3] and in order to understand this, the Schrödinger equation has to be solved.

The Schrödinger equation is a partial differential equation with much significance in quantum physics and solving this equation will give the eigenenergies for a system. From these eigenenergies the local density of states can be obtained. These density of states or local density of states can be analysed to find many different properties of material exemplified by *Local Density of States for Nanoplasmonics*[4] and *Understanding Open-Circuit Voltage Loss through the Density of States in Organic Bulk Heterojunction Solar Cells*[5].

For this project the Schrödinger equation will be solved by recursion methods for the configuration of electrons in metals. A recursion method solves a computational problem with a solution that is depending solutions of smaller parts in the problem.

The institution of physics and astronomy at Uppsala University already has an old code that solves this task in the programming language Fortran. In this project the new code will be written in MATLAB which later can be translated to Fortran and implemented in the original code.

1.1 Purpose

The purpose of this Bachelor Project is to learn about and explore the uses of calculations needed to solve the Schrödinger equation in an academic setting. This project is thus in large part a learning exercise, where a significant focus will be placed on understanding and going through the mathematics and logic behind the calculations. If possible, the project is also meant to contribute to the institution's own code in order to make it more efficient.

1.2 Problem statements

The project will be focusing on the following points:

- How is the Schrödinger equation solved, using the institution's implemented algorithms?
- Is it possible to make the institution's code more effective?
- Would the code run faster on a GPU?

1.3 Limitations

The projects is based on advanced physics which require a lot of practice and understanding to be able to use. The understanding of this theory is therefore both a necessity to be able to write the required code and also a limit for how effective the code will become. When learning about a new area there is a risk to have a shortage of knowledge that could limit the possibilities.

Knowledge is also a limitation when it comes to programming languages. The already existing code that the institution currently uses is written in Fortran, but due to a restricted understanding of Fortran the code will be rewritten in MATLAB in order to work on and optimize it.

Since this project is only a Bachelor Project it is also quite limited in time. Time will therefore restrict how much optimization that can be implemented.

1.4 Outline

The first part of this project will be merely theoretical and consists of reading and understanding the theory behind the Lanczos algorithm, which the code is based upon. A few simpler examples will also be done to practice the algorithm and better comprehend the theoretical parts before the main parts of the project is initiated.

After the theoretical stage the work on the main code will begin. This means that the institution's Fortran code will be translated into MATLAB code. At first, the code to calculate the coefficients for the eigenvalues in the Schrödinger equation will be written. After that, the next task is going to be to use the coefficients from the first part to approximate the Green's functions, which is used to solve the Schrödinger equation. When the whole code is working successfully and generating a correct result the process of optimizing it will start where different methods will be implemented to try to make the code faster and more efficient.

The institution is especially interested in whether or not it would be possible for the new code to run on GPU, Graphics Processing Unit, instead of a CPU, Central Processing Unit, in order to boost performance. This will therefore be tested in the MATLAB version of the code.

2 Theory

This Bachelor project uses several methods and algorithms that are all based on theoretical physics. The theory section is divided into different parts that all contain relevant information about the project. It is necessary to understand the theory to be able to comprehend the project.

2.1 Fortran and MATLAB

The already existing code that this project is based on is written in the programming language Fortran. For this project MATLAB will also be used as the

preferred tool, and it is therefore relevant to know the differences between the two computer programming languages.

Fortran, an abbreviation of Formula Translation, is a computer programming language. It was created in the 1950s and is still often used in science and engineering, since Fortran is suitable for numerical calculations.

MATLAB is one of the programming languages that is currently taught at Uppsala University and is a shortening of Matrix Laboratory. It is often used by many in engineering and science to analyze data, develop algorithms and to create models. MATLAB is also suitable for numerical calculations, however, since it is a higher level language compared to Fortran it will run slower, but with the benefit of being easier to program and test in. A higher level language such as MATLAB will often use libraries written in lower level languages, for example MATLAB uses the LAPACK library (Linear Algebra Package) library written in Fortran.[6] This means that the same program will oftentimes run faster if written in Fortran but as an example you need to reserve memory for vectors which MATLAB does by itself. All these things that MATLAB does automatically compared to Fortran makes it slower to run but easier to program in.

2.2 Schrödinger Equation

The Schrödinger Equation is an equation that is very central in quantum mechanics. It is a linear partial differential equation which dictates the wave function. The wave function is often written as $\psi(\boldsymbol{x}, t)$ if it is time dependent and $\psi(\boldsymbol{x})$ if it is time independent.

A PDE, a partial differential equation, describes the relations between the variables and their partial derivatives in a function with multiple variables. The Schrödinger Equation calculates the wave function in a quantum-mechanical system. The equation looks different depending on which variables that are included and in which coordinate system the system takes place, but this is what the time-independent Schrödinger equation looks like[7]:

$$\boldsymbol{H}\,|\boldsymbol{\psi}\rangle = E\,|\boldsymbol{\psi}\rangle \tag{1}$$

The Schrödinger equation from equation 1 is in this project solved to determine the eigenenergies and eigenvectors from which the local density of states in metals and several material properties can be extracted.

2.3 LDOS

The density of states (DOS) is a graph which shows the number of states an electron can take for a given energy. The local density of states (LDOS) is the DOS in a finite space and is used due to properties of the local system.[8] The LDOS can be calculated for a specific energy E by $LDOS(E) = -\mathrm{Im}((G(E))/\pi)$ where $G(E)$ is the Green's function.

2.4 The chain model

In order to solve the quantum mechanical model seen in equation 1 it is converted to a chain model using sets of orthonormal base functions $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, \dots\}$ together with sets of real parameters $\{\alpha_0, \alpha_1, \alpha_2, \dots\}$ and $\{\beta_0, \beta_1, \beta_2, \dots\}$ which can describe the Hamiltonian $\boldsymbol{H}$ using equation 2 and the matrix representation $\boldsymbol{H_{TD}}$ can be seen in equation 3[9]

$$\boldsymbol{H}\left|\boldsymbol{u_n}\right\rangle = \alpha_n\left|\boldsymbol{u_n}\right\rangle + \beta_{n+1}\left|\boldsymbol{u_{n+1}}\right\rangle + \beta_n\left|\boldsymbol{u_{n-1}}\right\rangle \qquad n = 0, 1, 2, \dots \tag{2}$$

$$\boldsymbol{H_{TD}} = \begin{bmatrix} \alpha_0 & \beta_1 & 0 & \dots & 0 \\ \beta_1 & \alpha_1 & \beta_2 & 0 & \vdots \\ 0 & \beta_2 & \alpha_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \beta_n \\ 0 & \dots & 0 & \beta_n & \alpha_n \end{bmatrix} \tag{3}$$

2.5 Transformation to the chain model

2.5.1 Derivation

The transformation of $\boldsymbol{H}$ to $\boldsymbol{H_{TD}}$ is done using Lanczos method[10, 11] seen below.

$$\begin{gathered} \boldsymbol{H}\left|\boldsymbol{u}_0\right\rangle = \alpha_0\left|\boldsymbol{u}_0\right\rangle + \beta_1\left|\boldsymbol{u}_1\right\rangle \iff \\ \left\langle\boldsymbol{u}_0\right|\boldsymbol{H}\left|\boldsymbol{u}_0\right\rangle = \left\langle\boldsymbol{u}_0\right|\alpha_0\left|\boldsymbol{u}_0\right\rangle + \left\langle\boldsymbol{u}_0\right|\beta_1\left|\boldsymbol{u}_1\right\rangle \iff \\ \left\langle\boldsymbol{u}_0\right|\boldsymbol{H}\left|\boldsymbol{u}_0\right\rangle = \alpha_0\left\langle\boldsymbol{u}_0\right|\left.\boldsymbol{u}_0\right\rangle = \alpha_0 \end{gathered} \tag{4}$$

Using α_0 from equation 4 we can calculate β_1 as seen in equation 5

$$\begin{gathered} \beta_1\left|\boldsymbol{u}_1\right\rangle = \boldsymbol{H}\left|\boldsymbol{u}_0\right\rangle - \alpha_0\left|\boldsymbol{u}_0\right\rangle = (\boldsymbol{H} - \alpha_0)\left|\boldsymbol{u}_0\right\rangle \iff \\ \left\langle\boldsymbol{u}_1\right|\beta_1^*\beta_1\left|\boldsymbol{u}_1\right\rangle = (\boldsymbol{H} - \alpha_0)\left|\boldsymbol{u}_0\right\rangle \cdot \left\langle\boldsymbol{u}_1\right|\beta_1^* \iff \\ \beta_1^2 = \left\langle\boldsymbol{u}_0\right|(\boldsymbol{H} - \alpha_0)^*(\boldsymbol{H} - \alpha_0)\left|\boldsymbol{u}_0\right\rangle \implies \beta_1 = \sqrt{(\boldsymbol{H} - \alpha_0)^*(\boldsymbol{H} - \alpha_0)} \end{gathered} \tag{5}$$

Using α_0 from equation 4 and β_1 from equation 5 we can calculate $\boldsymbol{u}_1$ as seen in equation 6

$$\left|\boldsymbol{u}_1\right\rangle = \frac{(\boldsymbol{H} - \alpha_0)\left|\boldsymbol{u}_0\right\rangle}{\beta_1} \tag{6}$$

Using $\boldsymbol{u}_1$ from equation 6 we can calculate α_1 and using that we can calculate β_2 and so on, this is Lanczos method and the general forms for $\boldsymbol{u_n}$, $\boldsymbol{\alpha_n}$ and $\boldsymbol{\beta_{n+1}}$ can be seen in equations 7, 8 and 9.

$$\left|\boldsymbol{u_n}\right\rangle = \frac{(\boldsymbol{H} - \alpha_{n-1})\left|\boldsymbol{u_{n-1}}\right\rangle}{\beta_n} \qquad n = 1, 2, 3, \dots \tag{7}$$

$$\alpha_n = \langle \boldsymbol{u_n} | \, \boldsymbol{H} \, | \boldsymbol{u_n} \rangle = H_{n,n} \quad n = 0, 1, 2, \ldots \tag{8}$$

$$\beta_{n+1} = \sqrt{\left(\langle \boldsymbol{u_n} | \, (\boldsymbol{H} - \alpha_n)^* - \langle \boldsymbol{u_{n-1}} | \, \beta_n^*\right) \cdot \left((\boldsymbol{H} - \alpha_n) \, | \boldsymbol{u_n} \rangle - \beta_n \, | \boldsymbol{u_{n-1}} \rangle\right)} \quad n = 1, 2, 3 \ldots \tag{9}$$

Using Lanczos method on an atom lattice as seen in figure 1 allows the chain model to approximate the real solution with a low number of iterations n even if the atom cluster is large. This is since $\boldsymbol{H_{TD}}$ will have eigenvalues corresponding to the n largest eigenvalues of $\boldsymbol{H}$.[12]

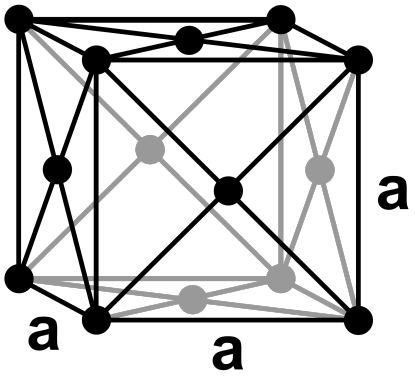


Figure 1: Copper lattice structure (FCC)

2.5.2 Code implementation

2.5.2.1 Chain model

The saving vector ψ for the base functions will contain each the base functions $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, \ldots\}$ for each orbital n_{orb}. As such the saving vector will be a $n \times n_{orb}$ matrix, since the calculation for $\boldsymbol{u_n}$ uses $\boldsymbol{u_{n-1}}$ as seen in equation 6 an additional vector *pmn* which is a $n \times n_{orb}$ matrix will be used to temporarily save ψ. How equation 9 and 6 is implemented in code can be seen below. The variable *summ* is the calculated β^2 in equation 5 and will be saved in the corresponding vector.

```
pmn = pmn-a_temp(LL)*psi;

summ = 0;
for col = 1:9
    summ = real(summ+pmn(:,col)'*pmn(:,col));
end
s = 1/sqrt(summ);

psi_temp = pmn*s;
pmn = psi;
psi = psi_temp;

s = sqrt(summ);
pmn = -pmn*s;
```

The calculation of α_n is done by summing all the contributions from the atoms in each recursion step, these contributions is calculated using equation 8. The calculations for these is done on lines 3,9 and 13 in the code below. In this step the new cumulative base function *pmn* is also calculated as seen on line 14.

```
        for m = 1:n_orb
            for L = 1:n_orb
                dum(L) = dum(L)+H(m,L)*psi(i,m);
            end
        end
...
                for m = 1:9
                    for L = 1:9
                        dum(L) = dum(L)+H((j-1)*9+m,L)*psi(nnmap,m);
                    end
                end
...
        summ = summ+real(dum(L)*conj(psi(i,L)));
        pmn(i,L) = dum(L)+pmn(i,L);
...
a_temp_temp = summ;
```

2.5.2.2 Mapping

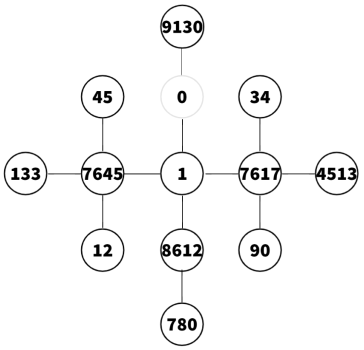


Figure 2: Example of cluster

The atom cluster is randomly generated as a text file and then read in as a matrix. The cluster can look as the one seen in figure 2 but each atom has up to 18 neighbours. A 0 means that the place for the atom is empty. Each recursion step covers the next neighbours so the mapping goes $1 \implies 1, 7645, 0, 7617, 8612 \implies 1, 7645, 0, 7617, 8612, 45, 133, 12, \cdots \implies \ldots$, this means that the entire cluster is covered in a low number of iterations. In order to keep track which atom's eigenfunction to use in the superposition a vector of zeros *izero* is created, this vector is the same length as the number of atoms in the cluster so index 1 corresponds to atom 1 and so on. When running through the recursions each atoms existing neighbours is checked and if $izero(neighbour) \neq 0$ it's eigenfunction is used in the superposition and $izero(atom)$ is set to 1 for the next recursion. This is done on line 25 for the dummyvector *idum* for *izero* and the updating is done on line 35 in section 8.3.3.

2.6 Approximation of Green's function

Green's function $G_0(E)$ can be calculated using equation 10.

$$G_0(E) = \langle \boldsymbol{u}_0 | \left(E - \boldsymbol{H_{TD}}\right)^{-1} | \boldsymbol{u}_0 \rangle \tag{10}$$

where

$$(E - \boldsymbol{H_{TD}})^{-1} = \begin{bmatrix} E-\alpha_0 & -\beta_1 & 0 & \dots & 0 \\ -\beta_1 & E-\alpha_1 & -\beta_2 & 0 & \vdots \\ 0 & -\beta_2 & E-\alpha_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -\beta_n \\ 0 & \dots & 0 & -\beta_n & E-\alpha_n \end{bmatrix} = A \tag{11}$$

From equation 10 equation 12 can be derived where $D_{n,n}$ is the determinant of A with the first n rows and columns suppressed.

$$G_0(E) = \frac{D_1(E)}{D_0(E)} \tag{12}$$

Using Laplace expansion equation 12 can be expanded into an infinite fraction as seen below.

$$G_0(E) = \frac{D_1(E)}{D_0(E)} = \frac{D_1(E)}{(-1)^2 A_{1,1} D_{1,1} + (-1)^3 A_{2,1} D_{2,1}} =$$

$$= \frac{D_1(E)}{(E-\alpha_0)D_1 - (-\beta_1)(-1)^2 A_{2,1} D_2} = \frac{D_1}{(E-\alpha_0)D_1 - \beta_1^2 D_2} = \frac{1}{E - \alpha_0 - \beta_1^2 \frac{D_2}{D_1}} =$$

$$= \frac{1}{E - \alpha_0 - \beta_1^2 \frac{D_2}{(E-\alpha_1)D_2 - \beta_2^2 D_3}} \implies$$

$$\implies G_0(E) = \cfrac{1}{E - \alpha_0 - \cfrac{\beta_1^2}{E - \alpha_1 - \cfrac{\beta_2^2}{E - \alpha_2 - \cfrac{\beta_3^2}{\ddots}}}} \tag{13}$$

In order to make a numerical approximation of 13 it is assumed that

$$\lim_{n\to\infty} \frac{\alpha_{n+1}}{\alpha_n} = 1 \quad \lim_{n\to\infty} \frac{\beta_{n+1}}{\beta_n} = 1$$

As such if a number N is chosen large enough we have that $\alpha_N = \alpha_{N-1} = \alpha$ and $\beta_{N+1} = \beta_N = \beta$, using these the infinite fraction seen in equation 13 can be truncated using the function $t(E)$ seen in equation 14. α and β is chosen using Pettifor termination (section 2.7) in order to have a strictly positive local density of states (LDOS).

$$t(E) = \frac{\beta^2}{E - \alpha - t(E)} \implies t(E) = \frac{1}{2}\left(E - \alpha \pm \sqrt{(E-\alpha-2\beta)(E-\alpha+2\beta)}\right) \tag{14}$$

Using this truncating function in equation 13 gives us the numerical approximation of Green's function as seen in equation 15

$$G_0(E) \approx \begin{cases} \cfrac{1}{E-\alpha_0-\cfrac{\beta_1^2}{E-\alpha_1-\cfrac{\beta_2^2}{E-\alpha_2-\cfrac{\beta_3^2}{\ddots \cfrac{}{E-\alpha_{N-1}-\cfrac{\beta_N^2}{E-\alpha_N-t(E)}}}}}} \\ t(E) = \frac{1}{2}\left(E-\alpha_N \pm \sqrt{(E-\alpha_N-2\beta_N)(E-\alpha_N+2\beta_N)}\right) \end{cases} \tag{15}$$

Equation 15 is easily implemented in code which can be seen below.

```
function out = bprldos(e,a,b2,LL,ebot,etop)
%Approximation of greens function for LDOS
ea = e-etop;
eb = e-ebot;
emid = 0.5*(etop+ebot);
det = ea*eb;
zoff = sqrt(det);
Qt = (e-emid-zoff)*0.5; %terminator 5

for L = LL-1:-1:1
    Qt = b2(L)/(e-a(L)-Qt);
end
out = -imag(Qt)/pi;
end
```

2.7 Pettifor termination

According to Gershgorin's Circle Theorem the eigenvalues for a $n \times n$ complex matrix with elements $\alpha_{i,j}$ lies within the union of Gershgorin discs $D(a_{i,i}, r_i(\alpha))$, where D is a disc centered on $\alpha_{i,i}$ with a radius of $r_i(\alpha) = \sum_{i \neq j} |\alpha_{i,j}|$ in the complex plane.[13] For the tridiagonal matrix $\boldsymbol{H_{TD}}$ in equation 10 Gershgorin's Circle Theorem means that the eigenvalues will be in the interval $[E_{bot}, E_{top}]$ seen below.

$$\begin{cases} E_{bot} = MIN(\alpha_n - \beta_n - \beta_{n+1}) & n = 0, 1, \ldots n-1 \\ E_{top} = MAX(\alpha_n + \beta_n + \beta_{n+1}) & n = 0, 1, \ldots n-1 \end{cases} \tag{16}$$

Choosing the Pettifor terminators α_{inf} and β_{inf} as in equation 17 guarantees that the DOS is strictly positive.[14]

$$\begin{cases} \alpha_{inf} = \frac{E_{bot}+E_{top}}{2} \\ \beta_{inf} = \frac{-E_{bot}+E_{top}}{4} \end{cases} \tag{17}$$

This method is implemented in the code since the density of states is required to be positive. α_{inf} and β_{inf} is calculated in the function Emami which can be found in the appendix 8.3.6 .

2.8 Processor

The existing code which the institution is using that this project is based on is currently being run on a CPU but a part of this project is to make the code possible to run on a GPU instead. Both of them are processing units but they have different advantages and areas of application.

2.8.1 Central processing unit

A Central processing unit (CPU) is the unit which processes the instructions which run the computer. There exists several different processors but the one which will be used in this project is the *Multithreaded Processor* which is able to execute instructions in parallel[15]. This can greatly decrease the runtime of programs where multiple calculations can be done in parallel.

2.8.2 Graphics processing unit

A Graphics processing unit (GPU) is a unit with a high bandwidth and a parallel structure which makes it efficient at running one operation on multiple data points but inefficient at running one operation in sequence on data points[16]. As such a GPU performs well for example when one operation is done on a big matrix but slow when it is done by using that operation on indexes in the matrix.

3 Method

3.1 Preparations

To prepare for the intended code a more simple code was first implemented. This code calculated coefficients, Green's function and plotted LDOS for Benzene in Matlab. This was done with the Lanczos process.

3.2 Execution

The execution of this project was divided into two different parts. The first part was to calculate coefficients for the eigenvalues in the Schrödinger equation and the second part was to use the coefficients to approximate the Greens functions that will be used to solve Schrödingers equation. After the implementation had been made the code was optimized to be more effective.

3.3 Implementations of transformation to the chain model

3.3.1 First implementation

At first the code was implemented in MATLAB with different subfunctions to calculate the coefficients. The code is reading in a file that contains a Hamiltonian $\mathcal{H}$ which is an $18x18$ matrix where each element H_{nm} is an $9x9$ matrix. The Hamiltonian describes how the atoms interact and the atoms are placed in the cluster randomly. The main code loops over the orbitals and uses different subfunctions such as *crecal* and *hop*. *Crecal* loops over the iteration steps n and calls on the subfunction *hop* which calculates $\boldsymbol{u}, \boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

3.3.2 Implementation using parallelisation

In this implementation the code was made to run over all 9 orbitals in parallel instead of in sequential on the CPU.

There is a command in Matlab called *parfor* which work like a for-loop but on parallel workers. When *parfor* is executed a parallel pool is created which enables a multi-core computer to use each core as a separate worker. The toolbox Parallel Computing Toolbox is required to be able to use *parfor*. For *parfor* to work the body of the loop must be independent and not depend on other loops since the loop iterations are executed in a nondeterministic order.[17] *Parfor* was implemented in the code for the loops to be able to run in parallel for the code to be more effective.

3.3.3 Implementation using GPU

In this implementation the code was instead made to run on the GPU but still in sequential order over the orbitals. This was done by using *gpuArray* in Matlab. *gpuArray* is an array that is being stored in GPU memory, instead of at the CPU like an usual array. A *gpuArray* is created by $G = gpuArray(X)$ where X is the original array and G is the *gpuArray* object.[18]

3.4 Calculations of Green functions

3.4.1 Implementation

The second part of the code uses the constants that are calculated in the first part to calculate the Green function and approximate the LDOS (Local Density Of States). The code uses several functions such as *Density*, *bpOPT*, *Emami* and *Bprldos*.

3.4.2 Optimisation

After implementing the code for calculations of greens function and the approximation of the LDOS it was determined to not be significant compared to the recursion in runtime. This meant that no optimisation for the calculation of Green's function was done.

3.5 Implementation of a new map

A new map for finding the neighbours of an atom was implemented in code. This was done by going through the starting atom and saving it's neighbours in a new cell. In figure 2 this would mean that cell 1 would contain atoms 1, 7617, 8612 and cell 2 would contain atoms 7617, 8612, 7645 and their neighbours. Cell 3 would contain the next layer of neighbours and so on for all recursions n as seen in figure 3. The benefit of using this method would be skipping having to go through all atoms and using the vector *izero*. An analogy to this is that instead of wandering randomly around in your neighbourhood until you find your door you would go straight to it using a map.

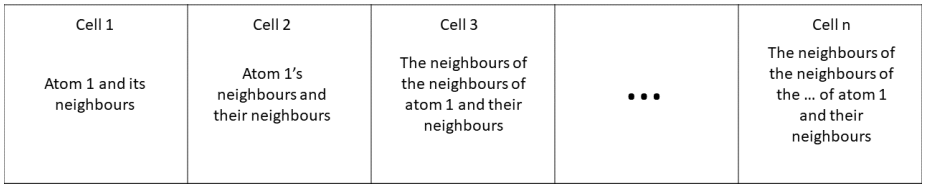


Figure 3: The new map

4 Results and Discussion

4.1 Preparation

Before the implementation of the intended code was done the same calculations were performed on the more simpler example Benzene. The Hamiltonian that was calculated was tridiagonalized and with the following shape and values:

$$H = \begin{pmatrix} a & \sqrt{2}b & 0 & 0 \\ \sqrt{2}b & a & b & 0 \\ 0 & b & a & \sqrt{2}b \\ 0 & 0 & \sqrt{2}b & a \end{pmatrix} = \begin{pmatrix} 0 & 1.4142 & 0 & 0 \\ 1.4142 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1.4242 \\ 0 & 0 & 1.4142 & 0 \end{pmatrix}$$

This means that the calculated coefficients were $a = 0$ and $b = 1$. From this were the local density of states for π electron of benzene plotted:

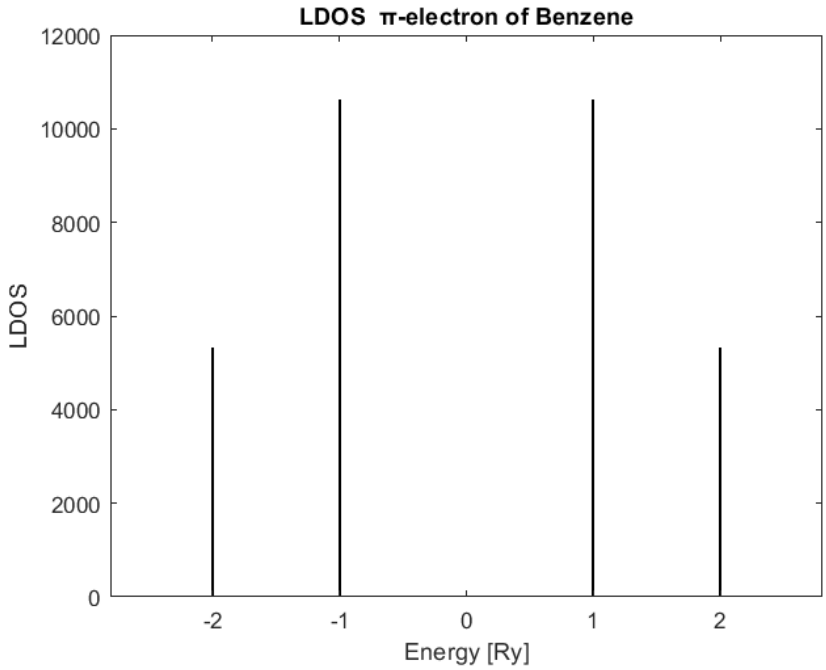


Figure 4: LDOS for Benzene

This was the correct result and the graph shows that there are 4 peaks for the energy at values -2, -1, 1 and 2 but not at 0.

4.2 Non block recursions

4.2.1 Runtimes

Table 1: Runtimes for different non block recursion codes with 21 recursions
All runs done on a computer with specification as seen in section 8.1

Version	time [s]
Ordinary	208.9
With parfor (including pool creation)	134.3
GPU	>27000

As seen in table 1 running the code in parallel over the orbitals using *parfor* on the CPU makes it run 1.56 times faster, this is since the CPU uses all of its cores which dramatically lowers the runtime.

The code running on the GPU is much slower than even the unoptimised code, after 27000 seconds the code was only on recursion 15/21 on orbital 1.

This is since indexing on a *gpuarray* is really slow and as seen below on lines 4 and 14 in the code the code uses a lot of indexing. These lines are called in total $\approx 2 \cdot 10^9$ times and in its current state the code can not be rewritten so that those lines are not using indexing. This is because of the way the mapping is done which will be shown in section 4.3.

```
1  if izero(i) ≠ 0
2      for m = 1:n_orb
3          for L = 1:n_orb
4              dum(L) = dum(L)+H(m,L)*psi(i,m);
5          end
6      end
7  end
8  for j = 2:nr
9      nnmap = nn(i,j);
10     if nnmap > 0
11         if izero(nnmap) > 0
12             for m = 1:9
13                 for L = 1:9
14                     dum(L) = dum(L)+H((j-1)*9+m,L)*psi(nnmap,m);
15                 end
16             end
17             idum(i) = 1;
18         end
19     end
20 end
```

4.2.2 Runtimes depending on recursions

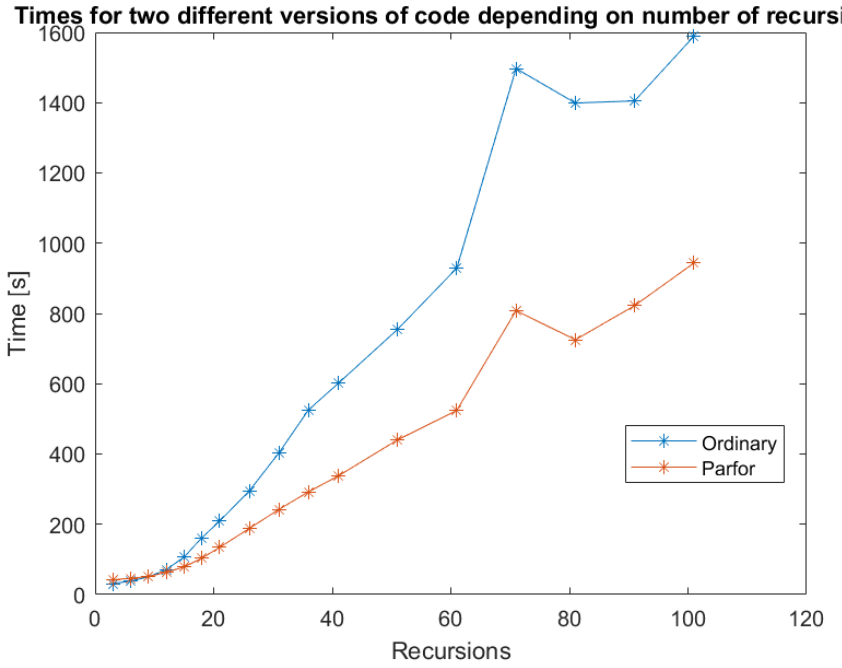


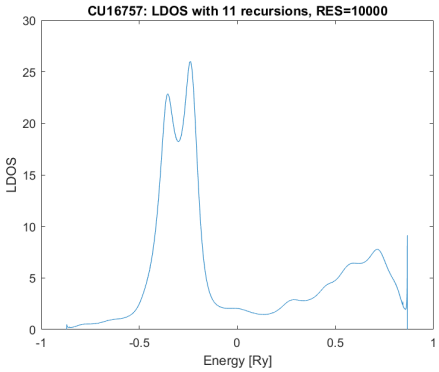
Figure 5: Runtimes for ordinary code and parfor including poolcreation

In figure 5 the benefit of using parallel computing is seen. At very low number of recursions the normal code is faster due to the time spent creating the pool used by $parfor$. At these low number of recursion a lot of detail is lost as seen in figure 6 and as such we are mostly interested in recursion $n \geq 21$ since for bigger clusters of atoms more recursion are needed to accurately approximate the LDOS. As the number of recursions grow the runtime for the code using $parfor$ is growing slower than the code not running in parallel and as such $parfor$ is better.

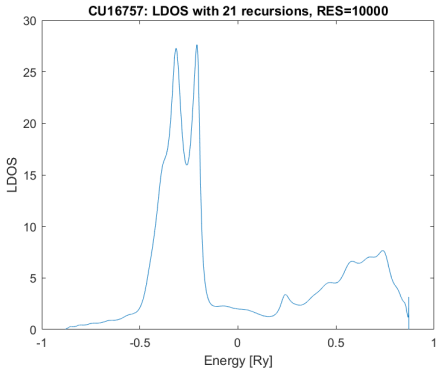
4.3 Implementation of a new map

The new map implementation was determined to not be possible. This is due to the amount of memory required to create the cell array. Since each atom has up to 18 neighbours the final cell n will have around 19^n entries, even with $n = 5$ this cell requires 45680 GB of memory which means this method is not feasible considering we want to use $n \geq 21$.

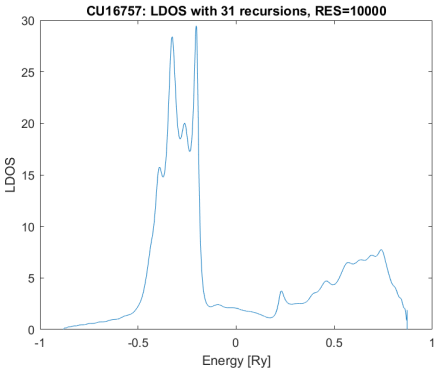
4.4 LDOS



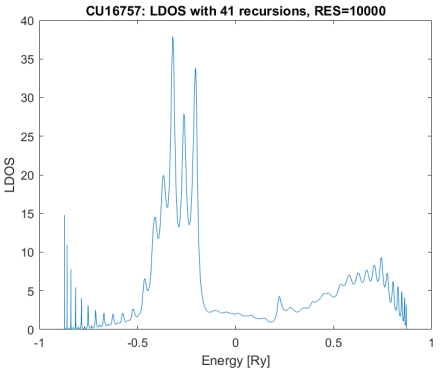
(a) 11 recursions



(b) 21 recursions



(c) 31 recursions



(d) 41 recursions

Figure 6: LDOS with different amount of recursions

In figure 6 it can be seen that as the number of recursion steps increases the resolution of the LDOS gets better. Between figure 6a and figure 6b we see the two big peaks is more defined and in figure 6c we see even more peaks starting to form. This higher resolution has a limit because of the size of our cluster. As the number of recursions increases we eventually cover the whole cluster and this creates instability in the solution. With a cluster of 16757 atoms this limit is reached by 41 recursions as seen in figure 6d where the instability is shown by the spikes in the LDOS.

4.5 Block Recursion

As seen in table 1 the GPU is very slow for the current code and this is because of the individual indexing. An alternative to this could be using block recursion

which instead calculates all orbitals in parallel using matrices. The theory for this method can be found in the appendix. This was not implemented in this project due to time constraints but since it uses matrix calculations it should be able to use more of the available bandwidth of the GPU. These matrices will be the same size as the orbitals so if there is 9 orbitals they will be 9×9. Research has shown that for matrices of this size even a single core CPU beats a GPU[19] but it could still be worth researching and testing further. This is since GPUs are in constant improvement and their architecture might become better at parallelising their workload in order to use more of their bandwidth.

5 Conclusions

5.1 Effective code

Based on the result a conclusion can be made that the code became more effective after the parallelization with *parfor*. This was a successful implementation and made the code faster which was the intention. The creation of a parpool takes some time but since the code performed a big number of iterations the small delay in the start was worth the extra time. *Parfor* makes it possible for the program to run several loops in parallel, the amount of cores is the limit, which reduces the runtime significantly especially with a big number of cores.

Another try to make the code more efficient was done by implementing a new map. The intention was to make it faster and to have fewer temporary variables and lesser indexing, but it turned out to be less effective and used far too much memory to be usable.

5.2 Implementation on GPU

The implementation on GPU in the code without block recursion showed clearly that it was not effective. *GpuArray* is very efficient on big matrices but for this code with a big amount of indexing is the GPU version much slower. So it is not possible to make this non block recursion code more efficient on a GPU.

To fully make use of a GPUs performance, another method such as the block Lanczos algorithm could perhaps be used. However it is still unclear whether or not this would be more efficient than using the normal Lanczos method on a CPU. There was not time enough to test this, however it would be interesting to know in order to make the calculations as effective as possible.

6 References

References

[1] M. Vogel, “The pull of history: human understanding of magnetism and gravity through the ages, by y. yamamoto: Scope: monograph. level: gen-

eral readership," *Contemporary physics*, vol. 59, no. 2, pp. 213–213, 2018.

[2] D. Sander, S. O. Valenzuela, D. Makarov, C. H. Marrows, E. E. Fullerton, P. Fischer, J. McCord, P. Vavassori, S. Mangin, P. Pirro, B. Hillebrands, A. D. Kent, T. Jungwirth, O. Gutfleisch, C. G. Kim, and A. Berger, "The 2017 magnetism roadmap," *Journal of Physics D: Applied Physics*, vol. 50, p. 363001, aug 2017.

[3] F. N. H. Robinson, "magnetism," *Contemporary physics*, 2019.

[4] T. V. Shahbazyan, "Local density of states for nanoplasmonics," *Phys. Rev. Lett.*, vol. 117, p. 207401, Nov 2016.

[5] S. D. Collins, C. M. Proctor, N. A. Ran, and T.-Q. Nguyen, "Understanding open-circuit voltage loss through the density of states in organic bulk heterojunction solar cells," *Advanced energy materials*, vol. 6, no. 4, pp. np–n/a, 2016.

[6] MathWorks, "Lapack in matlab." `https://se.mathworks.com/help/matlab/math/lapack-in-matlab.html`. Accessed: 2022-05-11.

[7] C. Nordling and J. Österman, *Physics Handbook for Science and Engineering*. Professional Publishing House, 2006.

[8] E. Yeganegi, A. Lagendijk, A. P. Mosk, and W. L. Vos, "Local density of optical states in the band gap of a finite one-dimensional photonic crystal," *Phys. Rev. B*, vol. 89, p. 045123, Jan 2014.

[9] R. Haydock, "The recursive solution of the schrodinger equation," *Solid State Physics*, vol. 35, pp. 215–294, 1980.

[10] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," 1950.

[11] I. Ojalvo and M. Newman, "Vibration modes of large structures by an automatic matrix-reduction method," *Aiaa Journal - AIAA J*, vol. 8, pp. 1234–1239, 07 1970.

[12] A. B. J. Kuijlaars, "Which eigenvalues are found by the lanczos method?," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 1, pp. 306–321, 2000.

[13] D. Marquis, "Gershgorin's circle theorem for estimating the eigenvalues of a matrix with known error bounds." `http://math.stmarys-ca.edu/wp-content/uploads/2017/07/David-Marquis.pdf`. Accessed: 2022-05-11.

[14] D. G. P. Bernhard Seiser and R. Drautz, "Analytic bond-order potential expansion of recursion-based methods." `https://journals-aps-org.ezproxy.its.uu.se/prb/pdf/10.1103/PhysRevB.87.094105`. Accessed: 2022-05-11.

[15] A. González, F. Latorre, G. Magklis, and I. ebrary, *Processor microarchitecture: an implementation perspective*, vol. 12;12.;. San Rafael, Calif.?: Morgan & Claypool, 1st ed., 2011;2010;.

[16] H. Kim and I. ebrary, *Performance analysis and tuning for general purpose graphics processing units (GPGPU)*, vol. 20.;20;. San Rafael, Calif.: Morgan & Claypool, 2012.

[17] MathWorks, "parfor." `https://se.mathworks.com/help/parallel-computing/parfor.html`. Accessed: 2022-05-16.

[18] MathWorks, "gpuarray." `https://se.mathworks.com/help/parallel-computing/gpuarray.html`. Accessed: 2022-05-17.

[19] Z. Huang, N. Ma, S. Wang, and Y. Peng, "Gpu computing performance analysis on matrix multiplication," *The Journal of Engineering*, vol. 2019, no. 23, pp. 9043–9048, 2019.

[20] T. Ozaki, "Note on recursion methods," pp. 1–27, 2003.

7 Populärvetenskaplig sammanfattning

Magnetism har använts av människan sedan urminnes tider och fortsätter än idag vara relevant, utan magnetismen skulle inte vårt samhälle se ut som det gör idag. För att fortsätta utveckla nya samt förbättra nuvarande teknologier behöver materials egenskaper såsom magnetism förstås. Detta görs genom att analysera atomerna i material, atomer i alla material är placerade i mönster och kommer alltid att påverka varandra i någon utsräckning.

I metaller bildar atomerna ett kristallikt mönster där varje atoms grannar kommer ha någon form av inverkan på denna. Baserat på hur dessa atomer influerar de atomer nära dem kan Schrödingers ekvation lösas. Eftersom det finns väldigt många atomer så kommer beräkningarna för att få fram detta att bli väldigt krävande, därför används datorer för att göra detta. Den metod som används är rekursiv, vilket innebär att metoden löser ett problem med en lösning som är beroende av lösningar av mindre delar av problemet. Målet med detta projekt var att skriva en programmeringskod på datorn som löser detta och därefter hitta sätt att effektivisera den koden för att göra den snabbare.

De största effektiviseringsförsök som gjordes var att parallellisera koden, alltså så att flera delar kan köra samtidigt, samt att anpassa det på ett sätt som går att köras på en GPU vilket är en grafisk processor som klarar av att göra många saker samtidigt. Slutsatsen som kunde dras var att koden var mycket snabbare när processerna kördes parallellt vilket då innebar att flera saker gjordes samtidigt istället för efter varandra. Försöket att implementera kod som kördes på en GPU var inte lika framgångsrikt och gjorde istället koden långsammare.

8 Appendix

8.1 Computer specifications

```
Processor Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz   2.81 GHz
GPU  NVIDIA GeForce GTX 1060 6GB
Driver version  512.15
Installed RAM 8,00 GB
RAM speed 2666 MHz
Edition Windows 10 Home
Version 21H2
MATLAB version  R2021b
```

8.2 Block recursion method

Instead of calculating each orbital by itself, all can be calculated at the same time using the block Lanczos algorithm.[20] Instead of using the numbers α_n and β_n, $\underline{A}_n$ and $\underline{B}_n$ are used, which are matrices $M \times M$ in size, where M is the number of valence orbitals of the starting site. The vectors $|U_n) = (|L_{n1}\rangle, |L_{n2}\rangle, \ldots, |L_{nM}\rangle)$ constitute the Lanczos basis states which are orthonormal and represent the arrangement of neighbouring atoms to the starting site. To find the $\underline{A}_n$, $\underline{B}_n$ and U_n the following calculations are made:

$$|U_0) = (|i1\rangle, |i2\rangle, \ldots, |iM_i\rangle) \tag{18}$$

$$\underline{A}_n = (U_n \left|\hat{H}\right| U_n) \tag{19}$$

$$|r_n) = \hat{H}|U_n) - |U_{n-1})^t \underline{B}_n - |U_n)\underline{A}_n \tag{20}$$

$$(\underline{B}_{n+1})^2 = (r_n|r_n) \tag{21}$$

$$(\underline{\lambda}_n)^2 = {}^t\underline{V}_n (\underline{B}_{n+1})^2 \underline{V}_n \tag{22}$$

$$\underline{B}_{n+1} = \underline{\lambda}_n \, {}^t\underline{V}_n \tag{23}$$

$$(\underline{B}_{n+1})^{-1} = \underline{V}_n \underline{\lambda}_n^{-1} \tag{24}$$

$$|U_{n+1}) = |r_n)(\underline{B}_{n+1})^{-1} \tag{25}$$

This is a modified version of the Lanczos block algorithm to prevent numerical instability. The conventional method for calculating $\underline{B}_{n+1}$ would be as follows:

$$\underline{B}_{n+1} = \underline{V}_n \underline{\lambda}_n \, {}^t\underline{V}_n \tag{26}$$

$$(\underline{B}_{n+1})^{-1} = \underline{V}_n \underline{\lambda}_n^{-1} {}^t\underline{V}_n \tag{27}$$

From the values calculated above, the Green's function can be calculated. This is done similar to the non-block method:

$$\underline{G}^{L}_{00}(Z) = \cfrac{1}{Z\underline{I} - \underline{A}_0 - \cfrac{{}^{t}\underline{B}_1\underline{B}_1}{Z\underline{I} - \underline{A}_1 - \cfrac{{}^{t}\underline{B}_2\underline{B}_2}{Z\underline{I} - \underline{A}_2 - \cfrac{{}^{t}\underline{B}_3\underline{B}_3}{\ddots}}}} \tag{28}$$

To terminate this continuous fraction the following terminator is used:

$$t(Z) = [Z - a - b^2 t(Z)]^{-1} = \frac{1}{b}\left[\frac{Z-a}{2b} - i\sqrt{1 - \left(\frac{Z-a}{2b}\right)^2}\right] \tag{29}$$

In the above terminator, it is assumed that the diagonal elements of $\underline{A}_\infty$ and $\underline{B}_\infty$ each are the same and are called a and b respectively.

8.3 Matlab code

8.3.1 Main code

```
%Solves and approximaties LDOS

%% Solves for a and b
fclose('all'); %close all files

%reading of files
files = untar('recurbundle.tar');
data_real = readmatrix(files{1},'CommentStyle','nn');
data_im = readmatrix(files{2},'CommentStyle','nn');
nn = readmatrix(files{3});
nn = nn(1:16757,:); %Remove the extra lines

H = data_real + data_im*1i; %Hamiltionian

kk = length(nn); %Total number of atoms
LLmax = 21; %Number of iterations
n_orb = 9; %Number of orbitals

a = zeros(n_orb,LLmax); %Saving vector
b2 = zeros(n_orb,LLmax); %Saving vector

j = nn(1,1); %Takes the atom we calculate from
parfor L = 1:n_orb %number of orbitals

%error search
%    disp(['Orbital number ' num2str(L)])

    izero = zeros(kk,1); %Vector for mapping
```

```
29      izero(j) = 1;
30
31      psi = zeros(kk,n_orb); %eigenvector
32      psi(j,L) = 1;
33
34      pmn = zeros(kk,n_orb); %eigenvector
35
36      a_temp = zeros(LLmax,1); %temp for a
37      a_temp(end) = 0;
38
39      b2_temp = zeros(LLmax,1); %temp for b
40      b2_temp(1) = 1;
41      [a_temp, b2_temp,izero] = ...
            crecal(a_temp,b2_temp,kk,n_orb,LLmax,H,psi,nn,pmn,izero);
42      for LL = 1:LLmax
43          a(L,LL) = a_temp(LL);
44          b2(L,LL) = b2_temp(LL);
45      end
46  end
47
48  %% Calculates density of states
49  [E,DENS] = density(a,b2,n_orb,LLmax); %calculates density
50
51  %Plotting of LDOS
52  plot(E,sum(DENS,2))
53  title('LDOS')
54
55  %time 188 s for booting and doing program
```

8.3.2 Crecal

```
1   function [a_temp, b2_temp,izero] = ...
        crecal(a_temp,b2_temp,kk,n_orb,LLmax,H,psi,nn,pmn,izero)
2   %Calculates b2
3   summ = b2_temp(1);
4
5   nm1 = LLmax-1;
6   for LL = 1:nm1
7
8   %error search
9   %    disp(['loop number ' num2str(LL)])
10
11      [a_temp(LL),pmn,izero] = hop(kk,izero,n_orb,H,psi,nn,pmn);
12      b2_temp(LL) = summ;
13      pmn = pmn-a_temp(LL)*psi;
14
15      summ = 0;
16      for col = 1:9
17          summ = real(summ+pmn(:,col)'*pmn(:,col));
18      end
19      s = 1/sqrt(summ);
20
21      psi_temp = pmn*s;
22      pmn = psi;
```

```
    psi = psi_temp;

    s = sqrt(summ);
    pmn = -pmn*s;
end
b2_temp(LLmax) = summ;
end
```

8.3.3 Hop

```
function [a_temp_temp,pmn,izero] = hop(kk,izero,n_orb,H,psi,nn,pmn)
%Calculates a
idum = zeros(kk,1);
v = zeros(kk,n_orb);
for i = 1:kk
    idum(i) = izero(i);
    dum = zeros(9,1);
    nr = 19;
    if izero(i) ≠ 0
        for m = 1:n_orb
            for L = 1:n_orb
                dum(L) = dum(L)+H(m,L)*psi(i,m);
            end
        end
    end
    for j = 2:nr
        nnmap = nn(i,j);
        if nnmap > 0
            if izero(nnmap) > 0
                for m = 1:9
                    for L = 1:9
                        dum(L) = dum(L)+H((j-1)*9+m,L)*psi(nnmap,m);
                    end
                end
                idum(i) = 1;
            end
        end
    end
    for L = 1:n_orb
        v(i,L) = dum(L);
    end
end
summ = 0;
for i = 1:kk
    izero(i) = idum(i);
    for L = 1:n_orb
        dum(L) = v(i,L);
        summ = summ+real(dum(L)*conj(psi(i,L)));
        pmn(i,L) = dum(L)+pmn(i,L);
    end
end
a_temp_temp = summ;
end
```

8.3.4 Density

```
function [ene,dens] = density(a,b2,n_orb,LLmax)
%Calculates the density of states

%Constants
npts = 10000;

%Vectors
alpha_inf = zeros(n_orb);
beta_inf = zeros(n_orb);
edge2 = zeros(n_orb,10);
width2 = zeros(n_orb,10);
weight2 = zeros(n_orb,10);
am2 = zeros(n_orb,LLmax);
bm2 = zeros(n_orb,LLmax);
nb2 = zeros(n_orb);

for orb = 1:n_orb
    aa = a(orb,:);
    bb = b2(orb,:);
    sqbb = sqrt(b2(orb,:));

    if or(orb==1,orb==10) %widens the b state for other orbitals
        b2 = 1.025*b2;
    end

    %Temp vectors
    am = zeros(LLmax,1);
    bm = zeros(LLmax,1);
    edge = zeros(10,1);
    width = zeros(10,1);
    weight = zeros(10,1);

    [am(1),bm(1)] = bpOPT(LLmax,aa,sqbb,LLmax-1);

    if or(orb==1,orb==10) %Widens bm for other states
        bm = 1.01*bm;
    end

    %temp constants
    alpha_inf(orb) = am(1);
    beta_inf(orb) = bm(1);
    nb = 1;
    edge(1) = am(1)-2*bm(1);
    width(1) = 4*bm(1);
    weight(1) = 1;
    nb2(orb) = nb;
    am = aa;
    bm = bb;

    for k = 1:nb %nb = 1 in this case
        a1 = edge(k);
        a2 = edge(k)+width(k);
        edge2(orb,k) = edge(k);
        width2(orb,k) = width(k);
```

```
55          weight2(orb,k) = weight(k);
56      end
57
58      if orb == 1 %Orb=1 means starting guess for high/low eigenvalue
59          emin = a1;
60          emax = a2;
61      else
62          emin = min([emin,a1]);
63          emax = max([emax,a2]);
64      end
65
66      if nb > 0 %nb = 1 in this case
67          for l = 1:LLmax
68              am2(orb,l) = am(l);
69              bm2(orb,l) = bm(l);
70          end
71      else
72          for l = 1:LLmax
73              am2(orb,l) = 0;
74              bm2(orb,l) = 0;
75          end
76      end
77  end
78
79  dens = zeros(npts,n_orb); %density vector
80  ene = linspace(emin,emax,npts); %vector for eigenvalues
81  for eidx = 1:npts
82      for orb = 1:n_orb
83          nb = nb2(orb);
84          if nb > 0
85              for l = 1:LLmax
86                  aa(l) = a(orb,l);
87                  bb(l) = b2(orb,l);
88                  am(l) = am2(orb,l);
89                  bm(l) = bm2(orb,l);
90              end
91              edge = zeros(nb,1);
92              width = zeros(nb,1);
93              weight = zeros(nb,1);
94              for k = 1:nb
95                  edge(k) = edge2(orb,k);
96                  width(k) = width2(orb,k);
97                  weight(k) = weight2(orb,k);
98              end
99              dens(eidx,orb) = bprldos(ene(eidx),aa,bb,LLmax,...
100                 edge(1),edge(1)+width(1));
101         else
102             dens(eidx,orb) = 0;
103         end
104     end
105 end
106 end
```

8.3.5 bpOPT

```
function [ainf,binf] = bpOPT(ll,a,b,n)
%Calculates optimal values for terminators using Pettifor's ...
    termination
ndime=ll; %Size of H_TRI
ifail = 0;
eps = 1e-5;
jiter = 0; %iteration counter
bmax = max(b);
bmin = min(b);
ainf = a(n);

%saving vectors
az = zeros(ll,1);
bz = zeros(ll,1);

bm = eps+1; %temp value

while bm > eps
    jiter = jiter+1;
    az(1) = 0.5*(a(1)-ainf);
    for i = 2:n-1
        az(i) = 0.5*(a(i)-ainf);
        bz(i) = 0.5*b(i);
    end
    az(n) = a(n)-ainf;
    bz(n) = 1/sqrt(2)*b(n);
    [bmax,bmin,emamifail] = emami(ndime,az,bz,n);
    bm = bmax+bmin;
    bm = abs(bm);
    ainf = ainf+bmax+bmin;
    if jiter > 300
        ifail = 1;
        disp('bpOPT has failed')
        break
    end
end
binf = (bmax-bmin)/2;
if emamifail == 1
    disp('Emami has failed')
end
if ifail == 1
    disp('bpOPT has failed')
end
end
```

8.3.6 Emami

```
function [emax,emin,ifail] = emami(ndime,az,bz,n)
%Obtains the maximum and minimum eigenvalues of H_TRI
%Initial guesses
emax0 = -1e6;
emin0 = 1e6;

```

```
%saving vectors
a = zeros(ndime,1);
b = zeros(ndime,1);
for i = 1:n
    a(i) = az(i);
    b(i) = bz(i);
end
b(1) = 0;
b(n+1) = 0;
for i = 1:n
    x1 = a(i)+abs(b(i))+abs(b(i+1));
    x2 = a(i)-abs(b(i))-abs(b(i+1));
    if emax0 ≤ x1
        emax0 = x1;
    end
    if emin0 > x2
        emin0 = x2;
    end
end

ifail = 0;
relfeh = 2^(-39); %incase p = 0

eps = 1e-6;
istop = 0; %iteration counter

emax = emax0; %starting guess
emin = emin0; %starting guess

%Calculation of emax

dele = eps+1; %temp value
while dele > eps
    E = (emax+emin)/2;
    istop = istop+1;
    if istop > 50
        ifail = 1;
        disp('emax has failed')
        break
    end
    num = 0;
    p = a(1)-E;
    if p < 0
        num = num+1;
    end
    for i = 2:n
        if p == 0
            p = a(i)-E-abs(b(i))/relfeh;
            if p < 0
                num = num+1;
            end
        else
            p = a(i)-E-b(i)^2/p;
            if p < 0
                num = num+1;
            end
        end
```

```
64      end
65      if num == n
66          emax = E;
67      end
68      if num < n
69          emin = E;
70      end
71      dele = (emax-emin)/((emax+emin)/2);
72      dele = abs(dele);
73  end
74  E1 = E;
75
76  %Calculation on emin
77  istop = 0;
78  emax = E1; %initial guess
79  emin = emin0; %initial guess
80
81  dele = eps+1; %temp value
82
83  while dele > eps
84      E = (emax+emin)/2;
85      istop = istop+1;
86      if istop > 50
87          ifail = 1;
88          disp('emin has failed')
89          break
90      end
91      num = 0;
92      p = a(1)-E;
93      if p < 0
94          num = num+1;
95      end
96      for i = 2:n
97          if p == 0
98              p = a(i)-E-abs(b(i))/relfeh;
99              if p < 0
100                 num = num+1;
101             end
102         else
103             p = a(i)-E-b(i)^2/p;
104             if p < 0
105                 num = num+1;
106             end
107         end
108     end
109     if num == 0
110         emin = E;
111     end
112     if num > 0
113         emax = E;
114     end
115     dele = (emax-emin)/((emax+emin)/2);
116     dele = abs(dele);
117 end
118 E2 = E;
119
120 %assign emax and emin
```

```
121  emax = E1;
122  emin = E2;
123  end
```

8.3.7 Bprldos

```
1   function out = bprldos(e,a,b2,LL,ebot,etop)
2   %Approximation of greens function for LDOS
3   ea = e-etop;
4   eb = e-ebot;
5   emid = 0.5*(etop+ebot);
6   det = ea*eb;
7   zoff = sqrt(det);
8   Qt = (e-emid-zoff)*0.5; %terminator 5
9
10  for L = LL-1:-1:1
11      Qt = b2(L)/(e-a(L)-Qt);
12  end
13  out = -imag(Qt)/pi;
14  end
```

8.3.8 GPU code

```
1   fclose('all');
2   files = untar('recurbundle.tar','Stage 1');
3   data_real = readmatrix(files{1},'CommentStyle','nn');
4   data_im = readmatrix(files{2},'CommentStyle','nn');
5   nn = readmatrix(files{3});
6   nn = nn(1:16757,:);
7
8   H = data_real + data_im*1i;
9
10  kk = length(nn);
11  LLmax = 21;
12  n_orb = 9;
13
14  a = zeros(n_orb,LLmax,"gpuArray");
15  b2 = zeros(n_orb,LLmax,"gpuArray");
16
17  j = nn(1,1);
18  for L = 1:1 %number of orbitals
19      izero = zeros(kk,1,"gpuArray");
20      izero(j) = 1;
21
22      psi = zeros(kk,n_orb,"gpuArray");
23      psi(j,L) = 1;
24
25      pmn = zeros(kk,n_orb,"gpuArray");
26
27      a_temp = zeros(LLmax,1,"gpuArray");
28      a_temp(end) = 0;
```

```
29
30      b2_temp = zeros(LLmax,1,"gpuArray");
31      b2_temp(1) = 1;
32      disp(['orbital number ' num2str(L)])
33      [a_temp, b2_temp,izero] = crecal(a_temp,b2_temp,kk,n_orb,...
34          LLmax,H,psi,nn,pmn,izero);
35      for LL = 1:LLmax
36          a(L,LL) = a_temp(LL);
37          b2(L,LL) = b2_temp(LL);
38      end
39  end
40
41  function [a_temp, b2_temp,izero] = ...
        crecal(a_temp,b2_temp,kk,n_orb,...
42      LLmax,H,psi,nn,pmn,izero)
43
44  summ = b2_temp(1);
45
46  nm1 = LLmax-1;
47  for LL = 1:nm1
48      disp(['loop number ' num2str(LL)])
49      [a_temp(LL),pmn,izero] = hop(kk,izero,n_orb,H,psi,nn,pmn);
50      b2_temp(LL) = summ;
51      pmn = pmn-a_temp(LL)*psi;
52
53      summ = 0;
54      for col = 1:n_orb
55          summ = real(summ+pmn(:,col)'*pmn(:,col));
56      end
57      s = 1/sqrt(summ);
58
59      psi_temp = pmn*s;
60      pmn = psi;
61      psi = psi_temp;
62
63      s = sqrt(summ);
64      pmn = -pmn*s;
65  end
66  b2_temp(LLmax) = summ;
67  end
68
69  function [a_temp_temp,pmn,izero] = hop(kk,izero,n_orb,H,psi,nn,pmn)
70  idum = zeros(kk,1,"gpuArray");
71  v = zeros(kk,n_orb,"gpuArray");
72  for i = 1:kk
73      idum(i) = izero(i);
74      dum = zeros(n_orb,1,"gpuArray");
75      nr = 19;
76      if izero(i) ≠ 0
77          for m = 1:n_orb
78              for L = 1:n_orb
79                  dum(L) = dum(L)+H(m,L)*psi(i,m);
80              end
81          end
82      end
83      for j = 2:nr
84          nnmap = nn(i,j);
```

```
85          if nnmap > 0
86              if izero(nnmap) > 0
87                  for m = 1:9
88                      for L = 1:9
89                          dum(L) = dum(L)+H((j-1)*9+m,L)*psi(nnmap,m);
90                      end
91                  end
92                  idum(i) = 1;
93              end
94          end
95      end
96      for L = 1:n_orb
97          v(i,L) = dum(L);
98      end
99  end
100 summ = 0;
101 for i = 1:kk
102     izero(i) = idum(i);
103     for L = 1:n_orb
104         dum(L) = v(i,L);
105         summ = summ+real(dum(L)*conj(psi(i,L)));
106         pmn(i,L) = dum(L)+pmn(i,L);
107     end
108 end
109 a_temp_temp = summ;
110 end
```