Development and Implementation of numerical methods for calculations of the electronic properties of carbon nano-systems on accelerated architectures

Keshvi Tuteja

Masterarbeit in Physik
angefertigt im Physikalischen Institut

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität
Bonn

August 2022

I hereby declare that this thesis was formulated by myself and that no sources or tools other than those cited were used.

Bonn, 24-08-2022 .....
Date

.....
Signature

1. Gutachter: Prof. Stefan Krieg
2. Gutachterin: Prof. Estela Suarez

Acknowledgements

I would like to thank my supervisors Prof. Stefan Krieg and Prof. Estela Suarez for providing me such an incredible opportunity to work at Forschungszentrum Jülich and for their guidance and support. Thank you for giving me the freedom to follow different approaches and for giving valuable suggestions whenever I faced any issues.
I express my sincere gratitude towards Marcel Rodekamp and Dr. Evan Berkowitz for being so patient and helpful whenever I had any doubts or needed any advice regarding the code, and for striking interesting discussions during the code review meetings, I got to learn so much during those meetings. Their dedication towards their work inspires me to work harder.
I would also like to thank Prof. Thomas Luu for giving pivotal suggestions in group meetings and the other group members for providing me a positive workspace. I couldn't have asked for a better place.
This acknowledgement would be incomplete without thanking my parents for always having my back and my sister who has been my support system since day one. I am also grateful for my friends for always being there.

Gratitude is all I have!

Contents

1 Introduction **1**

2 Formalism **3**
2.1 Hubbard Model . . . 3
2.1.1 Hubbard Hamiltonian . . . 3
2.1.2 Particle-hole Transformation . . . 4
2.2 Path Integral Formalism and Discretization . . . 5
2.3 Markov Chain Monte Carlo . . . 7
2.3.1 Metropolis-Hastings Algorithm . . . 7
2.3.2 Hamiltonian Monte Carlo . . . 8
2.4 Nanosystem Simulation Library (NSL) Setup . . . 9
2.4.1 Source Code Implementation . . . 10
2.5 Acceleration . . . 12

3 Implementation **15**
3.1 Computing and Implementing M, $M^\dagger$, $MM^\dagger$ and $M^\dagger M$ . . . 15
3.1.1 Diagonal Discretization . . . 15
3.1.2 Exponential discretization . . . 20
3.2 Computing and Implementing logDetM . . . 23
3.2.1 LU decomposition . . . 23
3.2.2 Diagonal Discretization . . . 24
3.2.3 Exponential Discretization . . . 26

4 Tests **29**
4.1 Testing `M()`, `M_dense()`, `Mdagger()`, `MMdagger()` and `MdaggerM()` . . . 29
4.1.1 `M()` . . . 29
4.1.2 `M_dense()` . . . 30
4.1.3 `Mdagger()` . . . 31
4.1.4 `MMdagger()` . . . 32
4.1.5 `MdaggerM()` . . . 33
4.2 Testing `logDetM()` . . . 33
4.2.1 Diagonal Discretization . . . 34
4.2.2 Exponential Discretization . . . 36
4.3 Accuracy . . . 39

5 Benchmarks **41**
5.1 Method . . . 41
5.1.1 `std::chrono` . . . 41
5.1.2 JURECA-DC [2] . . . 42
5.2 Results . . . 42
5.2.1 Diagonal discretization versus Exponential discretization . . . 42
5.2.2 CPU versus GPU . . . 46
5.2.3 Summary of Execution Times . . . 50

6 Conclusion and Outlook **53**

A Appendix A **55**
A.1 Building : Steps for compilation of NSL on Jureca DC module . . . 55
A.2 Testing . . . 55
A.3 Benchmarking . . . 56

B Appendix B **57**
B.1 Benchmark results for Square and Cube Lattices . . . 57

List of Figures **75**

Listings **77**

List of Tables **79**

CHAPTER 1

Introduction

In this era of industrialization, it is necessary to have a good understanding of materials for further predictions and advancement in condensed matter physics. There are various approximate models which help us do that, for example: Tight Binding Model, Hubbard Model, etc. The work done in this thesis is based on Hubbard Model.
Hubbard model helps us understand the interactions between particles on a lattice. But this model cannot be solved analytically beyond one dimension. Hence, various numerical methods have been developed for simulating a lattice system. One such method is the Hamiltonian Monte Carlo (HMC) which is suitable even for higher dimensions.
To implement HMC, fermionic action needs to be discretized, one has the freedom to formulate lattice action in many ways as long as the proper continuum limits are recovered. These differences in the discretizations lead to the differences in structure of the fermion matrix.

In this thesis, we will be discussing about the diagonal and the exponential discretizations of the fermion matrix. Each discretization has its own features and advantages and disadvantages. Formalism and theoretical framework relevant to the thesis has been discussed in chapter 2. The subsequent chapters discuss about implementation of the source code for fermion matrix, testing the code and lastly, benchmarking.

The code discussed in this thesis was written for the Nanosystem Simulation Library (NSL). The library is in development phase and is being developed to carry out Lattice simulations on nanoscales for structures such as graphene, carbon nanotubes, etc. To carry out a HMC simulation, one needs an implementation of fermion matrix. Therefore, the diagonal and the exponential discretizations (of fermion matrix) were implemented as described in Chapter 3, where also the (modified) source code is shown. In recent years, GPUs are being used for the simulations since they outperform the CPUs. We also present our attempt at implementing these discretizations on a GPU.

Having a solid testing framework for a library is essential to make sure that the computations are being carried out in an expected way. It makes sure that certain rules are being followed even when other functionalities are added or changed. Tests for the code implemented for the diagonal and the exponential discretizations are discussed in chapter 4.

Keeping the execution time low for a simulation is necessary. To measure the execution time or the bandwidth of the implemented code, benchmarks were run on a CPU as well as a GPU. As discussed in chapter 5, GPU provides a huge speed-up in certain cases as compared to CPU.

Conclusions and open issues regarding the work presented in this thesis are discussed in chapter 6.

CHAPTER 2

Formalism

2.1 Hubbard Model

The Hubbard model describes the interaction of electrons in a lattice. Unlike the tight binding model hamiltonian, which contains a kinetic term only, the Hubbard model Hamiltonian comprises of two terms: a kinetic term which allows hopping of particles between lattice sites, and an on-site interaction term which accounts for the Coulomb interaction between the electrons. The model was proposed in 1963 [14] by JH Hubbard and since then it has been used to simulate various solid-state systems and predicting the behavior of these systems in certain conditions. These conditions can be imposed either by changing the strength of the hopping term or the interaction term. By tweaking these parameters, one can study how these systems give rise to insulating, magnetic and even superconducting effects. It has also been useful in describing transitions from conductor to insulator. The Hubbard model even predicts the existence of Mott insulators.

2.1.1 Hubbard Hamiltonian

The nearest neighbour tight-binding Hamiltonian can be written as:

$$H_0 = -t \sum_{<x,y>} (c^{\dagger}_{x,\uparrow} c_{y,\uparrow} + c^{\dagger}_{y,\downarrow} c_{x,\downarrow}) \tag{2.1}$$

Here, the summation runs over all the nearest neighbours $< x, y >$ and t is the hopping parameter. The above equation accounts for the destruction of a fermion with spin s on site y and its creation on site x (and vice versa). Since the overlap between wave-functions of particles at two different lattice sites dies exponentially, only nearest-neighbours are considered. Here, $c^{\dagger}_{x,s}$ is a creation operator which creates a fermion of spin s at site x whereas $c_{x,s}$ is an annihilation operator which destroys a fermion of spin s at site x. These operators follow the following anti-commutation relation:

$$\left\{ c_{x,s}, c^{\dagger}_{y,s'} \right\} = \delta_{x,y}\delta_{s,s'}, \qquad \left\{ c^{\dagger}_{x,s}, c^{\dagger}_{y,s'} \right\} = 0, \qquad \left\{ c_{x,s}, c_{y,s'} \right\} = 0 \tag{2.2}$$

The above anti-commutation relation ensures Pauli's Exclusion Principle. The tight-binding model can be further extended by adding an interaction term. Hence, extending it further, the Hubbard

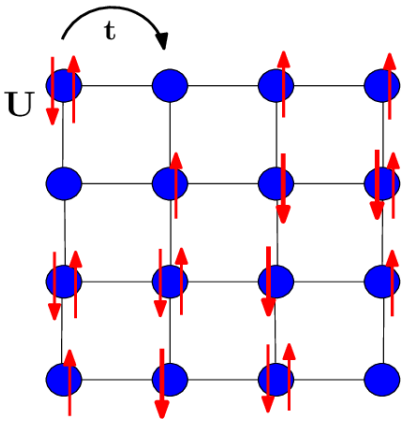


Figure 2.1: Pictorial representation of the kinetic energy t and the on-site repulsion U. [11]

Hamiltonian can be written as:

$$H = H_0 + U \sum_x (n_{x\uparrow} n_{x\downarrow}) - \mu \sum_x (n_{x\uparrow} + n_{x,\downarrow}) \tag{2.3}$$

Here, the first term H_0 is the kinetic term which comes from 2.1. Interactions between fermions are represented by a term which is zero if the site is empty or has a single fermion, and has the value U if the site is doubly occupied (spin-up and spin-down fermions). Hence, the second term accounts for interaction energy; where, $n_{x,s} = c^{\dagger}_{x,s} c_{x,s}$ is the number operator. The term adds up interaction energy for all the sites which are doubly occupied and is zero otherwise. The last term represents chemical potential which affects the filling of the sites.The pictorial representation of the Hubbard Hamiltonian is shown in fig. 2.1. Many systems show interesting behaviour in the half-filled case ($\mu = 0$), for example, anti-ferromagnetic nature, Mott insulation, etc. For this reason (and for simplicity), we are considering $\mu = 0$ in this work. (Note that, when μ=0, hamiltonian is not susceptible to sign problem).

2.1.2 Particle-hole Transformation

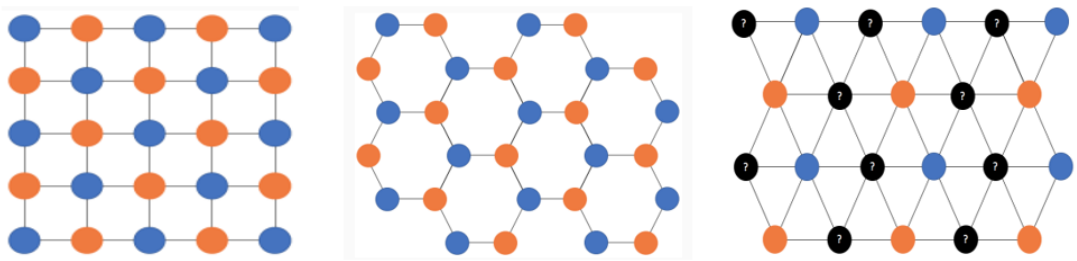


Figure 2.2: Square and Hexagonal Lattices can be divided into two sub-lattices A and B (represented by blue and orange dots here) such that the neighbours of sites of A are members of B only and vice versa. This is not possible for the Triangular Lattice as depicted by the black dots: Left: Square Lattice, Middle: Hexagonal Lattice and Right: Triangular Lattice [21]

Hubbard Hamiltonian follows a 'particle-hole' symmetry. The symmetry gives insight on attractive and repulsive Hubbard Hamiltonian and is useful in numerical simulations. Particle-hole transformation can only be done for bipartite lattices. A lattice is considered to be a bipartite lattice if its sites can be divided into two sub-lattices A and B such that the the neighbours of sites in A are members of B sub-lattice only, for example, Square and Hexagonal Lattices are bipartite whereas Triangular lattice is non-bipartite as can be seen in fig. 2.2. Bipartite lattices are likely to follow anti-ferromagnetic order since the sites in sub-lattice A can occupy the electrons with the opposite spin to that of electrons in sub-lattice B. Bipartite lattices follow the particle-hole symmetry, consider for example, the following transformation:

$$d^{\dagger}_{x,s} = (-1)^{x} c_{x,s} \tag{2.4}$$

The above expression exchanges the role of creation and destruction operator. Here, $(-1)^{x}$ takes the value +1 in one sub-lattice and -1 in another. One can write $n_{x,s} = d^{\dagger}_{x,s} d = 1 - c^{\dagger} c$. Hence, the eigenstates of the number operator, n=0,1, are interchanged. Therefore, this transformation is termed as particle hole transformation. Physically, this transformation signifies that a particle with energy E can be created or a particle with energy $-E$ can be destroyed at the same energy cost. The transformed hubbard hamiltonian in particle-hole basis can be written as:

$$H = -t \sum_{<x,y>} (c^{\dagger}_{x,\uparrow} c_{y,\uparrow} + c^{\dagger}_{y,\downarrow} c_{x,\downarrow}) + U \sum_{x} (n_{x\uparrow} - \frac{1}{2})(n_{x\downarrow} - \frac{1}{2}) - \mu \sum_{x} (n_{x\uparrow} + n_{x,\downarrow}) \tag{2.5}$$

Note that, 2.2 and 2.5 are completely equivalent. Hubbard Hamiltonian in particle-hole basis gives rise to imaginary Hubbard Stratnovich transformation and contribution of auxiliary fields of $e^{i\phi}$ ([9, 17, 24, 26]), which is used in this thesis.

2.2 Path Integral Formalism and Discretization

The partition function Z for our ensemble can be written as:

$$Z = Tr[e^{-\beta H}] \tag{2.6}$$

where, β is the inverse temperature and to simplify further, $e^{-\beta H}$ is split in N_t time slices:

$$e^{-\beta H} \equiv e^{-\delta H} e^{-\delta H} ... e^{-\delta H}, \qquad \delta = \beta / N_t \tag{2.7}$$

Introducing complete set of fermionic states: ψ and χ (for electrons and holes respectively) in the expression for partition function, we obtain:

$$Z = \int \prod_{t=0}^{N_t - 1} \left\{ \left[\prod_{\alpha} d\psi^{*}_{\alpha,t} d\psi_{\alpha,t} d\chi^{*}_{\alpha,t} d\chi_{\alpha,t} \right] e^{-\sum_{\alpha} (\psi^{*}_{\alpha,t+1} \psi_{\alpha,t+1} + \chi^{*}_{\alpha,t+1} \chi_{\alpha,t+1})} \left\langle \psi_{t+1}, \chi_{t+1} | e^{-\delta H} | \psi_t \chi_t \right\rangle \right\} \tag{2.8}$$

After the Hubbard Stratnovich transformation and introduction of auxiliary field, the partition function in spin basis can be written as [29]:

$$\begin{aligned} Z_{\uparrow\downarrow} &= \int \left[\prod_{x,t} d\phi_{x,t}\right] W[\phi] = \int \left[\prod_{x,t} d\phi_{x,t}\right] detM[\phi] detM[-\phi] exp\left(-\frac{1}{2\tilde{U}}\sum_{x,t}\phi_{x,t}^2\right) \\ &= \int \left[\prod_{x,t} d\phi_{x,t}\right] exp\left(-\frac{1}{2\tilde{U}}\sum_{x,t}\phi_{x,t}^2 + logdetM[\phi] + logdetM[-\phi]\right) \end{aligned} \tag{2.9}$$

Here, ϕ is an auxiliary field, $W[\phi]$ is the probability weight for configuration ϕ and $M[\phi]$ is the fermion matrix. In particle hole basis, imaginary auxiliary fields are introduced, therefore, the, partition function can be written as:

$$Z_{ph} = \int \left[\prod_{x,t} d\phi_{x,t}\right] exp\left(-\frac{1}{2\tilde{U}}\sum_{x,t}\phi_{x,t}^2 + logdetM[i\phi] + logdetM[-i\phi]\right) \tag{2.10}$$

Here, elements of the fermion matrix M can be written in different ways for different discretizations of the fermionic action. In the thesis, two such discretizations have been discussed: the diagonal and the exponential. For the diagonal discretization, hopping term is put on the diagonal. For this case, elements of the fermion matrix can be written as [7, 10, 24]:

$$M(\phi,\tilde{\kappa},\tilde{\mu})_{x't';xt} = (1+\tilde{\mu})\delta_{x'x}\delta_{t't} - \tilde{\kappa}_{x'x}\delta_{t't} - B_{t'}e^{i\phi_{xt}}\delta_{x'x}\delta_{t't+1} \tag{2.11}$$

where, B_t is for the anti-periodic boundary condition, κ is the hopping matrix, $\tilde{\kappa} = \delta\kappa$, $\tilde{\mu} = \delta\mu$ and ϕ is auxiliary field.

$$B'_t = \begin{cases} +1, & 0 < t < N_t \\ -1, & t = 0 \end{cases} \tag{2.12}$$

For the exponential discretization, elements of the fermion matrix can be written as [8, 18, 27]:

$$\hat{M}(\phi,\tilde{\kappa},\tilde{\mu})_{x't';xt} = \delta_{x'x}\delta_{t't} - B_{t'}(e^{(\tilde{\kappa}-\tilde{\mu})})_{x'x}e^{i\phi_{x(t'-1)}}\delta_{t'(t+1)} \tag{2.13}$$

where the boundary term is same as above. Note that, we have implemented these expressions for $\mu = 0$. Each discretization comes with its own pros and cons, for example, the exponential discretization in the particle-hole basis suffers from ergodicity problem unlike the diagonal discretization for 2D Hubbard Model Simulation [29]. Ergodicity occurs when the MCMC cannot sample through the relevant region of configuration space and is trapped only in one region (for example, when $detM[i\phi]$ vanishes). These discretizations are eventually used in HMC simulations, hence, it's necessary to have a brief introduction to stochastic methods before moving further.

2.3 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) techniques are used for sampling from high dimensional target distributions, it is usually employed for simulating systems with Gibbs canonical distribution:

$$p(x) \propto e^{-E(x)/T} \tag{2.14}$$

Here, $p(x)$ is the probability of the system to be in a state x where $E(x)$ is the energy of the system. Let's suppose that one wants to study properties of this model, for example, average energy:

$$< E >= \int E(x)p(x)dx \tag{2.15}$$

It gets very hard to calculate the above integral analytically or even numerically for higher dimensions and large lattice systems, this is where MCMC comes to our rescue. By sampling $p(x)$, we can calculate average energy as:

$$< E >= \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} E(x_i) \tag{2.16}$$

But, in the Gibbs distribution, probabilities $p(x)$ are unknown because of the partition function Z which cannot be computed:

$$p(x) = \frac{1}{Z} e^{\left(\frac{-E(x)}{T}\right)} \tag{2.17}$$

Here, Z, the partition function (or normalization constant) is:

$$Z = \int e^{\left(\frac{-E(x)}{T}\right)} dx \tag{2.18}$$

But the ratio of these probabilities does not depend on Z, therefore, this ratio is used to sample the Markov chains by using MH algorithm for MCMC.

2.3.1 Metropolis-Hastings Algorithm

A (Markov-) chain of states: $x_1, x_2, x_3, ..., x_n$ is produced such that:

$$x_{n+1} = x_n + \sigma \tag{2.19}$$

where, σ is some noise (usually sampled from a Gaussian function). The new state $x_{n'}$ is accepted with the probability:

$$p = min\left(1, e^{\left(\frac{E(x_n) - E(x_{n'})}{T}\right)}\right) \tag{2.20}$$

Thus we get our Markov-chain. Note that, the algorithm favours the states with lower energy, since when $U(x_n') < U(x_n)$, $p = 1$. Moreover, Markov chains are reversible:

$$p(x')p(x' \longrightarrow x) = p(x)p(x \longrightarrow x')\forall x, x' \tag{2.21}$$

where, $p(x' \longrightarrow x)$ is the probability of jumping from state x' to x. The above equality ensures that the sampling converges to the canonical distribution.

But, the MH algorithm has some disadvantages:

- A large step size σ leads to many rejected samples, whereas a very small σ might take too long to sample through the distribution.
- MH is very in efficient for higher dimensions.
- It is not possible to travel large distances (significantly larger than σ between isolated local minimums).

To overcome some of the disadvantages of MH, Hamiltonian Monte Carlo (HMC) is employed.

2.3.2 Hamiltonian Monte Carlo

To implement HMC, a trick is used, we introduce a new term $K(p)$ in the hamiltonian [23] (here p is momentum), such that:

$$H(p, x) = E(p, x) = U(x) + K(p) \tag{2.22}$$

Therefore, the parameters x and p have independent canonical distributions:

$$p(x, p) \propto e^{\left(\frac{-E(x,p)}{T}\right)} = e^{\left(\frac{-U(x)}{T}\right)} e^{\left(\frac{-K(p)}{T}\right)} \propto p(x)p(p) \tag{2.23}$$

Hence one can do joint sampling by initializing system parameters (x, v) To get a new pair of these parameters for the markov-chain, we let the system evolve according to Hamiltonian Dynamics.

Hamiltonian Dynamics

Hamiltonian Dynamics describe the evolution of a system in phase space. Hence, given some initial values of x and p, one can predict the values according to the following differential equations also known as Hamilton's equations:

$$\frac{\partial x}{\partial t} = \frac{\partial H}{\partial p_i} = \frac{\partial K(p)}{\partial p_i} \tag{2.24}$$

$$\frac{\partial p_i}{\partial t} = -\frac{\partial H}{\partial x_i} = -\frac{\partial U(x)}{\partial x_i} \tag{2.25}$$

The above equations can be simulated by various methods, one such method is Leapfrog Algorithm.

Leapfrog Algorithm

Leapfrog method is a symplectic integration algorithm, hence it preserves volume in phase space. Therefore, the energy of the ensemble remains conserved (unlike while using Euler's method) as

shown in figure 2.3. Following steps are followed while implementing the algorithm:

- Set $(p, x) = (p_0, x_0)$
- First (half) step:

$$p_i(t + \epsilon/2) = p_i(t) - (\epsilon/2)\frac{\partial U}{\partial x_i(t)} \tag{2.26}$$

- Next full step:

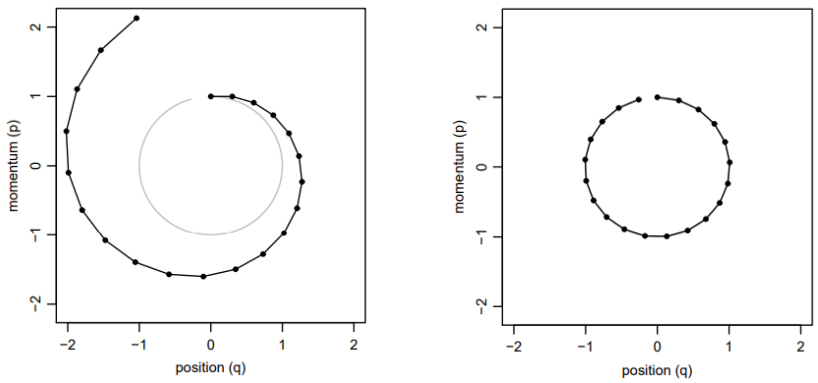
$$x_i(t + \epsilon) = x_i(t) + (\epsilon)\frac{\partial K}{\partial p_i(t + \epsilon/2)} \tag{2.27}$$

- Final (half) step:

$$p_i(t + \epsilon) = p_i(t + \epsilon/2) - (\epsilon/2)\frac{\partial U}{\partial x_i(t + \epsilon)} \tag{2.28}$$

- Set (p_f, x_f) = (p,x)

Hence, given some initial points (p_0, x_0), Hamilton's equations predict the evolution of these points such that $(p_0, x_0) \longrightarrow (p_f, x_f)$ and $\mathcal{H}(p_0, x_0) = \mathcal{H}(p_f, x_f)$. Here, $\epsilon = T/N$ is the step size and T is length of the trajectory. Here, the full step can be performed $N - 1$ number of times for the position as well as the momentum variables.
These new points (p_f, x_f) can either be accepted or rejected as already discussed in the above sections. Hence, one can find observables using HMC, which otherwise would be impossible to find using analytical techniques.



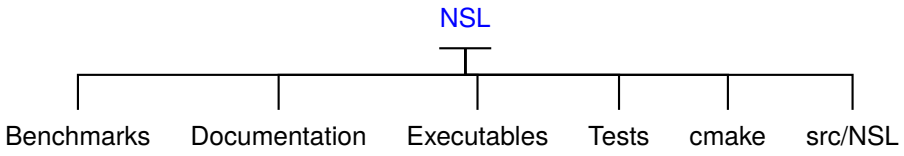
(a) Euler's method, step size 0.3 (b) Leapfrog Method, stepsize 0.3

Figure 2.3: Results using Euler's and Leapfrog methods for approximating Hamiltonian dynamics, when $H(q, p) = q^2/2 + p^2/2$. The initial state was $q = 0$, $p = 1$ [19]

2.4 Nanosystem Simulation Library (NSL) Setup

The Nanosystem Simulation Library (NSL) is being developed to carry out statistical simulations for systems at nanoscales. The programming language (currently) used in the source files is $C++$. The library is conceived as a merge of the two software packages developed by the group: *CNS* and

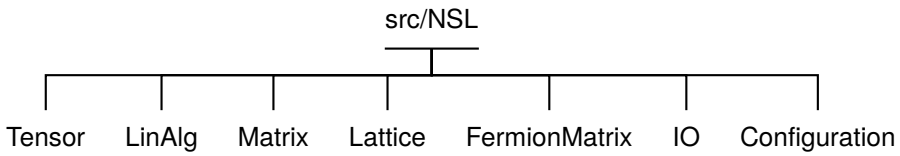
Isle [15]. Both of these libraries are designed for the Hubbard model, whereas NSL is being designed for a more general scope. Moreover, NSL will be able to utilize GPUs for greater efficiency. The relevant steps on how it can be compiled on JURECA DC Module [2] can be found in appendix A.1. NSL is a private *git* repository and can be accessed through the link mentioned in [3]. The hierarchy of the folders in the library (on the *devel* branch) can be represented by the following chart:



- Here, the ***Benchmarks*** folder has the Benchmarks for various functionalities of NSL. Benchmarks relevant to this thesis will be discussed in great detail in chapter 5. Benchmarking is done to get a better understanding of the efficiency of various functions and to make sure that everything is working at an expected bandwidth.
- Currently, ***Documentation*** contains the notes on the features that are planned to be implemented or that have already been implemented.
- ***Executables*** will contain various examples of different features implemented in NSL. These examples will be very helpful for a new NSL user who wishes to get more accustomed with the library or a user who wants to gain insight on how various functionalities of the NSL may be used in their own code.
- As the name suggests, ***Tests*** make sure that everything in the library is working according to certain rules and guidelines set up by the developers. The folder *Tests* contains various tests for the source files. *Catch*2 [1] has been used for the testing framework. The tests relevant to this thesis have been discussed in detail in chapter 4.
- ***cmake*** is being used to control the software compilation process. It contains files to fetch the dependencies on which NSL is built, set environment variables, etc.
- Lastly, ***src/NSL*** contains the source code implementation for NSL which will eventually be used to simulate lattice systems on nanoscales. More about these source files will be discussed in the next section.

2.4.1 Source Code Implementation

Before moving further, it is necessary to get familiar with the NSL semantics, hence, in this section we will discuss the source files which are relevant for the thesis. NSL is in development phase and the current version of the source code files in *src/NSL* can be represented by the following chart:



- ***Tensor*** has the implementation of the *Tensor* class. These *tensors* are the data structures used to represent the fields and configurations as matrices and vectors. *Tensor* class has been defined under the namespace `NSL`. We used *Torch* [4] for the definition of these data structures. The basic underlying data structure of this class is of the type `torch::Tensor`, therefore, the design principle abstracts the *torch* backend. Many in-place operations such as addition, subtraction, element wise multiplication and division, comparison operators, etc. and many functions such as matrix-vector product, slicing, expansion, transposition, shifting, etc. have also been implemented for the class. For example, one can initialize a *Tensor* as follows:

  Listing 2.1: Creates a `Tensor` of dimensions $(x \times y \times z)$ whose elements are of the type `double`

  ```
  NSL::Tensor<double> data(x,y,z);
  ```

  By default, all the elements of the tensor are initialized to 0, one can also define *Tensors* with more than three dimensions. A *Tensor* type is compatible with CPU, as well as GPU (discussed in section 2.5). However, to define a tensor on GPU, the above syntax has to be modified as shown in Listing 2.2:

  Listing 2.2: Creates a GPU `Tensor` of dimensions $(x \times y \times z)$ whose elements are of the type `double`

  ```
  NSL::Tensor<double> data(NSL::GPU(),x,y,z);
  ```

  These data structures have been used throughout the code implemented for the thesis.

- ***LinAlg*** defines various functions such as transposition, matrix-vector multiplication, adjoint, det, logdet, etc. for the *Tensor* types but these functions do not act in-place like the *Tensor* operators and functions. Hence, one can use these functions when one does not want to change the original data of the *tensors*. All the functions in *LinAlg* are defined under the namespace `NSL::LinAlg`. The *Solver* class is also defined under the `LinAlg` namespace. This class provides the interface for algorithms that solve equations of the form:

$$Mx = b \tag{2.29}$$

  for x, here, x and b are vectors and M is a matrix. The interface solves for the matrices defined via `NSL::FermionMatrix` and for general dense matrices.

- ***Matrix*** will have implementations for various types of useful matrices such as an identity matrix, Levi-Civita epsilon, etc. Currently it has the implementation of an identity matrix which has been defined under the namespace `NSL::Matrix`.

- ***Lattice*** contains a base class *SpatialLattice* for spatial lattices (spatial sites connected together). The class is defined under the namespace `NSL::Lattice`. This class offers a variety of default methods, for example, it provides generic methods for getting the adjacency matrix from the hopping matrix, computing and memorizing exponential of hopping matrix, etc. The class has three children classes: *Ring*, *Square* and *Complete*. Here, for a *Ring* object, sites are evenly distributed around a circle, for a *Square* object, sites are evenly spaced on orthogonal coordinates and for a *Complete* object, every vertex is connected to other (a complete graph on n vertices).

- ***FermionMatrix*** has the implementation of *FermionMatrix* class and its children classes: *HubbardDiag* and *HubbardExp*. The latter two classes contain various functionalities for the diagonal and the exponential discretization of the fermion matrix. Each of these discretizations have these functions: `M()`, `Mdagger()`, `MdaggerM()`, `MMdagger()` and `logDetM()`. Here, `M()` returns the matrix-vector multiplication of the fermion matrix M and the fermionic state ψ and similarly, `Mdagger()` returns the matrix-vector multiplication of $M^\dagger$ and ψ and so on. There is also an implementation of the dense form of fermion matrix inside the *FermionMatrix* class. More about the analytic expressions for these functions and their numerical implementation will be discussed in detail in chapter 3. The namespaces that are being used here are: `NSL::FermionMatrix`, `NSL::FermionMatrix::HubbardDiag` and lastly, `NSL::FermionMatrix::HubbardExp`. To construct a `HubbardDiag` object M on a CPU, one can simply write the following lines of code:

Listing 2.3: Construction of a `HubbardDiag` object on CPU

```
NSL::complex beta = {2.,0.};
//Test Configuration
NSL::Tensor<NSL::complex<double>> phi(nt,nx);
phi.rand();
//Lattice Object
NSL::Lattice::Ring<NSL::complex<double>> lat(nx);
//HubbardDiag Object
NSL::FermionMatrix::HubbardDiag M(lat,phi,beta);
```

Here, β is a complex number ($\beta = \delta N_t$ from equation 2.7, the default value of beta is taken as 1.0 if it is not passed as a parameter while constructing the object), `phi` and `psi` are complex vectors of dimensions $(nt \times nx)$ for *Ring* lattice and `rand()` initialises `phi` and `psi` with random complex values, `lat` is an object of `Ring` type which goes into the construction of a `HubbardDiag` object. One can construct a `HubbardExp` object in similar manner.

To construct a *HubbardDiag* object M on GPU, one can simply write the following lines of code (note that, the interface to construct this object on GPU might change in the future):

Listing 2.4: Construction of a `HubbardDiag` object on GPU

```
NSL::complex beta = {2.,0.};
// Test Configuration
NSL::Tensor<complex> phi(NSL::GPU(),Nt,Nx); phi.rand();
//Lattice Object
NSL::Lattice::Ring<complex> lat(Nx);
Lattice.to(NSL::GPU());
//HubbardDiag Object
NSL::FermionMatrix::HubbardExp M(lat,phi,beta);
```

2.5 Acceleration

The Central Processing Unit (CPU), often refereed to as the brain of a computer handles the core processing tasks. CPUs are capable of carrying out multiple operations simultaneously and are

efficient in switching from one operation to another within a fraction of a second. But graphical processing is considered to be challenging for a CPU, for example, for a video game, there might be hundreds (or even thousands) of polygons with different attributes being displayed on a screen at any given time. CPUs are not realy good at handling such operations, hence, to accelerate such operations, GPUs come to the rescue. GPUs were initially designed for rendering graphics, but their applications have evolved over time and are now extensively used in HPC, Artificial Intelligence, Economics and other scientific and commercial fields. Figure 2.4 depicts the architecture of a full NVIDIA GV100 GPU with 84 Streaming Multiprocessor (SM)s.



Figure 2.4: NVIDIA Volta GV100 Full GPU with 84 SM Units [20]

GPUs are very similar to CPUs but have multiple smaller cores, which allow for hardware acceleration and (the high-end GPUs used in HPC) integrate High Bandwidth Memory (HBM) on their package. Hence, they are efficient in carrying out parallel instructions with large amounts of data. Instead of context switching to manage multiple tasks (like CPU), GPU processes parallel data through multiple cores which results in a speed-up. Hence, NSL is being developed in such a way that it can run on a GPU so that we may obtain higher bandwidths. The JURECA DC Module [25] has been used for benchmarking various functionalities of the current version of NSL.

Torch provides supports for CUDA *tensor* types which are similar to CPU tensors, but they utilize GPUs for computation. This feature provided by $Torch$ has been exploited for benchmarking various functionalities of NSL. Comparison of the $FermionMatrix$ benchmarks on CPUs vs GPUs has been done in chapter 5.

CHAPTER 3

Implementation

This chapter describes the implementation of the fermion matrix M for the Hubbard model [28]. As discussed in chapter 2, we are following two discretizations namely: diagonal and the exponential discretizations for the computation of M. These discretizations have been implemented by the following classes:

- `HubbardDiag`
- `HubbardExp`

These classes have been implemented in the following files:

- `src/NSL/FermionMatrix/Impl/hubbardDiag.tpp`
- `src/NSL/FermionMatrix/Impl/hubbardExp.tpp`

In this chapter, we'll discuss the implementation of `M()`, `Mdagger()`, `MMdagger()`, `MdaggerM()` and `logDetM()` for each discretization. These functions accept *psi*, the fermionic states as a parameter. Note that, the source code snippets given in this chapter have been modified for better readability and understanding. The original source code can be read in `NSL/src/NSL/FermionMatrix/`.

3.1 Computing and Implementing M, $M^\dagger$, $MM^\dagger$ and $M^\dagger M$

In this section, numerical and analytical implementations for M, $M^\dagger$, $MM^\dagger$ and $M^\dagger M$ for the diagonal and the exponential discretizations will be worked out. The expressions for these terms were implemented as an application of M ($M^\dagger$, $MM^\dagger$, $M^\dagger M$) on fermionic states ψ (as matrix-vector multiplication). Therefore, `M()`, `Mdagger()`, `MMdagger()` and `MdaggerM()` take a *Tensor* type, `psi` as an argument and return a *Tensor* type. The method `logDetM()` does not require any arguments.

3.1.1 Diagonal Discretization

This discretization is called diagonal discretization because the hopping term is put on the diagonal of the matrix. For this case, as discussed in chapter 2, the fermion matrix can be written as:

$$\begin{aligned} M(\phi,\tilde{\kappa},\tilde{\mu})_{x't';xt} &= \delta_{x'x}\delta_{t't} - \tilde{\kappa}_{x'x}\delta_{t't} - B_{t'}e^{i\phi_{xt}}\delta_{x'x}\delta_{t',t+1} \\ &\equiv K(\tilde{\kappa})_{x'x}\delta_{t't} - B_{t'}F_{t'}(\phi)_{x'x}\delta_{t',t+1} \end{aligned} \tag{3.1}$$

where,

$$K(\tilde{\kappa},\tilde{\mu})_{x'x} = \delta_{x'x} - \tilde{\kappa}_{x'x}, \tag{3.2}$$

$$F_{t'}(\phi)_{x'x} = e^{i\phi_{x,t'-1}}\delta_{x'x} \tag{3.3}$$

The boundary conditions are same as mentioned in 2.12.

Implementing *M*

Since $\tilde{\kappa} = \delta\kappa$ in 3.1, we get the following expression (after replacing the indices $t' \longrightarrow t$, $t \longrightarrow i$ and $x' \longrightarrow x$, $x \longrightarrow y$):

$$M_{tx;iy} = \delta_{xy}\delta_{ti} - \delta\kappa_{xy}\delta_{ti} - B_t e^{i\phi_{iy}}\delta_{xy}\delta_{t,i+1} \tag{3.4}$$

The above expression was implemented as:

Listing 3.1: Implementation of `NSL::FermionMatrix::HubbardDiag::M()`

```
//To compute: M(psi)
//(M \psi)_{x,t} = \psi_{xt} - [(\delta \kappa)]_{xy} \psi_{yt} - B_t \delta_{t,i
    +1} (exp(i \phi_{ix}) \psi_{xi})

    //element-wise multiplication
    NSL::Tensor<Type> out = (this->phiExp_ * psi);
    out.shift(1);
    //apply Boundary condition
    out(0,NSL::Slice()) *= -1;

    out += NSL::LinAlg::mat_vec(
    //The needed matrix multiplication is on the spatial index.
        this->Lat.hopping_matrix(delta_),
    //To get correct broadcasting we transpose the element-wise multiplication
    //so that each column is Nx big.
        NSL::LinAlg::transpose(psi)
    ).transpose();
    //and then transpose back.

    //Now, we have
    //[(\delta \kappa)]_{xy} \psi_{yt} +  \exp(i \phi_{iy}) B_t \delta_{t,i+1} \
        delta_{xy} \psi_{yi}

//returning M(psi)
return psi - out;
```

Here, `shift(1)` shifts the `out` tensor by one time slice due to the presence of $\delta_{t,i+1}$ in 3.4, after which, the boundary conditions are applied using `NSL::Slice()` in the first time-slice. Here, `NSL::LinAlg::mat_vec()` is then used for matrix multiplication of *hopping term* and *psi*. Transpose of *psi* is done to account for the dimension mismatch while matrix multiplication. Note that, `Lat.hopping_matrix()` returns the *hopping matrix*.

Implementing $M^\dagger$

From the equation 3.4, we can derive $M^\dagger$ as follows:

$$\begin{aligned} M^*_{tx;iy} &= \delta_{xy}\delta_{ti} - \delta^* \kappa^*_{xy}\delta_{ti} - B_t e^{-i\phi^*_{iy}}\delta_{xy}\delta_{t,i+1} \\ &= (M^*_{iy;tx})^T = M^\dagger_{iy;tx} \end{aligned} \tag{3.5}$$

Since κ is always hermitian, one can write:

$$\begin{aligned} M^\dagger_{tx;iy} &= \delta_{yx}\delta_{it} - \delta^* \kappa^*_{yx}\delta_{ti} - B_i e^{-i\phi^*_{tx}}\delta_{yx}\delta_{t+1,i} \\ &= \delta_{xy}\delta_{ti} - \delta^* (\kappa^*_{xy})^T\delta_{ti} - B_i e^{-i\phi^*_{xt}}\delta_{yx}\delta_{i,t+1} \\ &= \delta_{xy}\delta_{ti} - \delta^* (\kappa_{xy})^\dagger\delta_{ti} - B_i e^{-i\phi^*_{xt}}\delta_{yx}\delta_{i,t+1} \\ &= \delta_{xy}\delta_{ti} - \delta^* (\kappa_{xy})\delta_{ti} - B_i e^{-i\phi^*_{tx}}\delta_{yx}\delta_{t+1,i} \end{aligned} \tag{3.6}$$

The above expression was implemented as:

Listing 3.2: Implementation of `NSL::FermionMatrix::HubbardDiag::Mdagger()`

```
//To compute: Mdagger(psi)
//(M^{\dagger} \psi)_{tx}  = \psi_tx - [\delta^* \kappa]_{xy}\delta_{t,i} \psi_{
    ix} - exp(-i\phi_{tx}^*) \delta_{t+1,i} B_i  \psi_{ix}

    NSL::Tensor<Type> Bexpconjphipsi(psi,true);

    //applying boundary condition
    Bexpconjphipsi(0,NSL::Slice()) *= -1;

    Bexpconjphipsi = NSL::LinAlg::conj(this->phiExp_) * Bexpconjphipsi.shift(-1);

//returning  Mdagger(psi)
return psi - NSL::LinAlg::mat_vec(this->Lat.hopping_matrix(NSL::LinAlg::conj(
    delta_)), NSL::LinAlg::transpose(psi)).transpose() - Bexpconjphipsi;
```

`Mdagger()` is implemented in a very similar way as `M()`. But here, we also use `NSL::LinAlg::conj()` to account for the complex conjugation of $e^{i\phi}$ and δ in 3.6.

Implementing $MM^\dagger$

To arrive at expression for $MM^\dagger$, substitute the values of M and $M^\dagger$ from 3.4 and 3.6 respectively, as shown below:

$$\begin{aligned} MM^\dagger_{tx;iy} &= M_{tx;uz}M^\dagger_{uz;iy} \\ &= (\delta_{xz}\delta_{tu} - \delta\kappa_{xz}\delta_{tu} - B_t e^{i\phi_{uz}}\delta_{xz}\delta_{t(u+1)}) \\ &\quad (\delta_{zy}\delta_{ui} - \delta^*(\kappa_{zy})\delta_{ui} - B_i e^{-i\phi^*_{uz}}\delta_{yz}\delta_{u+1,i}) \end{aligned} \tag{3.7}$$

Note, that the first term in each parentheses is an identity matrix, hence we can expand the above expression as follows:

$$\begin{aligned} MM^{\dagger}_{tx;iy} &= M_{tx;iy} + M^{\dagger}_{tx;iy} - 1_{tx;iy} + \delta\kappa_{xz}\delta_{tu}\delta^{*}(\kappa_{zy})\delta_{u,i} \\ &+ \delta\kappa_{xz}\delta_{t,u}B_i e^{-i\phi^{*}_{uz}}\delta_{yz}\delta_{u+1,i} + B_t e^{i\phi_{uz}}\delta_{xz}\delta_{t,u+1}\delta^{*}\kappa_{zy}\delta_{u,i} \\ &+ B_t e^{i\phi_{uz}}\delta_{xz}\delta_{t,u+1}B_i e^{-i\phi^{*}_{uz}}\delta_{yz}\delta_{u+1,i} \end{aligned} \tag{3.8}$$

The above expression was implemented as:

Listing 3.3: Implementation of `NSL::FermionMatrix::HubbardDiag::MMdagger()`

```
//To compute: MMdagger(psi)
//(MM^{\dagger}\psi)_{tx} = M\psi_{tx} + M^{\dagger}\psi_{tx} - \psi{tx} + [\
    delta \kappa]_{xz} [\delta^* \kappa]_{zy}\delta_{t,i}\psi_{iy} + [\delta \
    kappa]_{xy} exp(-i\phi_{ty}^* ) \delta_{i,t+1} B_i \psi_{iy} + B_t \delta_{t,
    i+1} exp(+i\phi_{ix} ) [\delta^* \kappa]_{xy}\psi_{iy} + exp(i (\phi-\phi^*)_
    {t-1,x}\psi_{tx}

    NSL::Tensor<Type> out, KexpconjphiBpsi, BexpconjdelKpsi, out_;

    out = this->M(psi) + this->Mdagger(psi) - psi + NSL::LinAlg::mat_vec(this->
        Lat.hopping_matrix((delta_)), NSL::LinAlg::mat_vec(this->Lat.
        hopping_matrix((NSL::LinAlg::conj(delta_))),NSL::LinAlg::transpose(psi)))
        .transpose();

    KexpconjphiBpsi = psi;

    //applying boundary conditions
    KexpconjphiBpsi(0,NSL::Slice()) *= -1;
    out += NSL::LinAlg::mat_vec(this->Lat.hopping_matrix(delta_), (NSL::LinAlg::
        conj(this->phiExp_) * KexpconjphiBpsi.shift(-1)).transpose()).transpose()
        ;

    BexpconjdelKpsi = (this->phiExp_) * NSL::LinAlg::mat_vec(this->Lat.
        hopping_matrix(NSL::LinAlg::conj(delta_)), NSL::LinAlg::transpose(psi)).
        transpose();

    BexpconjdelKpsi.shift(1);

    //aplying boundary conditions
    BexpconjdelKpsi(0,NSL::Slice()) *= -1;

    out += BexpconjdelKpsi + (((this->phiExp_) * NSL::LinAlg::conj(this->phiExp_)
        ).shift(1)) * psi;

//returning MMdagger(psi)
return (out);
```

For this method, we also call `M()` and `Mdagger()` as indicated in 3.8. After that, we implement all the other terms as explained in previous sections.

Implementing $M^\dagger M$

Similarly, substituting the values of M and $M^\dagger$ from 3.4 and 3.6 respectively, we get $M^\dagger M$:

$$\begin{aligned} M^\dagger M_{tx;iy} &= M^\dagger_{tx;uz} M_{uz;iy} \\ &= (\delta_{xz}\delta_{tu} - \delta^* \kappa_{xz}\delta_{tu} - B_u e^{-i\phi^*_{tx}}\delta_{xz}\delta_{(t+1)u}) \\ &(\delta_{zy}\delta_{ui} - \delta(\kappa_{zy})\delta_{ui} - B_u e^{i\phi_{iy}}\delta_{yz}\delta_{u,i+1}) \end{aligned} \tag{3.9}$$

Since the first term in each parentheses is an identity matrix, we can simplify it as follows:

$$\begin{aligned} M^\dagger M_{tx;iy} &= M_{tx;iy} + M^\dagger_{tx;iy} - 1_{tx;iy} + \delta^*\kappa_{xz}\delta_{tu}\delta\kappa_{zy}\delta_{u,i} \\ &+ \delta^*\kappa_{xz}\delta_{tu}B_u e^{i\phi_{iy}}\delta_{zy}\delta_{u,i+1} + B_u e^{-i\phi^*_{tx}}\delta_{zx}\delta_{t+1,u}\delta\kappa_{zy}\delta_{ui} \\ &+ B_u e^{-i\phi^*_{tx}}\delta_{zx}\delta_{t+1,u}B_u e^{i\phi_{iy}}\delta_{zy}\delta_{u,i+1} \end{aligned} \tag{3.10}$$

The above expression was implemented as:

Listing 3.4: Implementation of `NSL::FermionMatrix::HubbardDiag::MdaggerM()`

```
//To compute: MdaggerM(psi)
//(M^{\dagger}M \psi)_{tx} = M\psi_{tx} + M^{\dagger}\psi_{tx} - \psi{tx} + [\
    delta \kappa]_{xz} [\delta^* \kappa]_{zy}\delta_{t,i}\psi_{iy} + B_t \delta_{
    t,i+1} [\delta^* \kappa]_{xy} exp(+i\phi_{iy} ) \psi_{iy} exp(-i\phi_{tx}^* )
     [\delta \kappa]_{xy}  \delta_{i,t+1} B_i \psi_{iy} + exp(i (\phi-\phi^*)_{t,
    x}\psi_{tx}

    NSL::Tensor<Type> out, KexpconjphiBpsi, BexpconjdelKpsi, out_;

    out = this->M(psi) + this->Mdagger(psi) - psi + NSL::LinAlg::mat_vec(this->
        Lat.hopping_matrix((delta_)), NSL::LinAlg::mat_vec(this->Lat.
        hopping_matrix((NSL::LinAlg::conj(delta_))),NSL::LinAlg::transpose(psi)))
        .transpose();

    KexpconjphiBpsi = psi;
    //applying boundary conditions
    KexpconjphiBpsi(0,NSL::Slice()) *= -1;

    out += NSL::LinAlg::conj(this->phiExp_) * NSL::LinAlg::mat_vec(this->Lat.
        hopping_matrix(delta_), KexpconjphiBpsi.shift(-1).transpose()).transpose
        ();

    BexpconjdelKpsi =   NSL::LinAlg::mat_vec(this->Lat.hopping_matrix(NSL::LinAlg
        ::conj(delta_)), ((this->phiExp_) * psi).transpose()).transpose();
    BexpconjdelKpsi.shift(1);

    //applying boundary conditions
    BexpconjdelKpsi(0,NSL::Slice()) *= -1;

    out += BexpconjdelKpsi + ((this->phiExp_) * NSL::LinAlg::conj(this->phiExp_))
         * psi;

//returning MdaggerM(psi)
```

```
return (out);
```

The implementation of this method is very similar to that of `MMdagger()`.

3.1.2 Exponential discretization

This discretization is called exponential discretization due to the exponentiation of the hopping term in the expression of M. In this case, the fermion matrix M can be written as:

$$\hat{M}(\phi,\tilde{\kappa},\tilde{\mu})_{x't';xt} \equiv \hat{K}_{x'x}\delta_{t't} - B_{t'}\hat{F}_{t'}(\phi,\tilde{\kappa},\tilde{\mu})_{x'x}\delta_{t'(t+1)} \tag{3.11}$$

where,

$$\hat{K}_{x'x} \equiv \delta_{x'x} \tag{3.12}$$

$$\hat{F}_{t'}(\phi,\tilde{\kappa},\tilde{\mu})_{x'x} \equiv (e^{(\tilde{\kappa}-\tilde{\mu})})_{x'x}e^{i\phi_{x(t'-1)}} \tag{3.13}$$

Note that, the exponential discretization also uses the same *LinAlg* functions (such as `conj()`, `mat_vec()`, etc. But since we will be using exponential of hopping term, `Lat.exp_hopping_matrix()` will be called instead of `Lat.hopping_matrix()`. The code is implemented in a similar manner as it was done for the diagonal discretization (application of `shift()`, `transpose()`, `mat_vec()`, etc. remains similar), hence, it does not require a separate explanation.

Implementing M

Equation 3.11 can be written as (replacing $t' \longrightarrow t$, $t \longrightarrow i$ and $x' \longrightarrow x$, $x \longrightarrow y$):

$$M_{tx,iy} = \delta_{xy}\delta_{ti} - B_t(e^{\delta\kappa})_{xy}e^{+i\phi_{iy}}\delta_{t,i+1} \tag{3.14}$$

The boundary conditions are same as mentioned in 2.12.

The above expression was implemented as:

Listing 3.5: Implementation of `NSL::FermionMatrix::HubbardExp::M()`

```
//To compute: M(psi)
//(M \psi)_{tx} = psi -  B_t \delta_{t,i+1} [\exp(\delta \kappa)]_{xy} (\exp(i \
    phi_{iy}) \psi_{yi})

    NSL::Tensor<Type> Fpsi = NSL::LinAlg::mat_vec(
    //The needed matrix multiplication is on the spatial index.
        this->Lat.exp_hopping_matrix(delta_),
    //To get correct broadcasting we transpose the element-wise multiplication
    //so that each column is Nx big.
        (this->phiExp_ * psi).transpose()
    ).transpose();
    //and then transpose back.

    //Now Fpsi contains
    //[\exp(\delta \kappa)]_{xy} (\exp(i \phi_{iy}) \psi_{yi})
    //What remains is to shift it
    Fpsi.shift(1,0);
    //and apply B
```

```
    Fpsi(0,NSL::Slice()) *= -1;

//returning M(psi)
return psi - Fpsi;
```

Implementing $M^\dagger$

From 3.14, $M^\dagger$ can be calculated as follows:

$$\begin{aligned} M^*_{tx,iy} &= \delta_{xy}\delta_{ti} - B_t(e^{\delta\kappa})^*_{xy} e^{-i\phi^*_{iy}} \delta_{t,i+1} \\ &= (M^*_{iy,tx})^T = M^\dagger_{iy;tx} \end{aligned} \tag{3.15}$$

After relabelling the indices in the above equation, we get the expression for $M^\dagger$:

$$\begin{aligned} M^\dagger_{tx;iy} &= \delta_{yx}\delta_{it} - B_i(e^{\delta\kappa})^*_{yx} e^{-i\phi^*_{tx}} \delta_{i,t+1} \\ &= \delta_{xy}\delta_{it} - B_i((e^{\delta\kappa})_{xy})^\dagger e^{-i\phi^*_{tx}} \delta_{t+1,i} \\ &= \delta_{xy}\delta_{it} - B_i(e^{\delta^*\kappa})_{xy} e^{-i\phi^*_{tx}} \delta_{t+1,i} \end{aligned} \tag{3.16}$$

We could simplify $((e^{\delta\kappa})_{xy})^\dagger)$ to $((e^{\delta^*\kappa})_{xy})$ in the last step only because κ is hermitian.

The above expression was implemented as:

Listing 3.6: Implementation of `NSL::FermionMatrix::HubbardExp::Mdagger()`

```
//To compute: Mdagger(psi)
//(M^{\dagger} \psi)_{tx}  = \psi_tx - exp(-i\phi_{tx}^*)  \delta_{t+1,i} B_i  [
    exp(\delta^* \kappa)]_{xy}  \psi_{iy}

    NSL::Tensor<Type> BexpKpsi = NSL::LinAlg::mat_vec(
        this->Lat.exp_hopping_matrix(NSL::LinAlg::conj(delta_)),
        NSL::LinAlg::transpose(psi)
        ).transpose();
    BexpKpsi(0, NSL::Slice()) *= -1;

//returning Mdagger(psi)
return psi - ( NSL::LinAlg::conj(this->phiExp_) * (BexpKpsi.shift(-1,0)));
```

Implementing $MM^\dagger$

Substituting M and $M^\dagger$ from 3.14 and 3.16 respectively, $MM^\dagger$ can be calculated as:

$$\begin{aligned} MM^\dagger_{tx;iy} &= M_{tx;uz} M^\dagger_{uz,iy} \\ &= (\delta_{xz}\delta_{tu} - B_t(e^{\delta\kappa})_{xz} e^{+i\phi_{uz}} \delta_{t,u+1}) \\ &\quad (\delta_{zy}\delta_{ui} - B_i(e^{\delta^*\kappa})_{zy} e^{-i\phi^*_{uz}} \delta_{u+1,i}) \end{aligned} \tag{3.17}$$

Since the first term in each parentheses in the above expression is an identity matrix, we can expand it as follows:

$$\begin{aligned} MM^{\dagger}_{tx;iy} &= M_{tx;iy} + M^{\dagger}_{tx;iy} - \delta_{ti}\delta_{xy} \\ &\quad + (e^{\delta\kappa})_{xz} e^{+i\phi_{uz}} B_t \delta_{t,(u+1)} B_i (e^{\delta^*\kappa})_{zy} e^{-i\phi^*_{uz}} \delta_{(u+1),i} \\ &= M_{tx;iy} + M^{\dagger}_{tx;iy} - 1_{tx;iy} + (e^{\delta\kappa})_{xz}\delta_{ti} e^{i(\phi-\phi^*)_{i-1,z}} (e^{\delta^*\kappa})_{zy} \end{aligned} \tag{3.18}$$

The boundary term vanishes in the above expression since B_t^2 is +1 (as B_t is either +1 or −1).

The above expression was implemented as:

Listing 3.7: Implementation of `NSL::FermionMatrix::HubbardExp::MMdagger()`

```
//To compute: MMdagger(psi)
//(MM^{\dagger} \psi)_{tx} = (M\psi)_{tx} + (M^{\dagger}\psi)_{tx} - \psi_{tx} +
    exp(\delta \kappa)_{xz} exp(i(\phi - \phi^{*}))_{t-1,z} exp(\delta^{*} \kappa
    )_{zy} \psi_{iy} \delta_{ti}

//returning MMdagger(psi)
return (this->M(psi) + this->Mdagger(psi) - psi) + NSL::LinAlg::mat_vec(
    this->Lat.exp_hopping_matrix(delta_),
    (   NSL::LinAlg::shift(this->phiExp_ * NSL::LinAlg::conj(this->phiExp_), +1,
        0)
      * NSL::LinAlg::mat_vec(
            this->Lat.exp_hopping_matrix(NSL::LinAlg::conj(delta_)),
            NSL::LinAlg::transpose(psi)
        ).transpose()
    ).transpose()
).transpose();
```

Implementing $M^{\dagger}M$

Substituting M and $M^{\dagger}$ from 3.14 and 3.16 respectively, $M^{\dagger}M$ can be calculated as:

$$\begin{aligned} M^{\dagger}M_{tx;iy} &= (M)^{\dagger}_{tx;uz}(M)_{uz;iy} \\ &= (\delta_{xz}\delta_{tu} - B_u(e^{\delta^*\kappa})_{xz} e^{-i\phi^*_{tx}} \delta_{t+1,u}) \\ &\quad (\delta_{zy}\delta_{ui} - B_u(e^{\delta\kappa})_{zy} e^{+i\phi_{iy}} \delta_{u,i+1}) \end{aligned} \tag{3.19}$$

Note, that the first term in each parentheses is an identity matrix, hence, we can expand it as:

$$\begin{aligned} M^{\dagger}M_{tx;iy} &= M_{tx;iy} + M^{\dagger}_{tx;iy} - \delta_{ti}\delta_{xy} \\ &\quad + B_u^2 \delta_{t+1,u}\delta_{u,i+1} e^{-i\phi^*_{tx}} (e^{\delta^*\kappa})_{xz} (e^{\delta\kappa})_{zy} e^{+i\phi_{iy}} \\ &= M_{tx;iy} + M^{\dagger}_{tx;iy} - 1_{tx;iy} + e^{-i\phi^*_{ix}} (e^{(\delta^*+\delta)\kappa})_{xy} e^{+i\phi_{iy}} \end{aligned} \tag{3.20}$$

The above expression was implemented as:

Listing 3.8: Implementation of `NSL::FermionMatrix::HubbardExp::MdaggerM()`

```
//To compute: MdaggerM(psi)
//(M^{\dagger}M \psi)_{tx} = (M\psi)_{tx} + (M^{\dagger}\psi)_{tx} - \psi_{tx} +
    exp(i \phi^{*}_{tx}) exp((\delta^* + \delta)\kappa)_{xy} exp(i \phi)_{iy}
    psi_{iy} \delta_{ti}

 //returning MdaggerM(psi)
return this->M(psi) + this->Mdagger(psi) - psi + NSL::LinAlg::conj(this->phiExp_)
     * NSL::LinAlg::mat_mul(
    this->Lat.exp_hopping_matrix(NSL::LinAlg::conj(delta_)+delta_),
    (this->phiExp_ * psi ).transpose()
).transpose();
```

3.2 Computing and Implementing logDetM

To compute $logDetM$ for the diagonal and the exponential discretizations, Lower Upper (LU) decomposition of the matrix M was done such that $M = LU$. This decomposition is used in various numerical methods [16], mainly to solve a square set of linear equations. But here, we'll use it to find $detM$. LU decomposition technique is discussed in detail in the following subsection.

3.2.1 LU decomposition

The L and U in LU decomposition stand for Lower and Upper respectively, we write our given matrix A as a product of L and U where L is a lower triangular matrix and U is an upper triangular matrix:

$$A = LU \tag{3.21}$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} l_{11} & & & & \\ l_{2,1} & l_{2,2} & & & \\ \vdots & \ddots & \ddots & & \\ \vdots & \ddots & \ddots & \ddots & \\ l_{n,1} & l_{n,2} & \cdots & \cdots & l_{n,n} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & \cdots & u_{1,n} \\ & u_{2,2} & \ddots & \ddots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & \vdots \\ & & & & u_{n,n} \end{bmatrix} \tag{3.22}$$

If LU factorization exists and A is non-singular, then the determinant can be written as:

$$Det(LU) = Det(L)Det(U) \tag{3.23}$$

The following two subsections discuss how we use LU decomposition to solve for $detM$ for diagonal and exponential discretizations [28].

3.2.2 Diagonal Discretization

From 3.1, writing M for the diagonal discretization in expanded form:

$$M = \begin{bmatrix} K & & & & F_0 \\ -F_1 & K & & & \\ & -F_2 & K & & \\ & & \ddots & \ddots & \\ & & & -F_{N_t-1} & K \end{bmatrix} \tag{3.24}$$

Here, K's and F's are spatial matrices. Using the following ansatz:

$$L = \begin{bmatrix} \mathbb{1} & & & & \\ l_0 & \mathbb{1} & & & \\ & l_1 & \ddots & & \\ & & \ddots & \mathbb{1} & \\ & & & l_{n-3} & \mathbb{1} \\ & & & & l_{n-2} & \mathbb{1} \end{bmatrix}, U = \begin{bmatrix} d_0 & & & & v_0 \\ & d_1 & & & v_1 \\ & & d_2 & & \vdots \\ & & & \ddots & v_{n-3} \\ & & & d_{n-2} & v_{n-2} \\ & & & & d_{n-1} \end{bmatrix} \tag{3.25}$$

where, l's, d's and v's are matrices of dimensions $(N_x \times N_x)$. M can be written as:

$$M = LU \tag{3.26}$$

Substituting 3.24 and 3.25 in 3.26, one arrives at the following recursive equations:

$$d_i = K \quad for \quad 0 \le i \le N_t - 2 \quad d_{n_t-1} = K - l_{N_t-2} v_{N_t-2} \tag{3.27}$$

$$l_i d_i = -F_{i+1} \quad for \quad 0 \le i \le N_t - 2 \tag{3.28}$$

$$v_o = f_0 \quad and \quad l_{i-1} v_{i-1} + v_i = 0 \quad for \quad 0 \le i \le N_t - 2 \tag{3.29}$$

From 3.27, we know $d_i = K$ for $0 \le i \le N_t - 2$. Hence, substituting it in 3.28, we get:

$$l_i = -F_{i+1} K^{-1} \quad 0 \le i \le N_t - 2 \tag{3.30}$$

From the above equation and from 3.29, we get:

$$\begin{aligned} v_i &= -l_{i-1} v_{i-1} = F_i K^{-1} v_{i-1} = F_i K^{-1} F_{i-1} K^{-1} v_{i-2} \\ &= F_i K^{-1} F_{i-1} K^{-1} F_{i-2} K^{-1} v_{i-3} \\ &= F_i K^{-1} F_{i-1} K^{-1} F_{i-2} K^{-1} \cdots K^{-1} F_1 K^{-1} F_0 \\ &\equiv K A_{0i} \end{aligned} \tag{3.31}$$

where,

$$A_{tt'} = K^{-1} F_{t'} K^{-1} F_{t'-1} K^{-1} F_{t'-2} \cdots K^{-1} F_t \tag{3.32}$$

Hence, from 3.27, 3.31 and 3.32 we arrive at an expression of d_{N_t-1}, which will be used to solve for $detM$:

$$d_{N_t-1} = K + F_{N_t-1}K^{-1}KA_{0(n-2)} = K(\mathbb{1} + A) \tag{3.33}$$

where, A can be written as:

$$A = K^{-1}F_{N_t-1}K^{-1}F_{N_t-2}K^{-1}F_{N_t-3}\cdots K^{-1}F_1K^{-1}F_0 \tag{3.34}$$

From 3.23, we can write:

$$detM = detLdetU \tag{3.35}$$

Since the determinant of the upper or the lower triangular matrix is a product of the diagonal entries, we can write the above equation as:

$$detM = (\prod_{i=0}^{N_t-1} 1)(\prod_{i=0}^{N_t-1} d_i) \tag{3.36}$$

From 3.27 and 3.33, the above expression can be simplified to:

$$\begin{aligned} detM &= (detK)^{N_t-1}det(K + F_{N_t-1}K^{-1}F_{N_t-2}\cdots K^{-1}F_1K^{-1}F_0) \\ &= (detK)^{N_t}det(\mathbb{1} + K^{-1}F_{N_t-1}K^{-1}F_{N_t-2}\cdots K^{-1}F_1K^{-1}F_0) \end{aligned} \tag{3.37}$$

The above expression can be simplified further:

$$\begin{aligned} detM &= (detK)^{N_t-1})det(F_{N_{t-1}})det(F_{N_t-1}^{-1}K + K^{-1}F_{N_t-2}\cdots K^{-1}F_1K^{-1}F_0) \\ &= (detK)^{N_t-2})det(F_{N_{t-1}}F_{N_{t-2}})det(F_{N_t-2}^{-1}KF_{N_t-1}^{-1}K + K^{-1}F_{N_t-3}\cdots K^{-1}F_1K^{-1}F_0) \\ &= (\prod_{t=0}^{N_t-1} detF_t)det(\mathbb{1} + F_0^{-1}KF_1^{-1}K\cdots F_{N_t-1}^{-1}K) \end{aligned} \tag{3.38}$$

Substituting F_t from 3.3, one can simplify further:

$$detM = e^{i\Phi}det(\mathbb{1} + A^{-1}) \tag{3.39}$$

where, $\Phi \equiv \sum_{x,t}\phi_{x,t}$ and

$$A^{-1} = F_0^{-1}KF_1^{-1}K\cdots F_{N_t-1}^{-1}K \tag{3.40}$$

One can find $logDet(M)$ from the above equation as follows:

$$logDet(M) = i\Phi + logDet(\mathbb{1} + A^{-1}) \tag{3.41}$$

The above expression is implemented as:

Listing 3.9: Implementation of `NSL::FermionMatrix::HubbardDiag::logDetM()`

```
//To compute: logDetM()

    const int Nt = this->phi_.shape(0);
```

```
    const int Nx = this->phi_.shape(1);

    NSL::Tensor<Type> prod(Nt,Nx,Nx);
    NSL::Tensor<Type> sausage = NSL::Matrix::Identity<Type>(Nx), fk(Nx,Nx);
    Type sum, I = {0,1};

    //F^{-1} in expanded form
    prod = (NSL::LinAlg::shift(NSL::LinAlg::exp(this->phi_*(-1*I)),-1).expand(Nx)
        .transpose(1,2));
    for(int t = 0;  t < Nt ; t++){

        //mat-vec of F^{-1} and K
        fk = prod(t,NSL::Slice(),NSL::Slice())* (NSL::Matrix::Identity<Type>(Nx)
            - this->Lat.hopping_matrix(delta_));

        sausage.mat_mul(fk);
    }
    //sum over all the elements of phi
    sum = (this->phi_).sum();

//returning logDetM
return ((sum*I) + NSL::LinAlg::logdet(NSL::Matrix::Identity<Type>(Nx) + sausage))
    ;
```

Here, `Identity(Nx)` returns an identity matrix of dimensions $(N_x \times N_x)$. We have used `mat_mul()` instead of `mat_vec()` because we require an in-place operation here. Refer to the comments in the code for more explanation.

3.2.3 Exponential Discretization

For exponential discretization, from 3.11, one can write M as:

$$M = \begin{bmatrix} \mathbb{1} & & & & F_0 \\ -F_1 & \mathbb{1} & & & \\ & -F_2 & \mathbb{1} & & \\ & & \ddots & \ddots & \\ & & & -F_{N_t-1} & \mathbb{1} \end{bmatrix} \tag{3.42}$$

One can find $logDet(M)$ by replacing following terms in 3.38:

- K = $\mathbb{1}$
- $F_{t'} = \hat{F}_{t'}$

$$logDet(M) = logDet(\mathbb{1} + \hat{A}) \tag{3.43}$$

where,

$$\hat{A} \equiv \hat{F}_{N_t-1}\hat{F}_{N_t-2}...\hat{F}_1\hat{F}_0 \tag{3.44}$$

`logDet(M)` for the exponential discretization is implemented as:

Listing 3.10: Implementation of `NSL::FermionMatrix::HubbardExp::logDetM()`

```
//To compute: logDetM()

    const int Nt = this->phi_.shape(0);
    const int Nx = this->phi_.shape(1);

    NSL::Tensor<Type> prod(Nt,Nx,Nx);
    NSL::Tensor<Type> sausage = NSL::Matrix::Identity<Type>(Nx);

    prod = this->Lat.exp_hopping_matrix(this->delta_)* NSL::LinAlg::shift(this->
        phiExp_,-1).expand(Nx).transpose(1,2);

    //Computing F_{Nt-1}.F_{Nt-2}.....F_0
    for(int t = Nt-1;  t >= 0; t--){
        sausage.mat_mul(prod(t,NSL::Slice(),NSL::Slice()));
    }

//returning logDetM
return NSL::LinAlg::logdet(NSL::Matrix::Identity<Type>(Nx) + sausage);
```

The code is implemented in the similar manner as `logDetM()` for the diagonal discretization. Here, K reduces to $\mathbb{1}$ which simplifies the calculation and makes the implementation a little faster than the diagonal discretization.

CHAPTER 4

Tests

Tests are an essential part of a library like NSL as it makes sure that all the functionalities are working as per the rules and guidelines in the testing framework. It also makes sure that those rules are being followed even when other functionalities are added or changed. This chapter describes how various tests for the Diagonal and the Exponential discretizations were implemented. We used `catch2/catch.hpp` to implement all the tests for NSL. The custom test case macros and the other common functions being used to carry out the tests have been declared in `Tests/test.hpp`. The procedure for running the Tests can be found in A.2

The tests for the Diagonal and the Exponential discretizations have been implemented in the following files:

- `Tests/FermionMatrix/test_fermionMatrixHubbardDiag.cpp`
- `Tests/FermionMatrix/test_fermionMatrixHubbardExp.cpp`

For testing these discretizations, we rely on the accuracy of `M()` and `M_dense()` to test if `Mdagger()` is implemented correctly. `M()` and `Mdagger()` are further used to test the accuracy of `MMdagger()` and `MdaggerM()`. The tests for `logDetM()` were implemented by exploiting certain symmetries as discussed in [29]. The implementation of these tests is discussed in more detail in the following sections.

4.1 Testing `M()`, `M_dense()`, `Mdagger()`, `MMdagger()` and `MdaggerM()`

Implementation of tests for `M()`, `M_dense()`, `Mdagger()`, `MMdagger()` and `MdaggerM()` for both of the discretizations have been done in an identical manner, hence, the code snippets for the Exponential discretization only have been presented for these functions. Note that, the code has been modified for better readability and understanding. The original code can be read in `NSL/Tests/FermionMatrix/`.

4.1.1 `M()`

All the other tests are built on the accuracy of `M()`. The method `M()` is tested by hard-coding the entire calculation as shown in the code listing given below:

Listing 4.1: Implementation of `test_fermionMatrixHubbardExp_M()`

```
// ====================================================================
// Implementation Details: fermionMatrixHubbardExp_M
// ====================================================================

    NSL::size_t nx = Lattice.sites();

    NSL::Tensor<Type> phi(nt, nx);
    NSL::Tensor<Type> psi(nt, nx);
    phi.rand();
    psi.rand();

    Type delta = beta/nt;
    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);
    ComplexType I ={0,1};

    //hardcoding the calculation done in the method M of fermionMatrixHubbardExp
        class
    NSL::Tensor<Type> out = NSL::LinAlg::mat_vec(
        Lattice.exp_hopping_matrix(delta),
        ((phi*I).exp() * psi).transpose()
    );

    // anti-periodic boundary condition
    out.transpose();
    out.shift(1,0);
    out.slice(0,0,1)*=-1;

    //Hard-coded result
    NSL::Tensor<Type> result_exa = psi - out;
    //result returned by M.M(psi)
    NSL::Tensor<Type> result_alg = M.M(psi);

    //Test Condition
    REQUIRE( almost_equal(result_exa, result_alg, std::numeric_limits<Type>::
        digits10).all() );
```

4.1.2 `M_dense()`

`M_dense` is an implementation of dense form of M which returns a tensor with dimensions (N_t, N_x, N_t, N_x) (where N_t are number of time slices and N_x is used to refer to spatial coordinates). Here, the first two indices are rows and the latter two are columns. `M_dense()` is just an implementation of M acting on an identity matrix. The method `M_dense()` is currently being used for testing purposes only. The test for `M_dense()` relies on accuracy of `M()`. Note, that

$$(M\Psi)_{tx} = M_{tx;iy}\Psi_{iy} \tag{4.1}$$

In terms of the NSL methods, the above equation can be re-written as:

$$\texttt{M(psi)} = \texttt{mat_vec(M_dense(),psi)} \tag{4.2}$$

We test for the equality of LHS and RHS in 4.2 to test for the accuracy of M_dense() as shown below:

Listing 4.2: Implementation of test_fermionMatrixHubbardExp_M_dense()

```
// ========================================================================
// Implementation Details: fermionMatrixHubbardExp_M_dense
// ========================================================================

    NSL::Tensor<Type> phi(nt, nx);
    NSL::Tensor<Type> psi(nt, nx);
    NSL::Tensor<Type> dense(nt, nx);
    phi.rand();
    psi.rand();

    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);
    NSL::Tensor<Type> sparse = M.M(psi);

    NSL::Tensor<Type> M_dense = M.M_dense(nt);
    // Just do the most obvious mat-vec.
    for(int t=0; t < nt; t++){
        for(int x=0; x < nx; x++){
            for(int i=0; i< nt; i++){
                for(int y=0; y<nx; y++){
                    dense(t,x) += M_dense(t,x,i,y) * psi(i,y);
                }
            }
        }
    }

    //Test Condition
    REQUIRE( almost_equal(sparse, dense).all() );
```

4.1.3 Mdagger()

The test for Mdagger() relies on M_dense(). Note, that:

$$(M^{\dagger}\Psi)_{tx} = M^{\dagger}_{tx;iy}\Psi_{iy} \tag{4.3}$$

In terms of the NSL methods, the above equation can be re-written as:

$$\texttt{Mdagger(psi)} = \texttt{mat_vec(conj(transpose(M_dense())),psi)} \tag{4.4}$$

We test for the equality of LHS and RHS in 4.4 to test for the accuracy of Mdagger() as shown below:

Listing 4.3: Implementation of test_fermionMatrixHubbardExp_Mdagger()

```
// ========================================================================
// Implementation Details: fermionMatrixHubbardExp_Mdagger
// ========================================================================

    NSL::Tensor<ComplexType> phi(nt, nx);
    NSL::Tensor<ComplexType> psi(nt, nx);
```

```
    phi.rand();
    psi.rand();

    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);

    // Construct a dense representation of M^{\dagger} from a dense
        representation of M
    NSL::Tensor<ComplexType> Mdense_dagger = M.M_dense(nt).transpose(0,2).
        transpose(1,3).conj();


    // Compare two ways of computing M^{\dagger}\psi
    NSL::Tensor<ComplexType>Mdense_dagger_psi(nt, nx), M_dagger_psi(nt, nx);
    // by doing the obvious mat-vec,
    for(int t=0; t < nt; t++){
        for(int x=0; x < nx; x++){
            for(int i=0; i< nt; i++){
                for(int y=0; y<nx; y++){
                    Mdense_dagger_psi(t,x) += Mdense_dagger(t,x,i,y) * psi(i,y);
                }
            }
        }
    }
    // and by applying Mdagger
    M_dagger_psi = M.Mdagger(psi);

    //Test Condition
    REQUIRE( almost_equal(Mdense_dagger_psi, M_dagger_psi).all() );
```

4.1.4 `MMdagger()`

The methods `M()` and `Mdagger()` are used to test for the accuracy of `MMdagger()`. We used the following equation for the test:

$$\texttt{MMdagger(psi)} = \texttt{M(Mdagger(psi))} \tag{4.5}$$

In the above equation, on RHS, firstly `Mdagger()` acts on `psi` and then finally `M()` acts on `Mdagger(psi)`, giving us the LHS. We test for the equality of LHS and RHS to test for the accuracy of `MMdagger()`.

Listing 4.4: Implementation of `test_fermionMatrixHubbardExp_MMdagger()`

```
// ========================================================================
// Implementation Details: test_MMdagger
// ========================================================================

    NSL::size_t nx = Lattice.sites();

    NSL::Tensor<Type> phi(nt, nx);
    NSL::Tensor<Type> psi(nt, nx);
    phi.rand();
    psi.rand();
```

```
    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);
    ComplexType I={0,1};

    auto direct = M.MMdagger(psi);
    auto indirect = M.M(M.Mdagger(psi));
    auto diff = direct - indirect;

    //Test Condition
    REQUIRE( almost_equal(direct,indirect, std::numeric_limits<Type>::digits10-1)
        .all() );
```

4.1.5 MdaggerM()

As in the case of `MMdagger()`, the methods `M()` and `Mdagger()` are used to test for the accuracy of `MdaggerM()`. We know:

$$\texttt{MdaggerM(psi)} = \texttt{Mdagger(M(psi))} \tag{4.6}$$

In the above equation, on RHS, firstly `M()` acts on `psi` and then finally `Mdagger()` acts on `M(psi)`, giving us the LHS. We test for the equality of LHS and RHS to test for the accuracy of `MdaggerM()`.

Listing 4.5: Implementation of `test_fermionMatrixHubbardExp_MdaggeM()`

```
// ========================================================================
// Implementation Details: test_MdaggerM
// ========================================================================

    NSL::Tensor<Type> phi(nt, nx);
    NSL::Tensor<Type> psi(nt, nx);
    phi.rand();
    psi.rand();

    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);
    ComplexType I={0,1};

    auto direct = M.MdaggerM(psi);
    auto indirect = M.Mdagger(M.M(psi));

    INFO("nt:␣"+NSL::to_string(nt)+"␣nx:␣"+NSL::to_string(nx));

    //Test Condition
    REQUIRE( almost_equal(direct,indirect, std::numeric_limits<Type>::digits10-1)
        .all() );
```

4.2 Testing logDetM()

As mentioned, tests for `logDetM()` were implemented by exploiting certain symmetries as discussed in [29]. More details are provided in the following sub-sections:

4.2.1 Diagonal Discretization

Rewriting the expressions for $logDetM$ and A^{-1} for the diagonal discretization from 3.41 and 3.40:

$$logDet(M) = i\Phi + logDet(\mathbb{1} + A^{-1}) \tag{4.7}$$

$$A^{-1} = F_0^{-1} K F_1^{-1} K \cdots F_{N_t-1}^{-1} K \tag{4.8}$$

From 3.3,

$$F_t(\phi)_{x'x} = e^{i\phi_{x,(t-1)}} \delta_{x'x} \tag{4.9}$$

The following three tests were implemented for $logDetM$ for the diagonal discretization:

- **Phi Plus 2*pi**

  Note that,

$$\begin{aligned} F_t(\phi + 2n\pi)_{x'x} &= e^{i\phi_{x,(t-1)}} e^{i2n\pi} \delta_{x'x} \\ &= F_t(\phi)_{x'x} \end{aligned} \tag{4.10}$$

  where, n is an integer.

  From the 4.7, 4.8 and 4.10, it can be seen that if $2n\pi$ is added to one of the time slices of ϕ, F_t remains the same and hence, the real part of $logDetM$ does not change. Hence to test the accuracy of `logDetM()`, equality of real parts of the two $logDetM$s is checked (one with ϕ and the other with an addition of $2n\pi$ in one of the time slices of ϕ). The test is shown below:

Listing 4.6: Implementation of `test_logDetM_phi_plus_two_pi()`

```
// ========================================================================
// Implementation Details: test_logDetM_phi_plus_two_pi
// ========================================================================

    // We should find that by shifting any element of phi by 2ĭĂ the real
        part of the determinant doesn't change.
    NSL::size_t nx = Lattice.sites();
    Type delta = beta/nt;

    NSL::Tensor<Type> phi(nt, nx), phiShift(nt, nx), random(nt, nx);
    phi.rand();

    random.rand();
    NSL::Tensor<Type> orbits = static_cast<NSL::Tensor<int>>(10*random);

    RealType two_pi = 2*std::numbers::pi_v<RealType>;
    //shifted phi
    phiShift = phi + two_pi * orbits;

    NSL::FermionMatrix::HubbardDiag M     (Lattice,phi     ,beta);
    NSL::FermionMatrix::HubbardDiag Mshift(Lattice,phiShift,beta);

    Type result = M.logDetM();
    Type result_shift = Mshift.logDetM();
    RealType diff_imag_mod_two_pi = std::remainder(
```

```
            static_cast<RealType>(NSL::imag(result - result_shift)),
            static_cast<RealType>(two_pi)
            );

    //comparing only the real parts
    //Test Condition
    REQUIRE(almost_equal(result_shift.real(),result.real(),std::
        numeric_limits<Type>::digits10-1));
```

- **Time Shift Invariance**

  Using Sylvester's determinant identity,

$$det(\mathbb{1} + AB) = det(\mathbb{1} + BA) \tag{4.11}$$

  the matrices can be permuted cyclically around the determinant. So, the shift in phi in timeslices does not change $logDet(\mathbb{1} + A^{-1})$ and $i\Phi$,therefore, $logDet(M)$ remains unaffected. Hence, to test the accuracy of `logDetM()`, equality of the two $logDetM$s is checked (one with ϕ_t and the other with a shifted configuration $\phi_{t+shift}$).

Listing 4.7: Implementation of `test_logDetM_time_shift_invariance()`

```
// ========================================================================
// Implementation Details: test_logDetM_time_shift_invariance
// ========================================================================

    // We should find that shifting phi in time doesn't change the
        determinant.
    int slices_to_shift_by=4;

    typedef NSL::complex<typename NSL::RT_extractor<Type>::value_type>
        ComplexType;
    size_t nx = Lattice.sites();
    NSL::Tensor<Type> phi(nt, nx), phiShift(nt, nx);
    phi.rand();
    //shifted phi
    phiShift=NSL::LinAlg::shift(phi,slices_to_shift_by);

    Type delta = beta/nt;

    NSL::FermionMatrix::HubbardDiag<Type,LatticeType> M     (Lattice,phi
            ,beta);
    NSL::FermionMatrix::HubbardDiag<Type,LatticeType> Mshift(Lattice,
        phiShift,beta);

    Type result = M.logDetM();
    Type result_shift = Mshift.logDetM();

    //Test Condition
    REQUIRE(almost_equal(result_shift,result, std::numeric_limits<Type>::
        digits10-1) );
```

- **Non Interacting Case**

In the non-interacting case, we can safely consider $\phi = 0$. Hence, $F_t(\phi)$ in 4.9 reduces to an identity matrix. Therefore,

$$A^{-1} = (K)^{N_t} \tag{4.12}$$

$$logDet(M) = logDet(1 + (K)^{N_t}) \tag{4.13}$$

To test the accuracy of `logDetM()`, we test for the equality of LHS and RHS in 4.13.

Listing 4.8: Implementation of `test_logDetM_noninteracting()`

```
// ========================================================================
// Implementation Details: test_logDetM_noninteracting
// ========================================================================

    NSL::size_t nx = Lattice.sites();
    //phi=0
    NSL::Tensor<Type> phi(nt, nx), sausage = NSL::Matrix::Identity<Type>(nx
        );
    Type delta = beta/nt;
    NSL::FermionMatrix::HubbardDiag M(Lattice,phi,beta);

    //When phi=0, logDetM = logdet(1 + K^{t})
    for(int t=0; t<nt; t++){
        sausage.mat_mul(NSL::Matrix::Identity<Type>(nx) - Lattice.
            hopping_matrix(delta));
    }
    Type result_exa = NSL::LinAlg::logdet(
            NSL::Matrix::Identity<Type>(nx) + sausage
    );

    Type result_alg = M.logDetM();

    //Test Condition
    REQUIRE(almost_equal(result_alg,result_exa,std::numeric_limits<Type>::
        digits10-1));
```

4.2.2 Exponential Discretization

Rewriting the expressions for $logDetM$ and $\hat{A}$ for the exponential discretization from 3.43 and 3.44 respectively.

$$logDet(M) = logDet(1 + \hat{A}) \tag{4.14}$$

$$\hat{A} \equiv \hat{F}_{N_t-1}\hat{F}_{N_t-2}...\hat{F}_1\hat{F}_0 \tag{4.15}$$

where from 3.13,

$$\hat{F}_t(\phi, \tilde{\kappa}, \tilde{\mu})_{x'x} \equiv (e^{(\tilde{\kappa}-\tilde{\mu})})_{x'x} e^{i\phi_{x(t-1)}} \tag{4.16}$$

The following four tests were implemented for $logDetM$ for Exponential discretization:

- **Phi Plus 2*pi**

Similar to the 'Phi Plus 2*pi' test for the diagonal discretization, we can write

$$\begin{aligned}\hat{F}_t(\phi + 2n\pi)_{x'x} &= (e^{(\tilde{\kappa})})_{x'x} e^{i\phi_{x(t-1)}} e^{i2n\pi} \\ &= \hat{F}_t(\phi)_{x'x}\end{aligned} \tag{4.17}$$

where, n is an integer. From 4.14 and 4.17, it can be seen that if $2n\pi$ is added to one of the time slices of ϕ, $\hat{F}_t$ remains the same and hence, $logDetM$ does not change. Therefore, to test the accuracy of `logDetM()`, equality of the two $logDetM$s is checked (one with ϕ and the other with an addition of $2n\pi$ in one of the time slices of ϕ). It is implemented in a similar manner as it was done for the diagonal discretization in 4.2.1 except that here, both the real as well as the imaginary parts remain unchanged.

- **Time Shift Invariance**

  This test is implemented in the same way as already discussed for the diagonal discretization in section 4.2.1.

- **Non Interacting Case**

  In the non-interacting case, $\phi = 0$. Hence, $\hat{F}_t(\phi)$ in 4.16 reduces to the exponential of the hopping matrix. Therefore, from 4.14, we can write:

$$logDet(M) = logDet(1 + (e^{\tilde{\kappa}})^{N_t}) \tag{4.18}$$

  To test the accuracy of `logDetM()`, we test for the equality of LHS and RHS in 4.18.

Listing 4.9: Implementation of `test_logDetM_noninteracting()`

```
// ========================================================================
// Implementation Details: test_logDetM_noninteracting
// ========================================================================

    NSL::size_t nx = Lattice.sites();

    NSL::Tensor<Type> phi(nt, nx);
    Type delta = beta/nt;
    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);

    //When phi=0, logDetM = logdet(1 + exp_hopping_matrix(beta))
    Type result_exa = NSL::LinAlg::logdet(
            NSL::Matrix::Identity<Type>(nx) + Lattice.exp_hopping_matrix(
                beta)
    );

    Type result_alg = M.logDetM();
    //Test Condition
    REQUIRE(almost_equal(result_alg,result_exa,std::numeric_limits<Type>::
        digits10-1));
```

- **Uniform Time Slices**

When phi on a given time slice is the same on every spatial site, $e^{(i\phi_t)}$ is proportional to an identity matrix and can be treated like a scalar. Therefore, when every time slice is like that, we add all of these scalars and simplify 4.14. From 4.15 and 4.16, we can deduce:

$$\begin{aligned} logDet(M) &= logDet(1 + \hat{F}_{N_t-1}\hat{F}_{N_t-2}...\hat{F}_0) \\ &= logDet(1 + e^{\tilde{\kappa}}e^{i\phi_{N_t-2}}e^{\tilde{\kappa}}e^{i\phi_{N_t-3}}...e^{\tilde{\kappa}}e^{i\phi_{-2}}) \\ &= logDet(1 + e^{i(\sum_t \Phi_t)}(e^{\tilde{\kappa}})^{N_t}) \end{aligned} \tag{4.19}$$

To test for the accuracy of `logDetM()`, we check for the equality of LHS and RHS in the above equation.

Listing 4.10: Implementation of `test_logDetM_uniform_timeslices()`

```
// ========================================================================
// Implementation Details: test_logDetM_uniform_timeslices
// ========================================================================

    NSL::size_t nx = Lattice.sites();

    NSL::Tensor<Type> phi(nt, nx);
    Type delta = beta/nt;
    ComplexType I ={0,1};

    // When phi on a given timeslice is the same on every spatial site
    NSL::Tensor<Type> tmp (nt); tmp.rand();
    for(int i=0; i<nt; i++){
        phi(i, NSL::Slice() ) = tmp(i) ;
    }
    // exp(i phi(t)) matrix is proportional to the identity matrix and can
        be
    // treated like a scalar.
    // When EVERY timeslice is like that we gather all the scalars together
    NSL::Tensor<ComplexType> sum(1);
    sum(0) = I*phi( NSL::Slice(), 0).sum();
    ComplexType expsum = NSL::LinAlg::exp(sum)(0);

    // to get
    //      logdet M = logdet( 1 + exp(sum(phi(t))) exp(kappa_tilde * Nt))
    //               = logdet( 1 + exp(sum(phi(t))) exp(kappa beta)
    NSL::FermionMatrix::HubbardExp M(Lattice,phi,beta);
    Type result_alg = M.logDetM();
    Type result_exa = NSL::LinAlg::logdet(
            NSL::Matrix::Identity<Type>(nx) +
            expsum * Lattice.exp_hopping_matrix(beta)
    );

    //Test Condition
    REQUIRE(almost_equal(result_alg,result_exa, std::numeric_limits<Type>::
        digits10 - 2));
```

4.3 Accuracy

Since floating point values cannot be compared for equality, it can be hard to decide "how equal" should those values be. More about the comparison of float data types is discussed in [13] and [6]. We used a combination of relative tolerance and absolute tolerance to test for equality of two float values. If `x` and `y` are two float types such that `max` stores the larger of `1,fabs(x),fabs(y)`, then, x and y can be compared as follows:

Listing 4.11: Implementation of `almost_equal()`

```
return std::fabs(x-y) <= std::pow(10,1-std::numeric_limits<Type>::digits10) * max
        || std::fabs(x-y) < std::numeric_limits<Type>::min();
```

Here, the condition before `||` tests for the relative tolerance whereas the condition after that tests for subnormal cases. A 'true' or '1' value returned by this code means that x is "almost-equal" to y.

We also used the function `std::numeric_limits<T>::digits10` of $c++$ standard library in the above code snippet. As explained in [6], the value of `std::numeric_limits<T>::digits10` is the number of base-10 digits that can be represented by the `type T` without change, that is, any number with these many significant decimal digits can be converted to a value of `type T` and back to decimal form, without change due to rounding or overflow. For example, The standard 32-bit floating-point type has a 24 bit fractional part, therefore, it can represent 7 digit decimals since (24 * std::log10(2) is 7.22), but since rounding errors can be non-uniform, the digits are preserved only up to 6 decimal places. Hence, we calculate it as:

$$(24-1)log_{10}(2) = 6.92 \tag{4.20}$$

Here, `std::numeric_limits<Type>::min()` returns the minimum finite value re-presentable by a numeric type T.

CHAPTER 5

Benchmarks

Benchmarking is done to measure bandwidth and for baselining the performance of code. This helps in identifying bottlenecks while comparing algorithms and approaches to solve a certain problem. Hence, it leads us to a more optimal approach. This chapter discusses about the tools that we used to benchmark the methods (`M()`, `Mdagger()`, `MdaggerM()`, `MMdagger()` and `logDetM()`) in the two discretizations: diagonal and the exponential. Later in the chapter, the benchmark plots and results will also be discussed.

5.1 Method

The benchmarks were executed on JURECA DC Cluster [2] using the `chrono` library. More about it is discussed in the following sub-sections.

5.1.1 `std::chrono`

We use `std::chrono` [12] for benchmarking. The `chrono` library consists of three main types:

- Clocks:

  Clock consists of a starting point, or an epoch and tick rate. It also has the member function `now` which tells how much time has passed since the epoch. It provides three clocks: `system_clock` (it represents the system wide real-time wall clock), `steady_clock` (it represents a monotonic clock) and `high_resolution_clock` (it represents the clock with smallest tick period which is usually an alias of `system_clock` or `steady_clock`.

- Time points:

  Time point tells the duration of time that has passed since an epoch. For example, it can be declared as:

Listing 5.1: `Chrono` Time point declaration

```
const std::chrono::time_point<std::chrono::steady_clock> start = std::
    chrono::steady_clock::now();
```

Various operations like addition and subtraction are also a part of `time_point`, hence `time_points` can be added or subtracted like `floats`.

- Durations:

  `duration` represents span of time or time interval. It can use various time units such as nanoseconds, microseconds, hours, etc.

5.1.2 JURECA-DC [2]

JURECA DC is a supercomputer hosted at the Jülich Supercomputing Centre. It reaches the peak performance of 3.54 (CPU) + 14.98 (GPU) Petaflops per second.
To execute and build software on JURECA, different environment variables can be set using `module` command depending on which compilers and software one wants to load. In our case, we loaded `PyTorch` and `CMake`. Once our desired compilers and packages have been loaded, softwares can easily be built. In similar manner, NSL was compiled on JURECA-DC. Node allocation can be requested using *SLURM* job scripts [5] or the `salloc` command as explained in appendix A.3. In a similar manner, benchmarks for NSL were executed for both CPU as well as GPU.

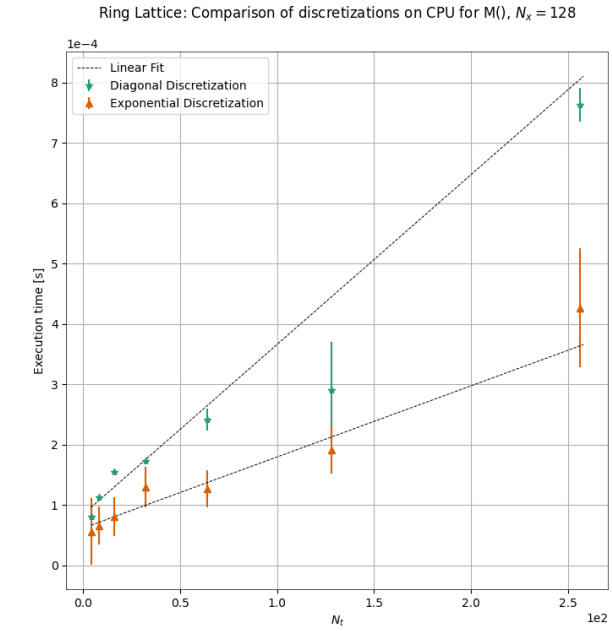
5.2 Results

The Benchmarks were executed for *Ring*, *Square* and *Cube* Lattices and for various lattice sizes. In this section, Benchmark results for `M()`, `MdaggerM()` and `logDetM()` on CPU as well as GPU are discussed. The results presented in this section were obtained for the *Ring* lattice, while the results obtained for *Square* and the *Cube* lattices are given in appendix (B). This section is divided in two parts, where results are presented in two ways for better understanding:

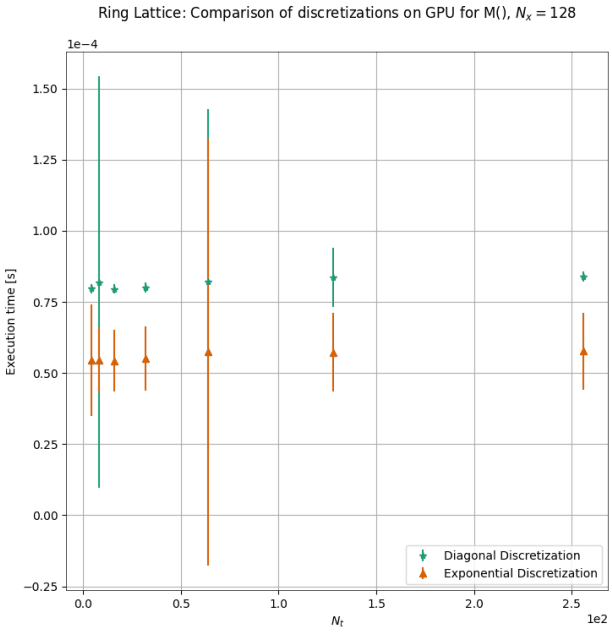
- Comparison of Diagonal discretization versus Exponential discretization (on CPU and GPU)
- Comparison of CPU versus GPU (for both of the discretizations)

5.2.1 Diagonal discretization versus Exponential discretization

In this section, execution times obtained for function call to `M()`, `MdaggerM()` and `logDetM()` for each discretization are compared. The analysis is done on CPU and GPU.

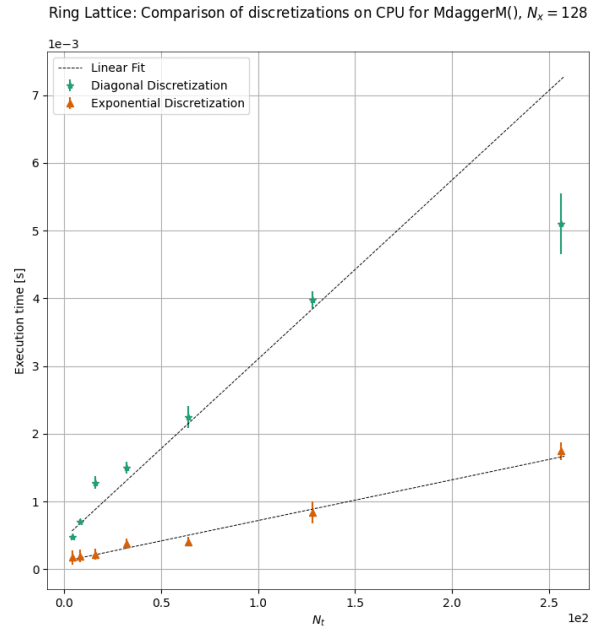


(a) CPU

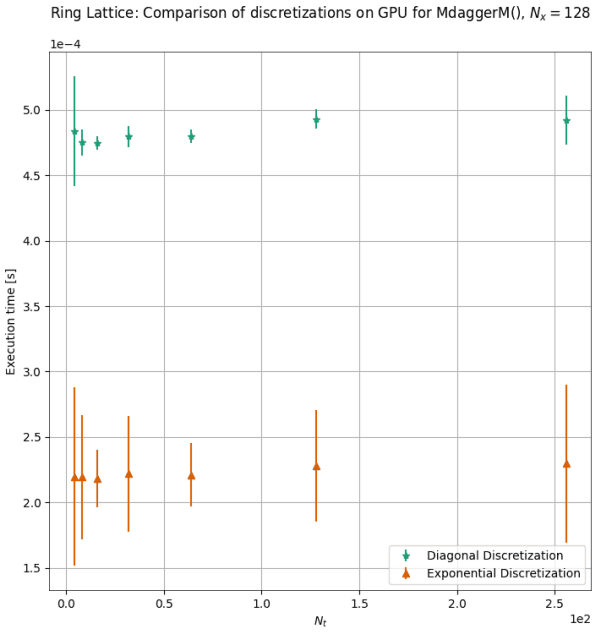


(b) GPU

Figure 5.1: Comparison of Diagonal and Exponential discretizations for function call to `M()` for Ring Lattice when $N_x = 128$

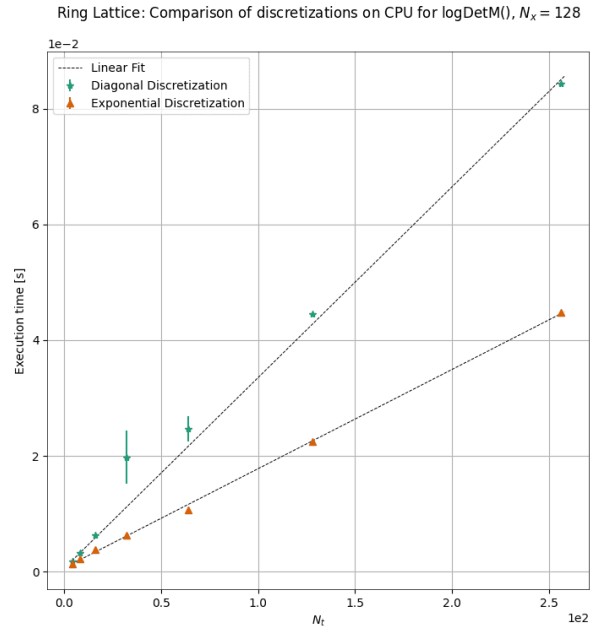


(a) CPU

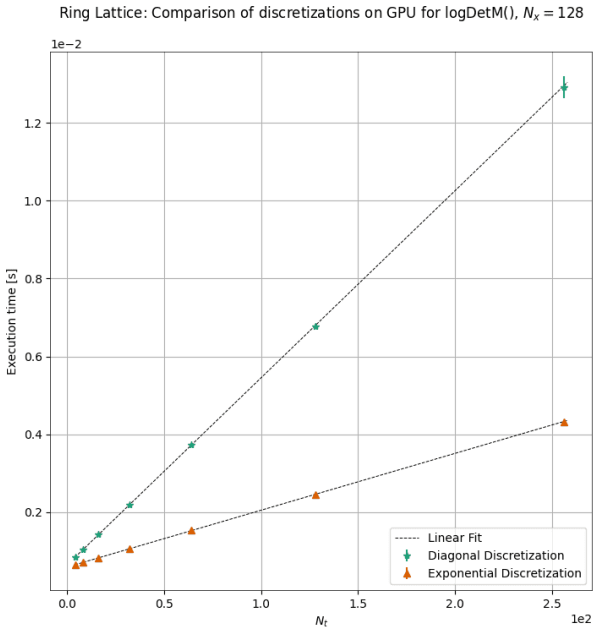


(b) GPU

Figure 5.2: Comparison of Diagonal and Exponential discretizations for function call to `MdaggerM()` for Ring Lattice when $N_x = 128$



(a) CPU



(b) GPU

Figure 5.3: Comparison of Diagonal and Exponential discretizations for function call to `logDetM()` for Ring Lattice when $N_x = 128$

Inference

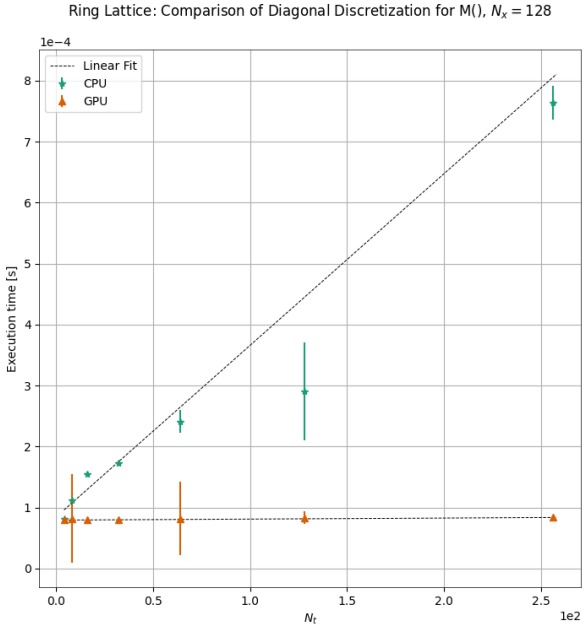
In plot 5.1(a), one observes that the execution time for the function call to `M()` varies linearly with the number of time slices, N_t, for both of the discretizations on CPU. This behaviour is expected due to the matrix-vector multiplication of $O(N_t \times N_x^2)$ inside the function `M()` as can be seen in code listings 3.1 and 3.5. We also observe that the exponential discretization is ~ 1.8 times faster than the diagonal discretization for $N_x = 128$ and $N_t = 256$. For GPU as well, the exponential discretization is (~ 1.4 times) faster as can be seen in 5.1(b) for $N_x = 128$ and $N_t = 256$. Although here, the execution times for both of the discretizations do not vary a lot with respect to N_t. Hence, we can infer that most of the time is being spent on accessing data rather than computation. Once the data has been collected, the computation is so fast that the number of time slices (N_t) do not make a difference, at least in the range that we studied. We observe very similar trend for the function call to `MdaggerM()` in plots 5.2(a) and 5.2(b) respectively. (Although we see some saturation for diagonal discretization on CPU, but we would need more data points to confirm that).
From plots B.3(a) and B.3(b), it can be seen that the execution time for function call to `logDetM()` varies linearly with N_t for both of the discretizations. This behavior is expected due to the loop which runs over N_t as can be seen in the code listings 3.9 and 3.10. We also expect the execution time to vary linearly with N_x^3, in this case due to the presence of a matrix-vector multiplication of $O(N_x^3)$. Here, exponential discretization is a little faster than the diagonal discretization because of an extra term in the expression for $K_{x'x}$ (equation 3.2) in case of the diagonal discretization. Due to this term, there is an additional element-wise multiplication in the `for` loop over N_t as can be seen in code listing 3.9.

Overall, exponential discretization performs a little better than the diagonal discretization because we are using the dense form of hopping matrix κ in our code. Performance for the diagonal discretization is expected to be better than the exponential, once the sparse hopping matrix κ is implemented.

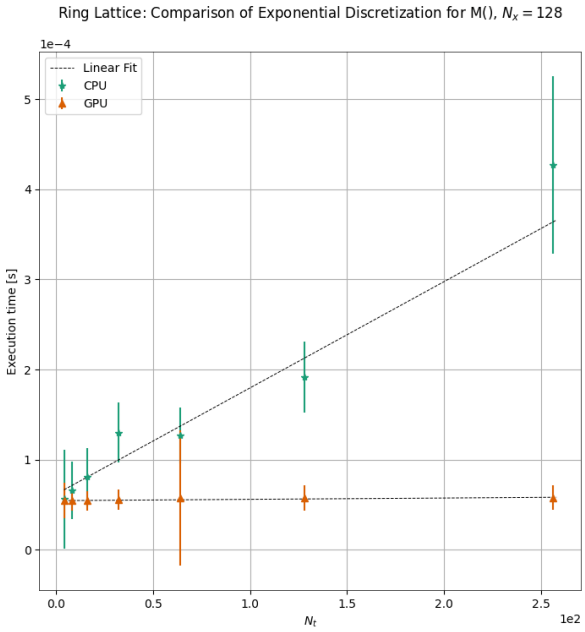
Behavior of results for the *Square* and the *Cube* lattices is similar to what we observed for the *Ring* lattice.

5.2.2 CPU versus GPU

The execution time with respect to N_t for both discretizations on CPU and GPU has already been discussed in the previous sub-section. We found out that the exponential discretization is faster in all cases. In this section, instead of comparing diagonal versus exponential, we focus on the comparison of execution times on CPU and GPU for `M`, `MdaggerM()` and `logDetM()`.

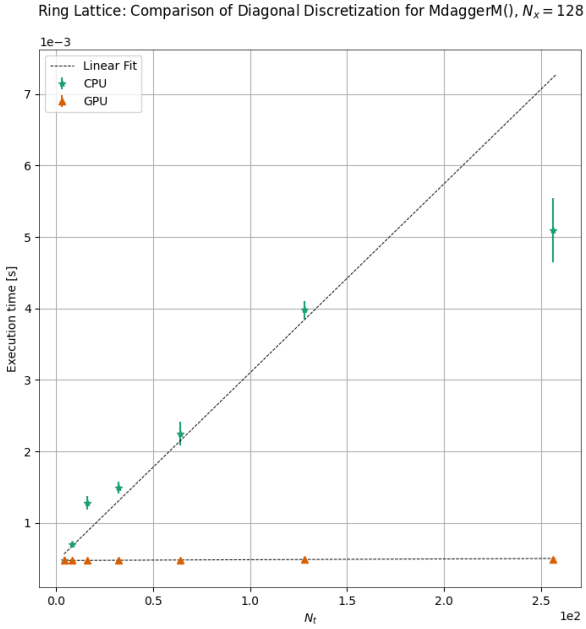


(a) Diagonal Discretization

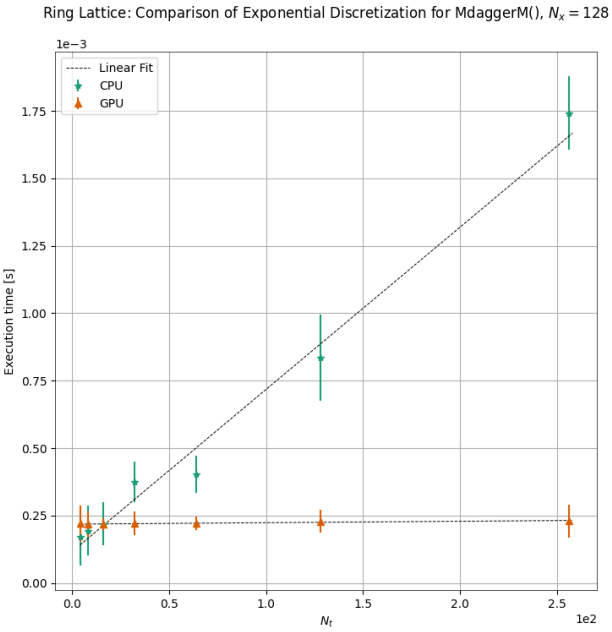


(b) Exponential Discretization

Figure 5.4: Comparison of `M()` on CPU versus GPU for Ring Lattice when $N_x = 128$

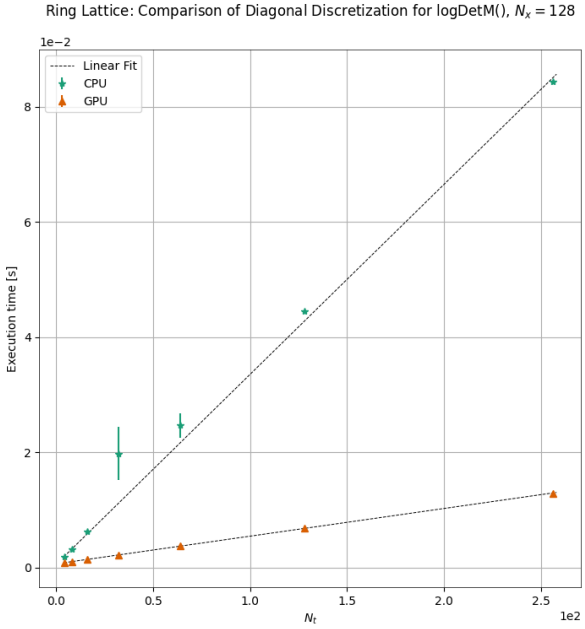


(a) Diagonal Discretization

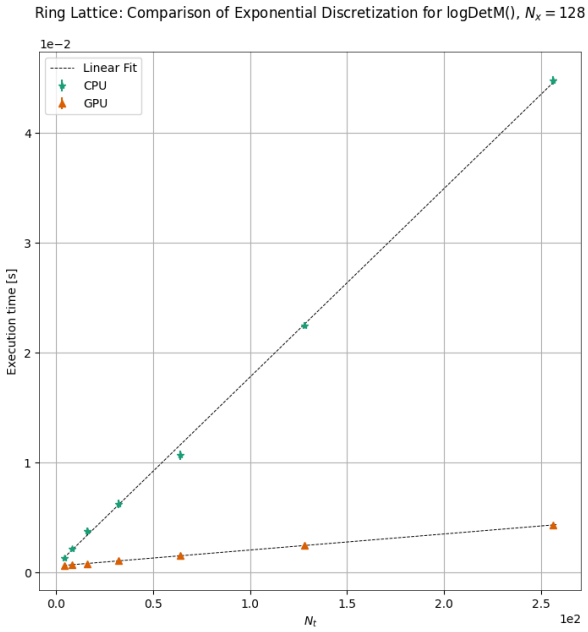


(b) Exponential Discretization

Figure 5.5: Comparison of `MdaggerM()` on CPU versus GPU for Ring Lattice when $N_x = 128$



(a) Diagonal Discretization



(b) Exponential Discretization

Figure 5.6: Comparison of `logDetM()` on CPU versus GPU for Ring Lattice when $N_x = 128$

Inference

In plots 5.4(a) and 5.4(b), one observes that the execution time for the function call to `M()` is higher for CPU as compared to GPU (as expected). And as explained in the previous section, the execution time does not seem to increase much when N_t increases in case of GPU. On the contrary, the slope for CPU is steep, which leads to slow-down with increase in N_t. Therefore, running the simulation on GPU for larger N_t will be more optimal. We observe similar behavior for `MdaggerM()` as shown in plots 5.5(a) and 5.5(b).

From plots 5.6(a) and 5.6(a) for `logDetM()`, we see that the execution time varies linearly on CPU as well as GPU and as expected, the slope for CPU is steeper, making it very slow for higher N_t.

Behavior of results for the *Square* and the *Cube* lattices is similar to what we observed for the *Ring* lattice.

5.2.3 Summary of Execution Times

The right-most values (for $N_t = 256$) on the plots discussed in the previous section can be summarized in the table 5.1. Results for the *Square* and the *Cube* lattices have also been presented in the the tables 5.2 and 5.3:

Ring Lattice

Device	Discretization	`M()`	`MdaggerM()`	`logDetM()`
CPU	Diagonal	7.64e−4 ± 2.74e−5 s	5.10e−3 ± 4.50e−4 s	8.44e−2 ± 1.46e−4 s
	Exponential	4.27e−4 ± 9.85e−5 s	1.74e−3 ± 1.37e−4 s	4.48e−2 ± 3.93e−4 s
GPU	Diagonal	8.38e−5 ± 1.76e−6 s	4.92e−4 ± 1.85e−5 s	1.29e−2 ± 2.74e−4 s
	Exponential	5.77e−5 ± 1.35e−5 s	2.30e−4 ± 6.04e−5 s	4.32e−3 ± 3.65e−5 s

Table 5.1: Execution Times for *Ring* Lattice: $N_x = 128$, $N_t = 256$

Square Lattice

Device	Discretization	`M()`	`MdaggerM()`	`logDetM()`
CPU	Diagonal	2.49e−4 ± 2.11e−5 s	1.72e−3 ± 1.25e−4 s	1.98e−2 ± 2.33e−4 s
	Exponential	1.73e−4 ± 2.50e−5 s	1.11e−3 ± 5.40e−5 s	1.37e−2 ± 1.75e−4 s
GPU	Diagonal	8.03e−5 ± 1.72e−6 s	4.77e−4 ± 1.58e−5 s	1.17e−2 ± 5.57e−5 s
	Exponential	5.52e−5 ± 1.35e−5 s	2.22e−4 ± 4.02e−5 s	3.60e−3 ± 2.94e−5 s

Table 5.2: Execution Times for *Square* Lattice: $N_x = 8$, $N_t = 256$

Cube Lattice

Device	Discretization	`M()`	`MdaggerM()`	`logDetM()`
CPU	Diagonal	2.97e−3 ± 7.55e−5 s	2.01e−2 ± 1.19e−3 s	1.38 ± 2.80e−2 s
	Exponential	2.34e−3 ± 5.64e−4 s	7.17e−3 ± 1.10e−3 s	1.17 ± 4.38e−3 s
GPU	Diagonal	8.53e−5 ± 3.88e−6 s	5.77e−4 ± 3.46e−5 s	3.47e−2 ± 4.80e−5 s
	Exponential	6.66e−5 ± 4.11e−5 s	2.63e−4 ± 5.06e−5 s	2.63e−2 ± 9.99e−5 s

Table 5.3: Execution Times for *Cube* Lattice: $N_x = 8$, $N_t = 256$

CHAPTER 6

Conclusion and Outlook

In this thesis, we worked on the formalism of the diagonal and the exponential discretizations of the fermion matrix which involved: coding up the discretizations, testing the code and benchmarking the code.

Testing was one of the most crucial parts while coding up the discretizations. It not only helped to test if the code was working in an expected manner, but also gave us insight on various underlying issues that could have gone unaddressed had we not implemented the tests. For example, we found out that there is an issue with the accuracy of *Torch* `logdet`, and due to this loss of precision, the *Phi Plus 2*pi* and the *Time shift invariance* tests fail for $float$ data types for the diagonal discretization. (Hence, using double data types is recommended in this case). We observe this loss in accuracy for many other methods of $Torch$.

The benchmarks for the exponential and the diagonal discretizations were set up on CPU and on GPU components on the HPC cluster JURECA-DC. On both processing units, one observes that the exponential discretization is faster than the diagonal discretization. However, we expect that the latter would be faster once the sparse hopping matrix, κ is implemented.

It was observed that the computations run slower than what was expected. We measure a lower bandwidth than expected for both CPU and GPU on JURECA–DC. A comparison for matrix vector multiplication for $C++$ vectors and $Torch$ tensors was also done, and a much lower bandwidth was observed in case of $Torch$ [22]. We suspect that it could either be an underlying issue with $Torch$ itself or a configuration issue on JURECA-DC Module. We observed a similar behavior on JUWELS. Contact with the support teams of both systems and the forums of Torch users was established, but no clarification response was obtained. It is still an open issue.

Lastly, we can conclude that these computations on GPU give promising results especially when N_t is large. We also made a (successful) attempt in computing `logDetM()` for $Cube$ lattice with ($N_t = 64$, $N_x = 8$) $16^3 \times 64 = 262{,}144$ lattice sites but the GPU ran out of memory when we tried to compute for larger N_t. For a $Cube$ lattice ($N_t = 2$, $N_x = 16$), we observe that GPU is ~ 350 times faster than CPU for both discretizations! Hence, we can most definitely expect reassuring results with the use of GPU for NSL in future, when it will be used to simulate more complex lattice systems.

Appendix A

A.1 Building : Steps for compilation of NSL on Jureca DC module

NSL can be built by following these simple steps:

1. Load the necessary modules required for building the library:

```
module load CMake
module load PyTorch
```

   The `Cmake` should be higher than version 3.18.0 and g++ should be higher than 10.3.0.

2. Clone the library to your system:

```
git clone git@github.com:Marcel-Rodekamp/NSL.git && cd NSL
```

3. Create a build directory and change the current directory to the build directory:

```
mkdir build && cd build
```

4. Generate build files by calling `CMake` (higher version than 13.8). The build files can be generated in `Debug` mode as well as `Release` mode.

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

5. Build the directory

```
make -j n
```

   Here, substitute `n` with the number of cores for building in parallel.

A.2 Testing

The tests can be executed by running:

```
make test
```

Alternatively, one can submit the job script similar to the one shown below for executing tests:

Listing A.1: SLURM jobscript for running Tests

```
#!/bin/bash

#Job parameters
#SBATCH --job-name=test_NSL
#SBATCH --output=output_file.out
#SBATCH --error=error_file.err
#SBATCH --mail-user=userName@provider.com --mail-type=FAIL

#Resources
#SBATCH --account=YourComputeTimeAccount
#SBATCH --time=00:10:00
#SBATCH --partition=dc-cpu
#SBATCH --nodes=1
#SBATCH --ntasks=1

# load required models
ml PyTorch

#Job steps
echo -e "Calling␣FermionMatrix/test_fermionMatrixHubbardDiag:\n"
srun ./Tensor/test_fermionMatrixHubbardDiag
echo -e "Calling␣FermionMatrix/test_fermionMatrixHubbardExp:\n"
srun ./LinAlg/test_fermionMatrixHubbardExp
```

The above job-file can be submitted by typing the command:

```
sbatch -A YourComputeTimeAccount jobfile.sh
```

A.3 Benchmarking

One can run the Benchmarks in a similar manner as one runs Tests by replacing the names of the Test executables to the name of the Benchmark executables and modifying the execution parameters in A.1. Alternatively, resources can also be requested using `salloc` command (with suitable parameters):

```
salloc -p dc-cpu-devel --nodes=1 --time=2:0:0 -A accountName
```

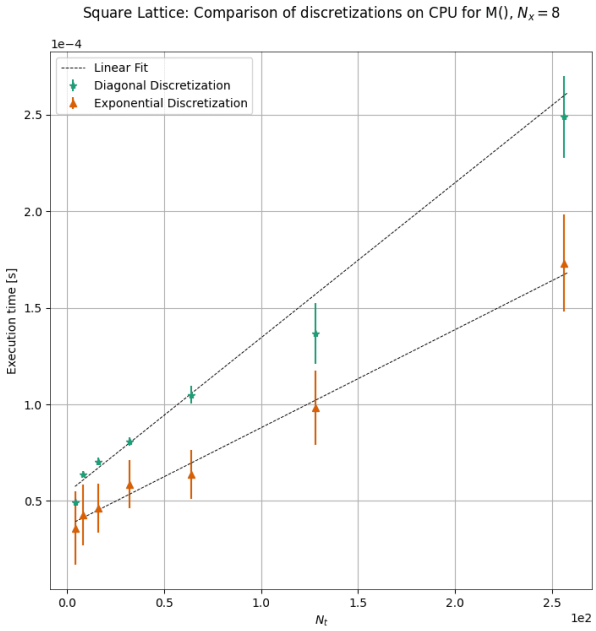
To request an allocation on a GPU, one can add `-partition=dc-gpu` in a job script or alternatively, the following command also works (with suitable parameters):

```
salloc -p dc-gpu-devel --gres=gpu:1 --time=2:0:0 -A accountName
```

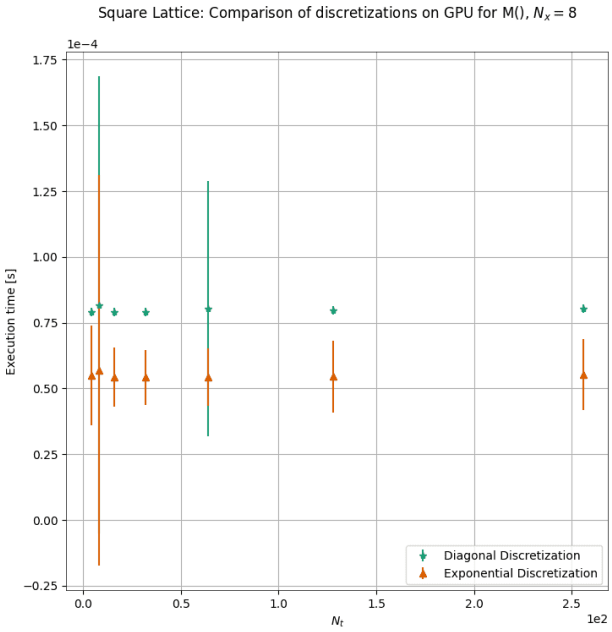
Appendix B

B.1 Benchmark results for Square and Cube Lattices

The Benchmarks were run for *Square* and *Cube* Lattices and for various lattice sizes. In this section, Benchmark results for `M()`, `MdaggerM()` and `logDetM()` on CPU as well as GPU are presented for both of these lattice types. The analysis of these plots for *Square* and *Cube* lattices can be done in similar way as done in 5.2.1 and 5.2.2. Here, the cube lattice ($N_x = 8$) shows more speed-up on GPU as compared to the *Ring* lattice and the *Square* lattice with same N_x. This is because, for a same N_x for each lattice type, *Cube* lattice has the most number of lattice sites and the speed-up increases with an increase in lattice sites. The following analysis was done for $N_x = 8$.

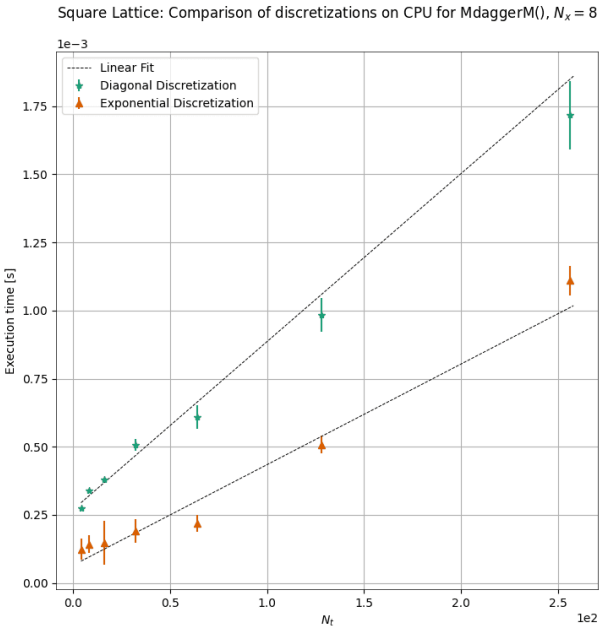


(a) CPU

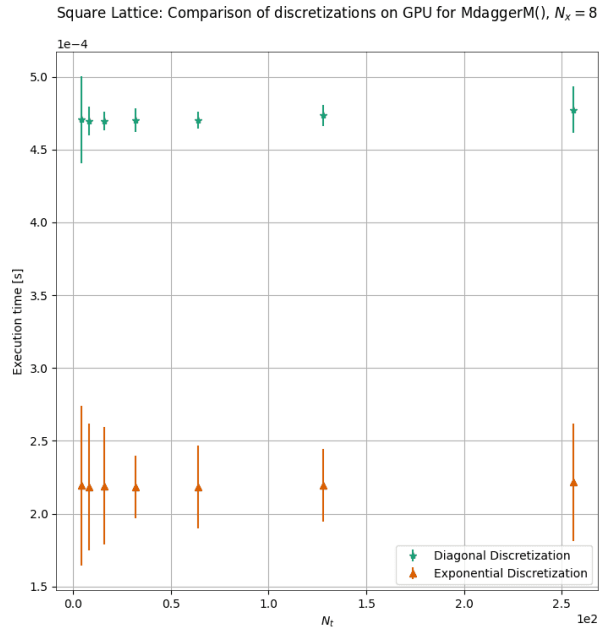


(b) GPU

Figure B.1: Comparison of Diagonal and Exponential discretizations for function call to `M()` for Square Lattice when $N_x = 8$

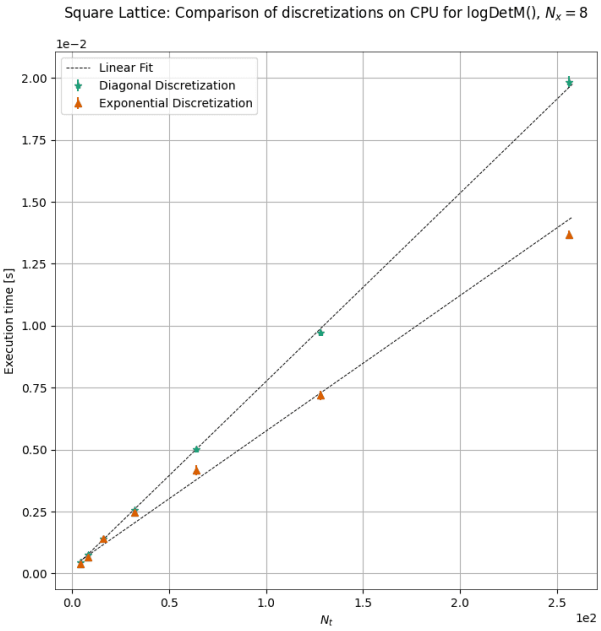


(a) CPU

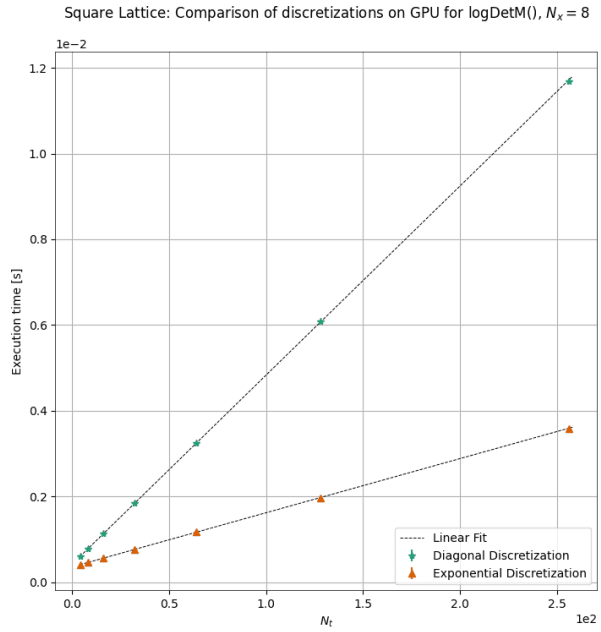


(b) GPU

Figure B.2: Comparison of Diagonal and Exponential discretizations for function call to `MdaggerM()` for Square Lattice when $N_x = 8$

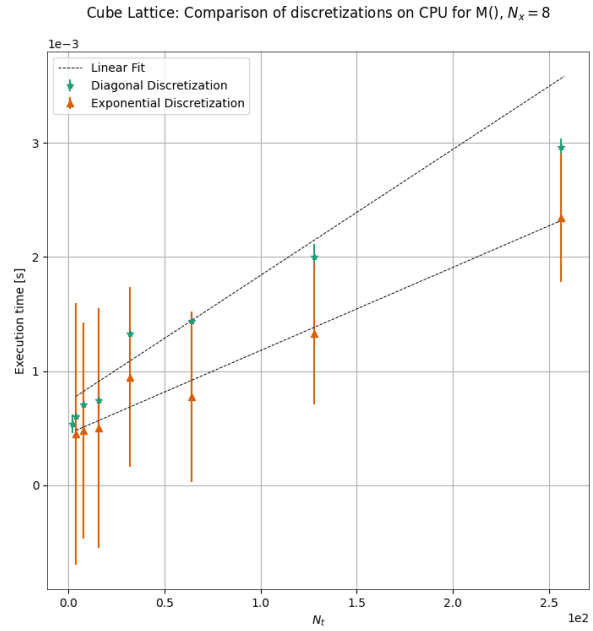


(a) CPU

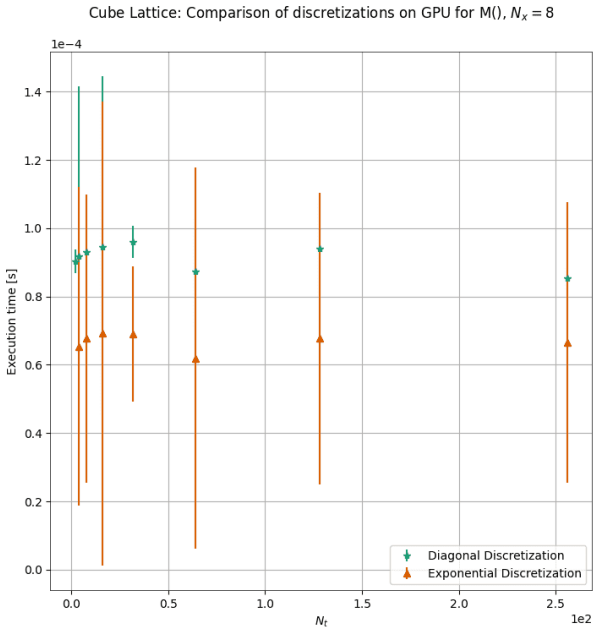


(b) GPU

Figure B.3: Comparison of Diagonal and Exponential discretizations for function call to `logDetM()` for Square Lattice when $N_x = 8$

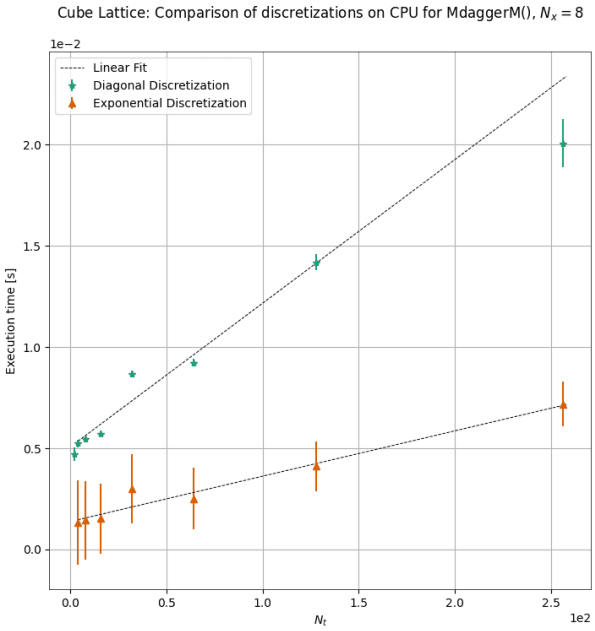


(a) CPU

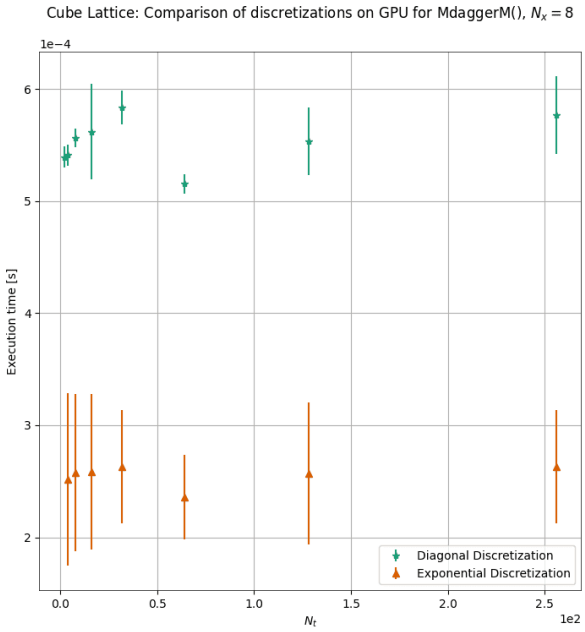


(b) GPU

Figure B.4: Comparison of Diagonal and Exponential discretizations for function call to `M()` for Cube Lattice when $N_x = 8$

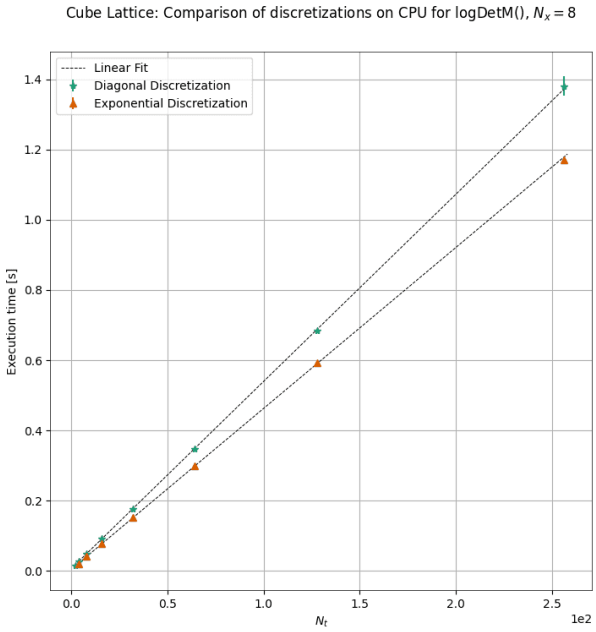


(a) CPU

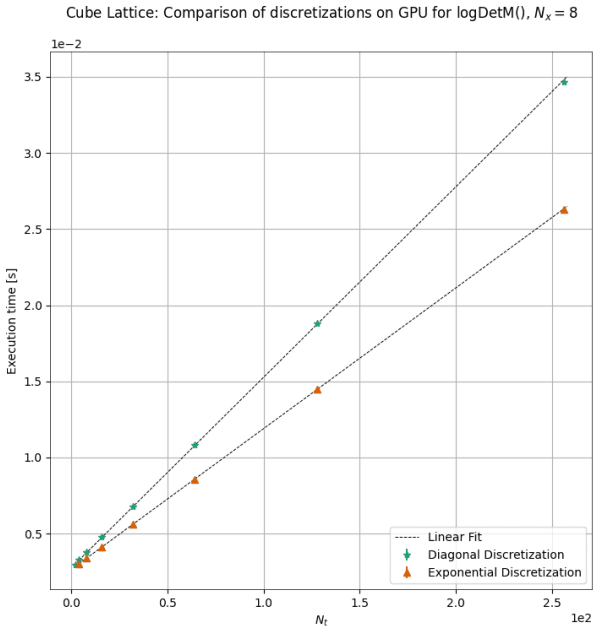


(b) GPU

Figure B.5: Comparison of Diagonal and Exponential discretizations for function call to `MdaggerM()` for Cube Lattice when $N_x = 8$



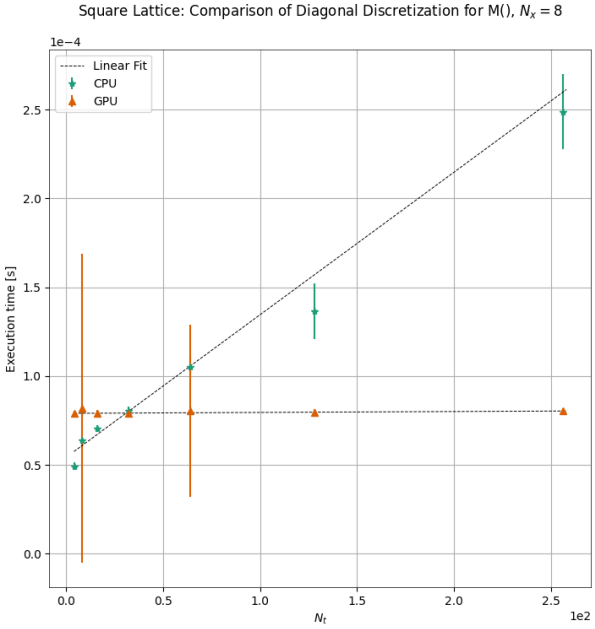
(a) CPU



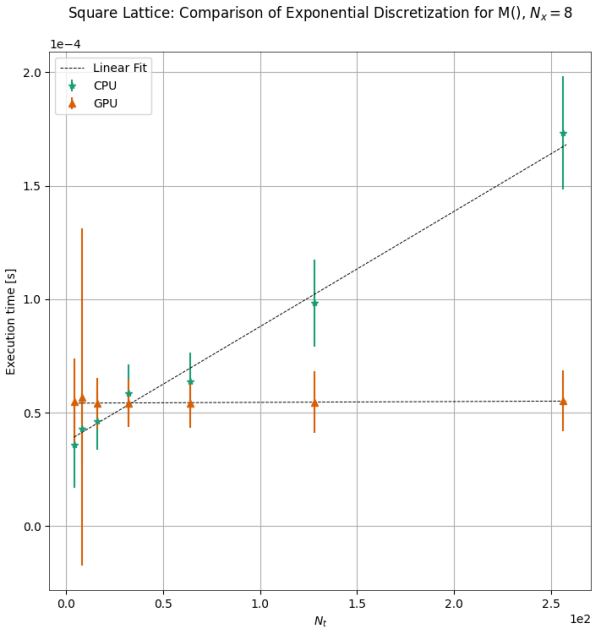
(b) GPU

Figure B.6: Comparison of Diagonal and Exponential discretizations for function call to `logDetM()` for Cube Lattice when $N_x = 8$

.

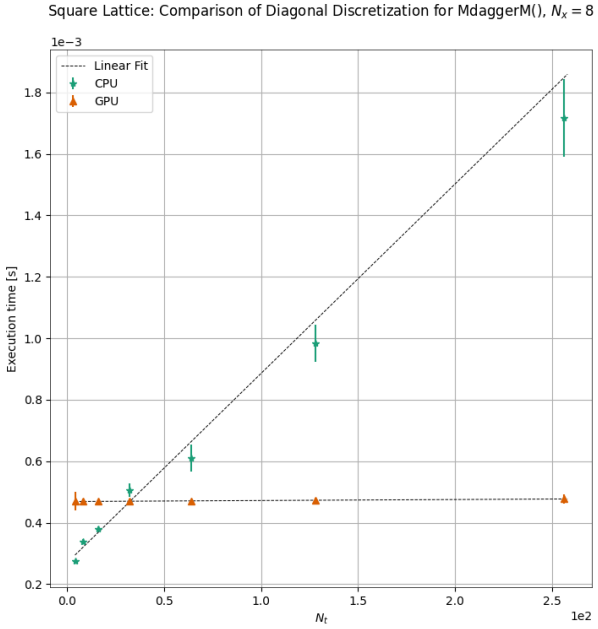


(a) Diagonal Discretization

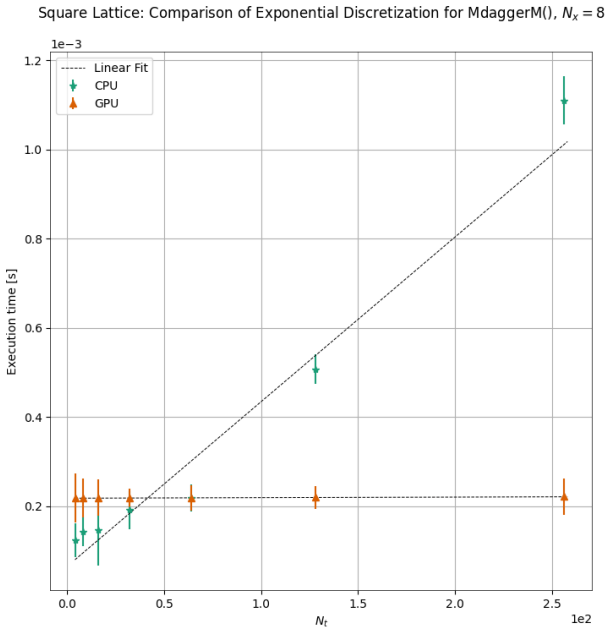


(b) Exponential Discretization

Figure B.7: Comparison of `M()` on CPU versus GPU when $N_x = 8$

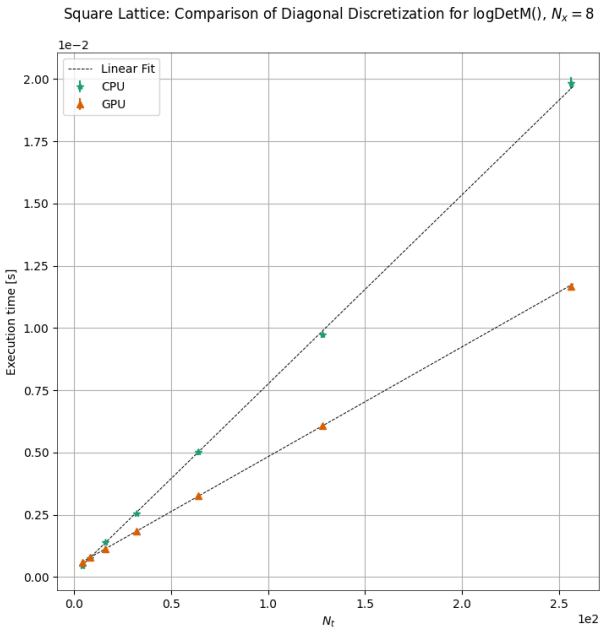


(a) Diagonal Discretization

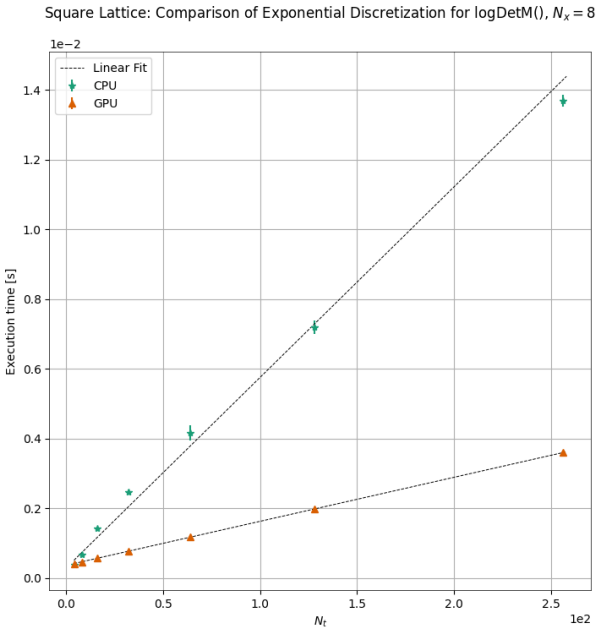


(b) Exponential Discretization

Figure B.8: Comparison of `M()` on CPU versus GPU when $N_x = 8$

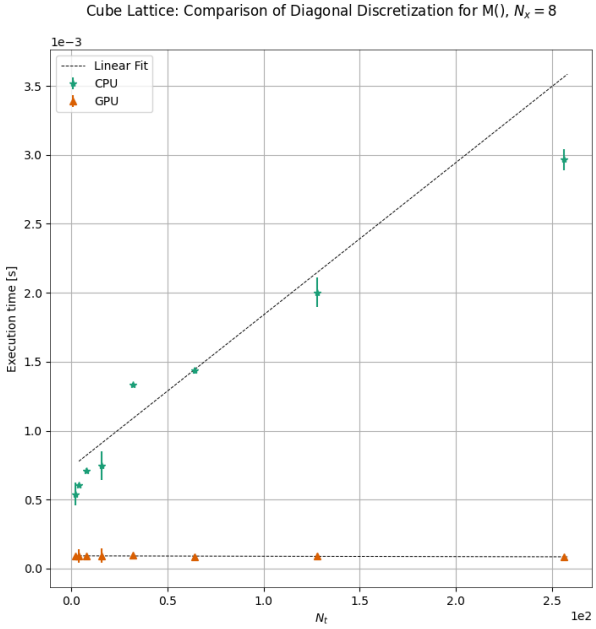


(a) Diagonal Discretization

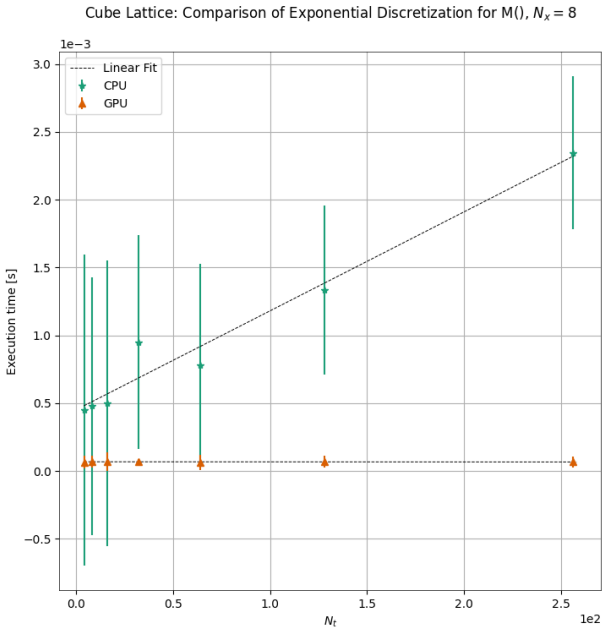


(b) Exponential Discretization

Figure B.9: Comparison of `M()` on CPU versus GPU when $N_x = 8$

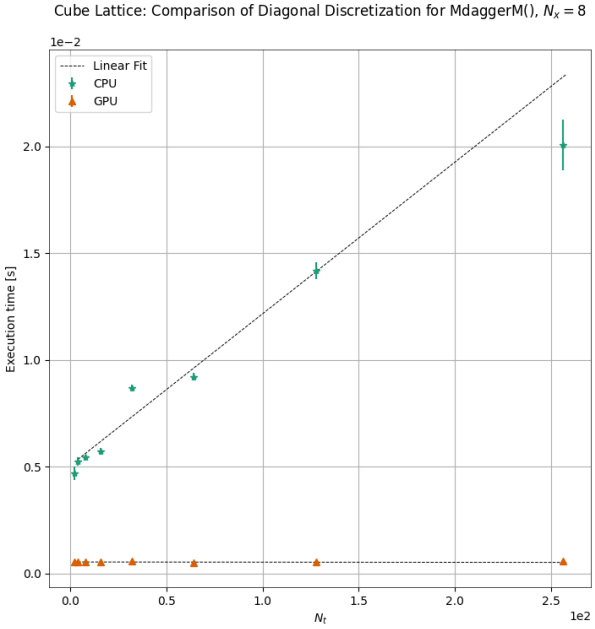


(a) Diagonal Discretization

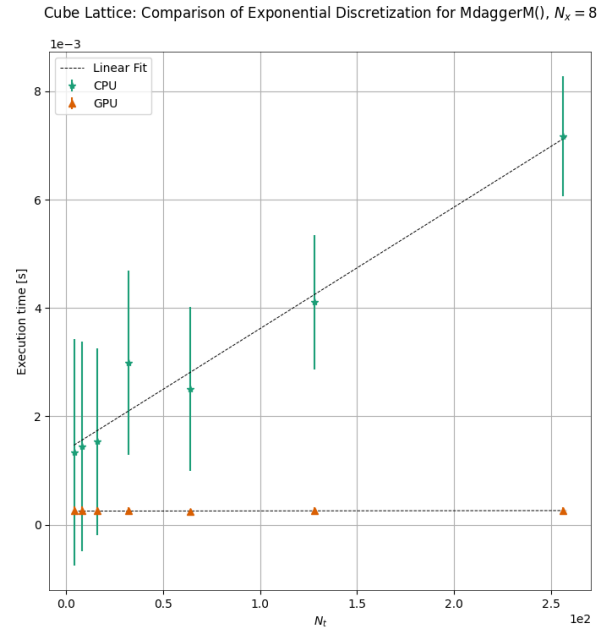


(b) Exponential Discretization

Figure B.10: Comparison of `M()` on CPU versus GPU when $N_x = 8$

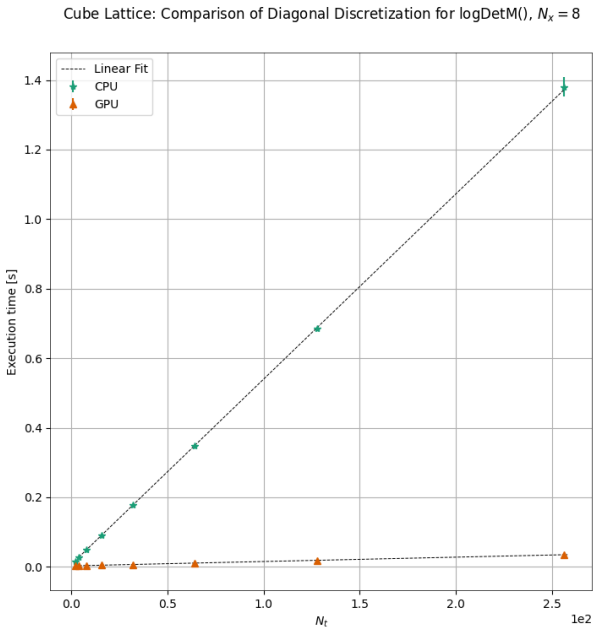


(a) Diagonal Discretization

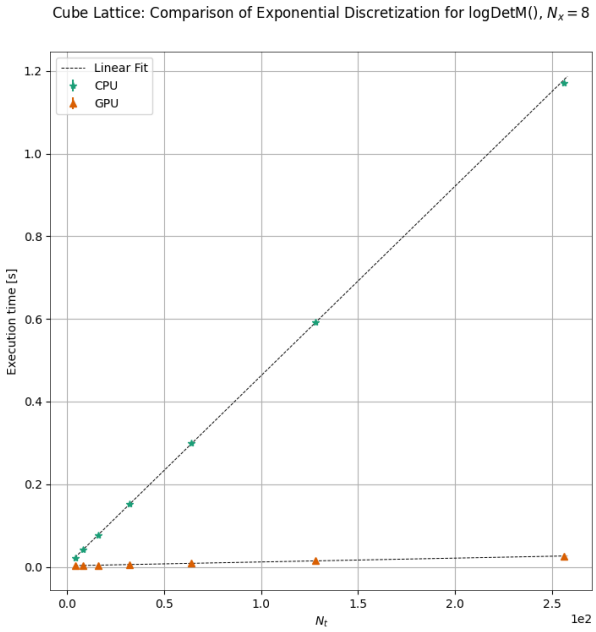


(b) Exponential Discretization

Figure B.11: Comparison of `M()` on CPU versus GPU when $N_x = 8$



(a) Diagonal Discretization



(b) Exponential Discretization

Figure B.12: Comparison of `M()` on CPU versus GPU when $N_x = 8$

Bibliography

[1] Catch2. URL: https://github.com/catchorg/Catch2.

[2] Jureca user documentation. URL: https://apps.fz-juelich.de/jsc/hps/jureca/configuration.html.

[3] Nanosystem simulation library. URL: https://github.com/Marcel-Rodekamp/NSL.git.

[4] Pytorch. URL: https://pytorch.org/.

[5] Slurm workload manager. URL: https://slurm.schedmd.com/.

[6] numeric_limits class documentation. 2021. URL: https://docs.microsoft.com/en-us/cpp/standard-library/numeric-limits-class?view=msvc-170#digits10.

[7] Evan Berkowitz, Christopher Körber, Stefan Krieg, Peter Labus, Timo A. Lähde, and Thomas Luu. Extracting the single-particle gap in carbon nanotubes with lattice quantum monte carlo. *EPJ Web of Conferences*, 175:03009, 2018. URL: https://doi.org/10.1051%2Fepjconf%2F201817503009, doi:10.1051/epjconf/201817503009.

[8] Stefan Beyl, Florian Goth, and Fakher F. Assaad. Revisiting the hybrid quantum monte carlo method for hubbard and electron-phonon models. *Physical Review B*, 97(8), feb 2018. URL: https://doi.org/10.1103%2Fphysrevb.97.085144, doi:10.1103/physrevb.97.085144.

[9] Richard Brower, Claudio Rebbi, and David Schaich. Hybrid monte carlo simulation on the graphene hexagonal lattice. 2012. URL: https://arxiv.org/abs/1204.5424, doi:10.48550/ARXIV.1204.5424.

[10] Richard Brower, Claudio Rebbi, and David Schaich. Hybrid monte carlo simulation on the graphene hexagonal lattice. 2012. URL: https://arxiv.org/abs/1204.5424, doi:10.48550/ARXIV.1204.5424.

[11] Herbert Fotso, Ka-Ming Tam, and Juana Moreno. Beyond quantum cluster theories: Multiscale approaches for strongly correlated systems, 11 2020.

[12] gcc.gnu.org. Chrono source code. URL: https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a00798_source.html.

[13] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[14] J. Hubbard. Electron correlations in narrow energy bands. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 276(1365):238–257, 1963. URL: http://www.jstor.org/stable/2414761.

[15] Wynen Jan-Lukas, Berkowitz Evan, Rodekamp Marcel, Gäntgen Christoph, and CI Travis. Isle. URL: https://github.com/evanberkowitz/isle.

[16] Jun Lu. Matrix decomposition and applications, 2022. URL: https://arxiv.org/abs/2201.00145, doi:10.48550/ARXIV.2201.00145.

[17] Thomas Luu and Timo A. Lähde. Quantum monte carlo calculations for carbon nanotubes. *Physical Review B*, 93(15), apr 2016. URL: https://doi.org/10.1103%2Fphysrevb.93.155106, doi:10.1103/physrevb.93.155106.

[18] Z. Y. Meng, T. C. Lang, S. Wessel, F. F. Assaad, and A. Muramatsu. Quantum spin liquid emerging in two-dimensional correlated dirac fermions. *Nature*, 464(7290):847–851, apr 2010. URL: https://doi.org/10.1038%2Fnature08942, doi:10.1038/nature08942.

[19] Radford M. Neal. Mcmc using hamiltonian dynamics. URL: https://arxiv.org/abs/1206.1901v1.

[20] NVIDIA. Nvidia tesla v100 gpu architecture.

[21] Moritz Reh. The doped hubbard model.

[22] Marcel Rodekamp. URL: https://github.com/Marcel-Rodekamp/TorchBandwidth.

[23] Alex Rogozhnikov. Hamiltonian monte carlo explained, 12 2016.

[24] Dominik Smith and Lorenz von Smekal. Monte carlo simulation of the tight-binding model of graphene with partially screened coulomb interactions. *Physical Review B*, 89(19), may 2014. URL: https://doi.org/10.1103%2Fphysrevb.89.195429, doi:10.1103/physrevb.89.195429.

[25] Philipp Thörnig. Jureca: Data centric and booster modules implementing the modular supercomputing architecture at jÃijlich supercomputing centre. october 2021. URL: https://juser.fz-juelich.de/record/902211, doi:https://doi.org/10.17815/jlsrf-7-182.

[26] M. V. Ulybyshev, P. V. Buividovich, M. I. Katsnelson, and M. I. Polikarpov. Monte carlo study of the semimetal-insulator phase transition in monolayer graphene with a realistic interelectron interaction potential. *Physical Review Letters*, 111(5), jul 2013. URL: https://doi.org/10.1103%2Fphysrevlett.111.056801, doi:10.1103/physrevlett.111.056801.

[27] M. V. Ulybyshev and S. N. Valgushev. Path integral representation for the hubbard model with reduced number of lefschetz thimbles, 2017. URL: https://arxiv.org/abs/1712.02188, doi:10.48550/ARXIV.1712.02188.

[28] Jan-Lukas Wynen. Algorithms for fermion action in hubbard model. 2021.

[29] Jan-Lukas Wynen, Evan Berkowitz, Christopher Körber, Timo A. Lähde, and Thomas Luu. Avoiding ergodicity problems in lattice discretizations of the hubbard model. *Physical Review B*, 100(7), aug 2019. URL: https://doi.org/10.1103%2Fphysrevb.100.075141, doi:10.1103/physrevb.100.075141.

List of Figures

2.1 Pictorial representation of the kinetic energy t and the on-site repulsion U. [11] . . . 4
2.2 Square and Hexagonal Lattices can be divided into two sub-lattices A and B (represented by blue and orange dots here) such that the neighbours of sites of A are members of B only and vice versa. This is not possible for the Triangular Lattice as depicted by the black dots: Left: Square Lattice, Middle: Hexagonal Lattice and Right: Triangular Lattice [21] . . . 4
2.3 Results using Euler's and Leapfrog methods for approximating Hamiltonian dynamics, when $H(q,p) = q^2/2 + p^2/2$. The initial state was $q = 0$, $p = 1$ [19] . . . 9
2.4 NVIDIA Volta GV100 Full GPU with 84 SM Units [20] . . . 13

5.1 Ring Lattice: Comparison of Diagonal and Exponential discretizations for `M()` . . . 43
5.2 Ring Lattice: Comparison of Diagonal and Exponential discretizations for `MdaggerM()` 44
5.3 Ring Lattice: Comparison of Diagonal and Exponential discretizations for `logDetM()` 45
5.4 Ring Lattice: Comparison of `M()` on CPU versus GPU . . . 47
5.5 Ring Lattice: Comparison of `MdaggerM()` on CPU versus GPU . . . 48
5.6 Ring Lattice: Comparison of `logDetM()` on CPU versus GPU . . . 49

B.1 Square Lattice: Comparison of Diagonal and Exponential discretizations for `M()` . . . 58
B.2 Square Lattice: Comparison of Diagonal and Exponential discretizations for `MdaggerM()` 59
B.3 Square Lattice: Comparison of Diagonal and Exponential discretizations for `logDetM()` 60
B.4 Cube Lattice: Comparison of Diagonal and Exponential discretizations for `M()` . . . 61
B.5 Cube Lattice: Comparison of Diagonal and Exponential discretizations for `MdaggerM()` 62
B.6 Cube Lattice: Comparison of Diagonal and Exponential discretizations for `logDetM()` 63
B.7 Square Lattice: Comparison of `M()` on CPU versus GPU . . . 64
B.8 Square Lattice: Comparison of `MdaggerM()` on CPU versus GPU . . . 65
B.9 Square Lattice: Comparison of `logDetM()` on CPU versus GPU . . . 66
B.10 Cube Lattice: Comparison of `M()` on CPU versus GPU . . . 67
B.11 Cube Lattice: Comparison of `MdaggerM()` on CPU versus GPU . . . 68
B.12 Cube Lattice: Comparison of `logDetM()` on CPU versus GPU . . . 69

Listings

2.1 Creates a `Tensor` of dimensions $(x \times y \times z)$ whose elements are of the type `double` . 11
2.2 Creates a GPU `Tensor` of dimensions $(x \times y \times z)$ whose elements are of the type `double` 11
2.3 Construction of a `HubbardDiag` object on CPU 12
2.4 Construction of a `HubbardDiag` object on GPU 12

3.1 Implementation of `NSL::FermionMatrix::HubbardDiag::M()` 16
3.2 Implementation of `NSL::FermionMatrix::HubbardDiag::Mdagger()` 17
3.3 Implementation of `NSL::FermionMatrix::HubbardDiag::MMdagger()` 18
3.4 Implementation of `NSL::FermionMatrix::HubbardDiag::MdaggerM()` 19
3.5 Implementation of `NSL::FermionMatrix::HubbardExp::M()` 20
3.6 Implementation of `NSL::FermionMatrix::HubbardExp::Mdagger()` 21
3.7 Implementation of `NSL::FermionMatrix::HubbardExp::MMdagger()` 22
3.8 Implementation of `NSL::FermionMatrix::HubbardExp::MdaggerM()` 22
3.9 Implementation of `NSL::FermionMatrix::HubbardDiag::logDetM()` 25
3.10 Implementation of `NSL::FermionMatrix::HubbardExp::logDetM()` 26

4.1 Implementation of `test_fermionMatrixHubbardExp_M()` 30
4.2 Implementation of `test_fermionMatrixHubbardExp_M_dense()` 31
4.3 Implementation of `test_fermionMatrixHubbardExp_Mdagger()` 31
4.4 Implementation of `test_fermionMatrixHubbardExp_MMdagger()` 32
4.5 Implementation of `test_fermionMatrixHubbardExp_MdaggeM()` 33
4.6 Implementation of `test_logDetM_phi_plus_two_pi()` 34
4.7 Implementation of `test_logDetM_time_shift_invariance()` 35
4.8 Implementation of `test_logDetM_noninteracting()` 36
4.9 Implementation of `test_logDetM_noninteracting()` 37
4.10 Implementation of `test_logDetM_uniform_timeslices()` 38
4.11 Implementation of `almost_equal()` 39

5.1 `Chrono` Time point declaration 41

[. 55
A.1 SLURM jobscript for running Tests 56

List of Tables

5.1 Execution Times for *Ring* Lattice: $N_x = 128$, $N_t = 256$ 50
5.2 Execution Times for *Square* Lattice: $N_x = 8$, $N_t = 256$ 50
5.3 Execution Times for *Cube* Lattice: $N_x = 8$, $N_t = 256$ 51

Acronyms

CNS Carbon Nano Structures. 9

CPU Central Processing Unit. 1, 11–13, 42, 46, 50, 53, 57, 77

CUDA Compute Unified Device Architecture. 13

GPU Graphics Processing Unit. 1, 10–13, 42, 46, 50, 53, 56, 57, 77

HBM High Bandwidth Memory. 13

HMC Hamiltonian Monte Carlo. 1, 6, 8, 9

HPC High Performance Computing. 13

JUWELS Jülich Wizard for European Leadership Science. 53

LHS Left Hand Side. 31–33, 36–38

LU Lower Upper. 23

MCMC Markov Chain Monte Carlo. 6, 7

MH Metropolis Hastings. 7, 8

NSL Nanosystem Simulation Library. vi, 1, 9, 10, 13, 29–31, 42, 53, 55

RHS Right Hand Side. 31–33, 36–38

SM Streaming Multiprocessor. 13, 75