A Compression Algorithm for Quantum Field Theoretic Calculations

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Xiao Pu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

John R. Hiller

July, 2013

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school. First I would like to express my sincere gratitude to my advisor, Dr. John R. Hiller, for his continuous support and guidance throughout my graduate study. I would like to thank my committee, Dr. Thomas Jordan and Dr. Bruce Peckham, for their valuable time. Last but not the least, I would like to thank the Department of Physics at the University of Minnesota Duluth for giving me teaching assistantships for my graduate studies.

Dedication

To my loving parents and my elder brother

Abstract

In this study, we present a compression algorithm to solve quantum field eigenvalue problems with dramatically reduced memory requirements. We represent light-cone quantized basis states as 2D matrices, in which one index stands for longitudinal momenta and the other represents transverse momenta. These matrices are stored as singular value decompositions, compressed to retain only the n_{svd} largest contributions. Basic Lanczos iterations are applied to obtain eigenvalues. At each Lanczos iteration, each sector of the Lanczos vector is projected into n_{svd} longitudinal and transverse vectors using singular value decomposition. The converged smallest real eigenvalues are obtained by these Lanczos iterations. The convergence of the ground state with respect to n_{svd} is then studied. It shows that our compression algorithm reduces memory cost significantly with little sacrifice in the accuracy of calculations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Discretized Light-Cone Quantization (DLCQ) [1–3] is a general method for solving quantum field theories for their mass spectrum and wave functions. In this method, quantization conditions are specified on the characteristic surface $x^+ \equiv (ct + z) = 0$, and by introducing periodic boundary conditions, a discrete basis set is induced. As the basis set is infinite, one needs to truncate the basis set by some procedure in order to solve quantum field problems numerically.

Once a quantum field problem is stated in the form of integral equations and the truncation is carried out, the basis set is used to discretize the Hamiltonian into a finite matrix, which leads to a numerical diagonalization problem. One can solve for the eigenvalues and eigenvectors of that matrix numerically, which becomes an approximation to the physical eigenvalues and wave functions.

To get a good approximation, truncation of the basis set is carried out in a way such that the important part of the basis set is not excluded. Usually the basis size can be very large even after truncation, and for many-body problems the basis size can be even larger. The Lanczos algorithm [4] [5] is commonly used to efficiently approximate extreme eigenvalues of large matrices. However, in quantum field theoretic calculations, the length of the Lanczos vectors(which are stored in the dynamical memory) and the size of the Hamiltonian matrix(which is stored in static memory) increase tremendously,

when the number of particles becomes large and when we want less truncation for accuracy. Therefore, quantum field theoretic calculations are prevented primarily by memory limitations, rather than processor speed.

The central idea of this thesis is to significantly reduce the memory cost, in order to enable numerical diagonalization of larger matrices in quantum field theoretic calculations. We modify the Lanczos algorithm by applying singular value decomposition(SVD) to compress Lanczos vectors, for the purpose of reducing memory cost. An algorithm to compress Lanczos vectors by SVD has been derived by M. Weinstein et al [6], to diagonalize lattice Hamiltonians. While their approach to apply SVD on lattice problems is to partition the lattice into two subclusters, we present a new approach to apply this compression algorithm to quantum field theoretic calculations.

Provided with a quantum field theory and DLCQ, we first construct the basis set with a specific cut-off and store the basis states into sectors[1] of a 2D matrix, in which one dimension stands for longitudinal momenta and the other represents transverse momenta. With the basis states stored in 2D matrices, Lanczos vectors become sectors of 2D matrices, instead of 1D vectors. With this adjustment, the application of SVD on the lanczos vectors is enabled. Detailed techniques are given in section 2.4 and section 3.1.

For the cut-off, we choose uncorrelated limitations for the two dimensions, instead of taking a limitation that has coupling between longitudinal and transverse momenta. For a given cut-off limitation, with M longitudinal states and N transverse states in a sector, the matrix size is $M \times N$. We can apply SVD to compress each sector of the Lanczos vectors into a set of n_{svd} vectors of length M and n_{svd} vectors of length N. As long as $n_{svd} \ll min(M, N)$, one can greatly economize on memory.

The thesis is organized as follows:

- In Chapter 2, we begin by introducing Light-cone Coordinates, light-cone point

[1] Generally each sector contains states with the same number of particles. In this thesis, states with the same number of physical bosons and same number of Pauli-Villars are in the same sector.

particle, and proceed to present DLCQ and a soluble model [7], which has been solved by Brodsky and co-workers using the Lanczos Algorithm. This model will serve as a testing model for our compression algorithm.

- In Chapter 3, the compression algorithm, which enables numerical diagonalization with reduced memory requirements, is outlined. SVD and its applications are briefly introduced first, followed by SVD applied to light-cone wave functions and to the application of the Hamiltonian to Lanczos vectors. Then we introduce the Lanczos algorithm and describe necessary modifications of the Lanczos algorithm for our calculations.

- Chapter 4 demonstrates the implementation of the compression algorithm on the soluble model presented in Chapter 2. Two cutoff limitations and their corresponding basis sizes are given first, followed by numerical techniques used when applying the compression algorithm. The calculated results, as well as comparisons with analytic solutions are then presented, followed by the study of compression ratio and the convergence with respect to the compression parameter, n_{svd}.

- Chapter 5 presents a final discussion of the algorithm and summarizes what we have learned from this study. Some possible ideas to modify the compression algorithm as well as some possible extensions of the work are also included at the end of this chapter.

Chapter 2

Background

2.1 Light-cone Coordinates

In special relativity, light-cone coordinates [8] form a special coordinate system where two of the coordinates, x^+ and x^- are null coordinates and all the other coordinates are spatial. Consider the light-cone coordinates defined by

$$x^{\pm} \equiv x^0 \pm x^3, \tag{2.1}$$

where x^0 is the time coordinate and x^3 is a chosen spatial coordinate. The other two spatial coordinates x^1 and x^2 are kept as transverse coordinates $\mathbf{x}_\perp$. The complete set of light-cone coordinates become (x^+, x^-, x^1, x^2) and the x^+ is to be thought of as a new time coordinate.

In any inertial system, there is an invariant

$$I \equiv (x^0)^2 - (x^1)^2 - (x^2)^2 - (x^3)^2 = \frac{1}{2}(x^+x^- + x^-x^+) - (x^1)^2 - (x^2)^2. \tag{2.2}$$

The invariant I can be written as a double sum:

$$I = \sum_{\mu=0}^{3}\sum_{\nu=0}^{3} g_{\mu\nu}x^\mu x^\nu = g_{\mu\nu}x^\mu x^\nu, \tag{2.3}$$

where $g_{\mu\nu}$ is the spacetime metric, whose components can be displayed as a matrix. In

light-cone coordinates, the metric is

$$g_{\mu\nu} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}. \tag{2.4}$$

To simplify the expression for invariant I, $g_{\mu\nu}$ is used to define the lower light-cone indices,

$$x_\mu \equiv g_{\mu\nu} x^\nu. \tag{2.5}$$

The invariant becomes

$$I = x_+ x^+ + x_- x^- + x_1 x^1 + x_2 x^2. \tag{2.6}$$

The light-cone components of any Lorentz vector a^μ are defined in analogy with Equation (2.1) :

$$a^\pm \equiv (a^0 \pm a^3). \tag{2.7}$$

The scalar product between vectors, can be written using light-cone components and lower light-cone indices (2.5),

$$\begin{aligned} \mathbf{a}\cdot\mathbf{b} = g_{\mu\nu} a^\mu b^\nu = g_{\nu\mu} a^\mu b^\nu = a_\nu b^\nu = a_+ b^+ + a_- b^- + a_1 b^1 + a_2 b^2, \\ \mathbf{a}\cdot\mathbf{a} = a_+ a^+ + a_- a^- + a_1 a^1 + a_2 a^2. \end{aligned} \tag{2.8}$$

The light-cone components p^+ and p^- of the momentum Lorentz vector are obtained using the rule (2.7):

$$p^\pm = (p^0 \pm p^3) = \frac{1}{2} p_\mp. \tag{2.9}$$

In standard coordinates, we have

$$\mathbf{p}\cdot\mathbf{x} = p_0 x^0 + p_1 x^1 + p_2 x^2 + p_3 x^3. \tag{2.10}$$

In light-cone coordinates, using (2.8),

$$\mathbf{p}\cdot\mathbf{x} = p_+ x^+ + p_- x^- + p_1 x^1 + p_2 x^2. \tag{2.11}$$

In standard coordinates, $p_0 = -E/c$ appears together with the time x^0. In light-cone coordinates, we have x^+ as our light-cone time and $p_+ = \frac{1}{2} p^-$ appears together with x^+.

We would therefore expect p^-, p^+ and $\mathbf{p}_\perp$ be the light-cone energy, longitudinal momentum and transverse momentum respectively. Here we use the Schrödinger equation as an illustration. The wavefunction of a point particle with energy E and momentum $\mathbf{p}$ is given by

$$\psi(t,\vec{x}) = \exp\left(-\frac{i}{\hbar}(Et - \vec{p}\cdot\vec{x})\right) = \exp\left(\frac{i}{\hbar}(p_0x^0 + \vec{p}\cdot\vec{x})\right) = \exp\left(\frac{i}{\hbar}\mathbf{p}\cdot\mathbf{x}\right). \quad (2.12)$$

Using (2.1), we have

$$\psi(\mathbf{x}) = \exp\left(\frac{i}{\hbar}(p_+x^+ + p_-x^- + p_1x^1 + p_2x^2)\right). \quad (2.13)$$

In standard coordinates, the wavefunction satisfies the Schrödinger equation

$$i\hbar\frac{\partial\psi}{\partial x^0} = \frac{E}{c}\psi. \quad (2.14)$$

Similarly, light-cone time and light-cone energy E_{lc} should be related by

$$i\hbar\frac{\partial\psi}{\partial x^+} = \frac{E_{lc}}{c}\psi. \quad (2.15)$$

We can return to (2.13) and evaluate:

$$i\hbar\frac{\partial\psi}{\partial x^+} = -p_+\psi \longrightarrow p_+ = -\frac{E_{lc}}{c}. \quad (2.16)$$

This confirms our identification of p_+ as light-cone energy. Since $p_+ = \frac{1}{2}p^-$, it is equivalent to use p^- as the light-cone energy,

$$p^- = -\frac{2E_{lc}}{c}. \quad (2.17)$$

As for the other three coordinates, p^+ is longitudinal light-cone momentum while p^1 and p^2 are two transverse ones.

2.2 Light-cone point particle

We have discussed the energy and momentum in light-cone coordinate, and now we come to the light-cone point particle. The action for a relativistic point particle is given by the arc length of its trajectory

$$S \equiv -m\int_1^2 ds = -m\int_{\tau_1}^{\tau_2}\sqrt{g_{\mu\nu}\dot{x}^\mu\dot{x}_\nu}d\tau = -m\int_{\tau_1}^{\tau_2}\sqrt{\dot{x}_\mu\dot{x}^\mu}. \quad (2.18)$$

The action S corresponds to a Lagrangian L as

$$S = \int_{\tau_1}^{\tau_2} L d\tau, \quad L = -m\sqrt{\dot{x}_\mu \dot{x}^\mu}. \tag{2.19}$$

By differentiating the Lagrangian with respect to the velocity, the momentum is obtained as

$$p^\mu = -\frac{\partial L}{\partial \dot{x}_\mu} = \frac{m\dot{x}^\mu}{\sqrt{\dot{\mathbf{x}}^2}}. \tag{2.20}$$

Consider the + component of the above equation:

$$p^+ = \frac{m}{\sqrt{\dot{\mathbf{x}}^2}}\dot{x}^+ = \frac{1}{\sqrt{\dot{\mathbf{x}}^2}}\frac{p^+}{m}, \tag{2.21}$$

we immediately figure out that,

$$\dot{\mathbf{x}}^2 = \frac{1}{m^2}. \tag{2.22}$$

This result helps us to simplify the expression (2.20) for the momentum:

$$p_\mu = m^2 \dot{x}_\mu. \tag{2.23}$$

Using(2.23) we can rewrite (2.22) as

$$\mathbf{p}^2 = m^2. \tag{2.24}$$

Expanding in light-cone components,

$$p^+p^- - (p^1)^2 - (p^2)^2 = m^2 \longrightarrow p^- = \frac{m^2 + \mathbf{p}_\perp^2}{p^+}. \tag{2.25}$$

Having solved for p^-, equation (2.23) gives

$$\frac{dx^\mu}{d\tau} = \frac{1}{m^2}p^\mu, \tag{2.26}$$

which is integrated to find

$$x^\mu(\tau) = x_0^\mu + \frac{p^\mu}{m^2}\tau, \tag{2.27}$$

where x_0^μ is a constant determined by initial conditions, except that the light-cone gauge condition

$$x^+ = \frac{1}{m^2}p^+\tau \tag{2.28}$$

implies that $x^+(\tau)$ has no constant piece x_0^+.

The specification of the motion of the point particle is now complete. Equation (2.25) indicates that the energy of a particle is determined by $\mathbf{p}_\perp$ and p^+. From Equation (2.27), we can see that once we fix the value of x_0^- and $\mathbf{x}_\perp$, the motion of the particle is also determined if we presume to know $\mathbf{p}_\perp$ and p^+ . Our independent dynamical variables for the point particle are therefore $\left(\mathbf{x}_\perp, x_0^-, \mathbf{p}_\perp, p^+\right)$. We will use an underline to indicate the three components of position $\underline{x} = (x_0^-, \mathbf{x}_\perp)$ and momentum $\underline{p} = (p^+, \mathbf{p}_\perp)$.

2.3 Discretized Light-Cone Quantization

Discretized Light-Cone Quantization(DLCQ) is a suggested computational method for solving quantum field theories for their mass spectrum and wave functions [1–3]. This method has been applied successfully to various theories including Yukawa theroy and QED, and proposed as a method for solving QCD. In each of these applications, the mass spectrum and wave functions are successfully obtained, and results agree with previous analytical and numerical calculations. Here we present a brief introduction to this method.

We first specify quantization conditions on the characteristic surface $x^+ \equiv (ct + z) = 0$, and then introduce periodic conditions to induce a discrete basis. Consider a light-cone box

$$-L < x^- < L, -L_\perp < x, y < L_\perp. \tag{2.29}$$

By imposing periodic boundary conditions for bosons,

$$\phi(x^- = -L) = \phi(x^- = L), \tag{2.30}$$

and antiperiodic boundary conditions for fermions,

$$\psi(x^- = -L) = -\psi(x^- = L), \tag{2.31}$$

one obtains discrete momenta

$$p^+ \to \frac{\pi}{L} n, \ \mathbf{p}_\perp \to (\frac{\pi}{L_\perp} n_x, \frac{\pi}{L_\perp} n_y), \tag{2.32}$$

with n even for bosons and odd for fermions. Integrals can then be expressed as discrete sums obtained by trapezoidal approximations on the grid of momentum values,

$$\int dp^+ \int d^2p_\perp f(p^+, \mathbf{p}_\perp) \simeq \frac{2\pi}{L}\left(\frac{\pi}{L_\perp}\right)^2 \sum_n \sum_{n_x,n_y=-N_\perp}^{N_\perp} f(n\pi/L, \mathbf{n}_\perp\pi/L_\perp). \tag{2.33}$$

The Limit $L \to \infty$ can be exchanged for a limit in terms of the integer resolution $K \equiv (L/\pi)P^+$. The integers n_x and n_y range between limits associated with some maximum integer $N_\perp$, which is fixed by $L_\perp$ and a cutoff that limits transverse momentum. The momentum-space continuum limit is reached when K and $N_\perp$ become infinite. The transverse length scale $L_\perp$ is chosen such that $N_\perp\pi/L_\perp$ is the largest transverse momentum allowed by the cutoff.

In quantum field theory, the eigenvector of a Hamiltonian is an expansion in multiparticle occupation Fock states. Suppose a Fock state has n bosons and one fermion with momenta $\underline{q}_i$ and $\underline{p}$ respectively. Here i ranges from 1 to n. This is then an eigenstate of particle number and total momentum

$$\underline{P} = \sum_i \underline{q}_i + \underline{p}. \tag{2.34}$$

In the DLCQ approximation, the momenta are represented by integers

$$\underline{q}_i = (\frac{\pi}{L}m_i, \frac{\pi}{L_\perp}\mathbf{m}_\perp) \;\text{ and }\; \underline{p} = (\frac{\pi}{L}n, \frac{\pi}{L_\perp}\mathbf{n}_\perp), \tag{2.35}$$

with

$$\underline{m}_i = (m_i, \mathbf{m}_{i\perp}) \;\text{ and }\; \underline{n} = (n, \mathbf{n}_\perp). \tag{2.36}$$

We will work in a frame where the total transverse momentum is zero. Therefore we have

$$K = \sum_i m_i + n,\; 0 = \sum_i m_{ix} + n_x,\; \text{and } 0 = \sum_i m_{iy} + n_y. \tag{2.37}$$

2.4 A Soluble Model

DLCQ works well for solving quantum field problems, but it also has a renormalization problem caused by removing an infinite set of high energy states. Brodsky and

co-workers proposed a solution to this problem by adding enough Pauli-Villars fields to the theory in order to perform a consistent renormalization of the operators [7]. They also illustrated this procedure by presenting and solving a model field theory, which is similar to the scalar field model studied by Greenberg and Schweber [9]. We will introduce their soluble model in this section. It will serve as the testing model for our compression algorithm, which is to be discussed in the following chapters.

The effective light-cone Hamiltonian given by Brodsky and his co-workers [7] is

$$\begin{aligned} H_{LC}^{eff} = & \int \frac{dp^+ d^2p_\perp}{16\pi^3 p^+}(M_0^2 + M_0' p^+) \sum_\sigma b_{\underline{p}\sigma}^\dagger b_{\underline{p}\sigma} \\ & + P^+ \int \frac{dq^+ d^2q_\perp}{16\pi^3 q^+}\left[\frac{\mu^2 + q_\perp^2}{q_\perp} a_{\underline{q}}^\dagger a_{\underline{q}} + \frac{\mu_1^2 + q_\perp^2}{q_\perp} a_{1\underline{q}}^\dagger a_{1\underline{q}}\right] \\ & + g \int \frac{dp_1^+ d^2p_{\perp 1}}{\sqrt{16\pi^3 p_1^+}} \int \frac{dp_2^+ d^2p_{\perp 2}}{\sqrt{16\pi^3 p_2^+}} \int \frac{dq^+ d^2q_\perp}{16\pi^3 q^+} \sum_\sigma b_{\underline{p}1\sigma}^\dagger b_{\underline{p}2\sigma}\Bigg[\left(\frac{p_1^+}{p_2^+}\right)^\gamma a_{\underline{q}}^\dagger \delta(\underline{p}_1 - \underline{p}_2 + \underline{q}) \\ & + \left(\frac{p_2^+}{p_1^+}\right)^\gamma a_{\underline{q}} \delta(\underline{p}_1 - \underline{p}_2 - \underline{q}) + i\left(\frac{p_1^+}{p_2^+}\right)^\gamma a_{\underline{q}}^\dagger \delta(\underline{p}_1 - \underline{p}_2 + \underline{q}) + i\left(\frac{p_2^+}{p_1^+}\right)^\gamma a_{\underline{q}} \delta(\underline{p}_1 - \underline{p}_2 - \underline{q})\Bigg], \end{aligned} \tag{2.38}$$

with $\underline{p} \equiv (p^+, \mathbf{p}_\perp)$ and

$$\begin{aligned} \left[a_{\underline{q}}, a_{\underline{q}'}^\dagger\right] &= 16\pi^3 q^+ \delta(\underline{q} - \underline{q}'), \\ \{b_{\underline{p}\sigma}, b_{\underline{p}'\sigma'}^\dagger\} &= 16\pi^3 p^+ \delta(\underline{p} - \underline{p}')\delta_{\sigma\sigma'}. \end{aligned} \tag{2.39}$$

The operator $a_{\underline{q}}^\dagger$ creates a boson with momentum $\underline{q}$ and $b_{\underline{p}\sigma}^\dagger$ a fermion with momentum $\underline{p}$ and spin σ.The first integral is the fermion kinetic energy, where M_0 is the free invariant mass of a fermion and M_0' is the parameter that sets the strength of what is called a counter-term[1] . The second integral is the boson kinetic energy, which has two parts: one for physical bosons and the other one Pauli-Villars(PV), in which μ and μ_1 are the mass for physical bosons and PV bosons respectively. The third term in the Hamiltonian describes the interaction between the fermion and the boson field, in which the first two terms depict emission and absorption of a physical boson respectively, while the other two terms are emission and absorption of a PV boson similarly. The factor

[1] Such terms are added to Hamiltonians in quantum field theory to subtract infinities that appear and to make the resulting theory better defined.

g in this term is the (real) coupling constant [2] for bosons and γ is a parameter in the interaction[3] . As has been indicated, a PV field is included in this Hamiltonian serving as the necessary counter-terms to preserve the symmetries of the theory and to cancel the infinities.

This model is a many body problem that involves emission and absorption of bosons, the states of this system are typically a superposition of an infinite number of 'bare' states. The Fock-state expansion of an eigenvector is as follows:

$$\begin{aligned}\Phi_\sigma =&\sqrt{16\pi^3 P^+}\sum_{n,n1}\int\frac{dp^+d^2p_\perp}{\sqrt{16\pi^3p^+}}\prod_{i=1}^{n}\int\frac{dq_i^+d^2q_{\perp i}}{\sqrt{16\pi^3q_i^+}}\prod_{j=1}^{n_1}\int\frac{dr_j^+d^2r_{\perp j}}{\sqrt{16\pi^3r_j^+}}\delta\Big(\underline{P}-\underline{p}-\sum_i^n\underline{q}_i-\sum_j^{n_1}\underline{r}_j\Big)\\ &\times\phi^{(n,n_1)}(\underline{q}_i,\underline{r}_j;\underline{p})\frac{1}{\sqrt{n!n_1!}}b^\dagger_{\underline{p}\sigma}\prod_i^n a^\dagger_{\underline{q}_i}\prod_j^{n_1}a^\dagger_{1\underline{r}_j}|0\rangle.\end{aligned} \tag{2.40}$$

The normalization condition for this state is

$$\Phi'^\dagger_\sigma\cdot\Phi_\sigma = 16\pi^3P^+\delta(\underline{P}'-\underline{P}), \tag{2.41}$$

which yields the following condition on the individual amplitudes:

$$1=\sum_{n,n1}\prod_i^n\int dq_i^+d^2q_{\perp i}\prod_j^n\int dr_j^+d^2r_{\perp j}\left|\phi^{(n,n_1)}\left(\underline{q}_i,\underline{r}_j;\underline{P}-\sum_i\underline{q}_i-\sum_j\underline{r}_j\right)\right|^2. \tag{2.42}$$

Brodsky and co-workers solved the equation

$$H^{eff}_{LC}\Phi_\sigma = M^2\Phi_\sigma \tag{2.43}$$

both analytically and numerically. With $y_i = q_i^+/P^+$ and $z_j = r_j^+/P^+$, the coupled

[2] For quantum electrodynamics, the coupling constant is just e, the electron charge.

[3] An analogous situation in ordinary mechanics would be a potential of the form $\frac{g}{r^\gamma}$ instead of just $\frac{g}{r}$. Here, $\gamma = 1/2$ is the natural choice that matches what the Yukawa model would yield for an infinitely massive fermion.

equation of (2.43) is given as

$$\Big[M^2 - M_0^2 - M_0' p^+ - \sum_i \frac{\mu^2 + q_{\perp i}^2}{y_i} - \sum_j \frac{\mu_1^2 + r_{\perp j}^2}{z_j}\Big]\phi^{(n,n_1)}(\underline{q}_i, \underline{r}_j, \underline{p})$$
$$= g\Big\{\sqrt{n+1}\int \frac{dq^+ d^2q_\perp}{\sqrt{16\pi^3 q^+}}\Big(\frac{p^+ - q^+}{p^+}\Big)^\gamma \phi^{(n+1,n_1)}(\underline{q}_i, \underline{q}, \underline{r}_j, \underline{p} - \underline{q})$$
$$+ \frac{1}{\sqrt{n}}\sum_i \frac{1}{\sqrt{16\pi^3 q_i^+}}\Big(\frac{p^+}{p^+ - q_i^+}\Big)^\gamma \phi^{(n-1,n_1)}(\underline{q}_1, ...\underline{q}_{i-1}, \underline{q}_{i+1}, ..., \underline{q}_n, \underline{r}_j, \underline{p} + \underline{q}_i)$$
$$+ i\sqrt{n_1+1}\int \frac{dr^+ d^2r_\perp}{\sqrt{16\pi^3 r_+}}\Big(\frac{p^+ - r^+}{r^+}\Big)^\gamma \phi^{(n,n_1+1)}(\underline{q}_i, \underline{r}_j, \underline{r}, \underline{p} - \underline{r})$$
$$+ \frac{i}{\sqrt{n_1}}\sum_j \frac{1}{\sqrt{16\pi^3 r_j^+}}\Big(\frac{p^+}{p^+ - r_j^+}\Big)^\gamma \phi^{(n,n_1-1)}(\underline{q}_i, \underline{r}_1, ...\underline{r}_{j-1}, \underline{r}_{j+1}, ..., \underline{r}_{n_1}, \underline{p} + \underline{r}_j).$$

Its analytic solution is given as

$$\phi^{(n,n_1)} = \sqrt{Z}\frac{(-g)^n(-ig)^{n_1}}{\sqrt{n!n_1!}}\Big(\frac{p^+}{P^+}\Big)^\gamma \prod_i \frac{yi}{\sqrt{16\pi^3 q_i^+}(\mu^2 + q_{\perp i}^2)}\prod_j \frac{z_j}{\sqrt{16\pi^3 r_j^+}(\mu_1^2 + r_{\perp i}^2)}, \tag{2.44}$$

provided that $M_0 = M$ and

$$M_0' = \frac{g^2/P^+}{16\pi^2}\frac{\mu_1/\mu}{\gamma + 1/2}. \tag{2.45}$$

This soluble model is indeed a good testing model for our compression algorithm, since we can compare our numerical results with this analytic solution, and also compare with the calculations by Brodsky and co-workers.

Before we discuss our compression algorithm in the next chapter, we will first apply DLCQ to this soluble model. As indicated in (2.33) , integrals can be replaced by discrete sums as trapezoidal approximations on the grid of momentum values. Substitute (2.33) into equation (2.42), and the normalization condition becomes

$$1 = \sum_{n,n1}\prod_{i=1}^{n}\sum_{\underline{m}_i}\prod_{j=1}^{n_1}\sum_{\underline{l}_j}\big|\tilde{\phi}^{(n,n_1)}\big(\underline{m}_i, \underline{l}_j; \underline{K} - \sum_i \underline{m}_i - \sum_j \underline{l}_j\big)\big|^2, \tag{2.46}$$

where

$$\tilde{\phi}^{(n,n_1)} = \big[\frac{2\pi}{L}\big(\frac{\pi}{L\perp}\big)^2\big]^{(n+n_1)/2}\phi^{(n,n_1)} \tag{2.47}$$

and $\underline{m}_i$, $\underline{l}_j$ and $\underline{K}$ denote the light-cone momentum integers for physical bosons, PV bosons and total momentum integers respectively.

The summation in (2.47) includes states that differ by only rearrangement of bosons with the same momenta. To eliminate this effect, it is convenient to introduce the number basis here. We introduce factorials $N_{\underline{m}_i} \equiv N_{\underline{m}_1}! N_{\underline{m}_2}! \cdots$, where $N_{\underline{m}_1}$ is the number of times that $\underline{m}_1$ appears in the collection $\underline{m}_i$. We also define collections of sums with a prime $\prod_{i=1}^{n} \sum'_{\underline{m}_i}$ as being restricted to one ordering of the momenta. The amplitudes for this number basis are related to $\tilde{\phi}^{(n,n_1)}$ by

$$\psi^{(n,n_1)} = \sqrt{\frac{n! n_1!}{N_{\underline{m}_i} N_{\underline{l}_j}}} \tilde{\phi}^{(n,n_1)}, \tag{2.48}$$

with normalization

$$1 = \sum_{n,n1} \prod_{i=1}^{n} \sum_{\underline{m}_i}{}' \prod_{j=1}^{n_1} \sum_{\underline{l}_j}{}' |\psi^{(n,n_1)}|^2. \tag{2.49}$$

In this basis the discretization of the coupled equations (2.44) yields

$$\begin{aligned}
&\left[\tilde{M}^2 - \tilde{M}_0^2 - \tilde{M}_0' \frac{n}{K} - \sum_i \frac{1 + (m_{ix}^2 + m_{iy}^2)/\tilde{L}_\perp^2}{m_i/K} - \sum_j \frac{\tilde{\mu}_1^2 + (l_{jx}^2 + l_{jy}^2)/\tilde{L}_\perp^2}{l_j/K}\right] \psi^{(n,n_1)}(\underline{m}_i, \underline{l}_j, \underline{n}) \\
&= \frac{g/\mu}{\tilde{L}_\perp \sqrt{8\pi^3}} \Bigg\{ \sum_{\underline{m}} \frac{1}{\sqrt{m}} \sqrt{\frac{N_{\{\underline{m}, \underline{m}_i\}}}{N_{\{\underline{m}_i\}}}} \left(\frac{n-m}{n}\right)^{\gamma} \psi^{(n+1,n_1)}(\underline{m}_i, \underline{m}, \underline{l}_j, \underline{n} - \underline{m}) \\
&+ \sum_i \frac{1}{\sqrt{m_i}} \sqrt{\frac{N_{\{\underline{m}_i\}'}}{N_{\{\underline{m}_i\}}}} \left(\frac{n}{n+m_i}\right)^{\gamma} \psi^{(n-1,n_1)}(\underline{m}_1, ...\underline{m}_{i-1}, \underline{m}_{i+1}, ..., \underline{m}_n, \underline{l}_j, \underline{n} + \underline{m}_i) \\
&+ i \sum_{\underline{l}} \frac{1}{\sqrt{l}} \sqrt{\frac{N_{\{\underline{l}, \underline{l}_j\}}}{N_{\{\underline{l}_j\}}}} \left(\frac{n-l}{n}\right)^{\gamma} \psi^{(n,n_1+1)}(\underline{m}_i, \underline{l}_j, \underline{l}, \underline{n} - \underline{l}) \\
&+ i \sum_j \frac{1}{\sqrt{l_j}} \sqrt{\frac{N_{\{\underline{l}_j\}'}}{N_{\{\underline{l}_j\}}}} \left(\frac{n}{n+l_j}\right)^{\gamma} \psi^{(n,n_1-1)}(\underline{m}_i, \underline{l}_1, ...\underline{l}_{j-1}, \underline{l}_{j+1}, ..., \underline{l}_{n_1}, \underline{n} + \underline{l}_j) \Bigg\},
\end{aligned} \tag{2.50}$$

where $\underline{n} = \underline{K} - \sum_i \underline{m}_i - \sum_j \underline{l}_j$, $\{\underline{m}_i\}'$ is the set of boson momenta without $\underline{m}_i$, and a tilde implies division by μ except for $\tilde{L}_\perp = \mu L_\perp / \pi$.

This is a matrix eigenvalue problem, which for given g, μ, M_0 and $\tilde{M}_0'$, we solve for ψ and M. This model has been solved numerically by Brodsky and co-workers using the

Lanczos algorithm. However the Lanczos vectors in their calculations cost too much memory, which limited the resolution of the calculations. In the next chapter, we will introduce a compression algorithm which reduces the memory cost significantly. The idea is to apply singular value decomposition (SVD) to the eigenvectors and store the Lanczos vectors in SVD form.

Before introducing the algorithm, we will first reconstruct the basis for equation (2.50) for later use. We propose to reshape the original 1D Lanczos vectors into sectors of 2D matrices, in which one dimension stands for longitudinal momenta and the other for transverse momenta. We will also factorize each term of the Hamiltonian into a longitudinal part and a transverse part, so that the factorized Hamiltonians can be applied to longitudinal eigenvectors and transverse eigenvectors separately.

Notice that the factorial $N_{\{\underline{m}_i\}}$ in Equation (2.50) involves both longitudinal and transverse momenta and is impossible to be factorized. So we define

$$\tilde{\psi}^{(n,n_1)} \equiv \sqrt{N_{\{\underline{m}_i\}} N_{\{\underline{l}_j\}}}\psi^{(n,n_1)} = \sqrt{n!n_1!}\tilde{\phi}^{(n,n_1)}. \tag{2.51}$$

Equation (2.50) then becomes

$$\begin{aligned}
&\left[\tilde{M}^2 - \tilde{M}_0^2 - \tilde{M}_0' \frac{n}{K} - \sum_i \frac{1 + (m_{ix}^2 + m_{iy}^2)/\tilde{L}_\perp^2}{m_i/K} - \sum_j \frac{\tilde{\mu}_1^2 + (l_{jx}^2 + l_{jy}^2)/\tilde{L}_\perp^2}{l_j/K}\right]\tilde{\psi}^{(n,n_1)}(\underline{m}_i, \underline{l}_j, \underline{n}) \\
&= \frac{g/\mu}{\tilde{L}_\perp\sqrt{8\pi^3}}\Bigg\{\sum_{\underline{m}} \frac{1}{\sqrt{m}}\left(\frac{n-m}{n}\right)^\gamma \tilde{\psi}^{(n+1,n_1)}(\underline{m}_i, \underline{m}, \underline{l}_j, \underline{n} - \underline{m}) \\
&+ \sum_i \frac{1}{\sqrt{m_i}}\left(\frac{n}{n+m_i}\right)^\gamma \tilde{\psi}^{(n-1,n_1)}(\underline{m}_1, ...\underline{m}_{i-1}, \underline{m}_{i+1}, ..., \underline{m}_n, \underline{l}_j, \underline{n} + \underline{m}_i) \\
&+ i\sum_{\underline{l}} \frac{1}{\sqrt{l}}\left(\frac{n-l}{n}\right)^\gamma \tilde{\psi}^{(n,n_1+1)}(\underline{m}_i, \underline{l}_j, \underline{l}, \underline{n} - \underline{l}) \\
&+ i\sum_j \frac{1}{\sqrt{l_j}}\left(\frac{n}{n+l_j}\right)^\gamma \tilde{\psi}^{(n,n_1-1)}(\underline{m}_i, \underline{l}_1, ...\underline{l}_{j-1}, \underline{l}_{j+1}, ..., \underline{l}_{n_1}, \underline{n} + \underline{l}_j)\Bigg\},
\end{aligned} \tag{2.52}$$

which does not include $N_{\{\underline{m}_i\}}$ factors. However, the normalization contains extra factors,

because (2.49) becomes

$$1 = \sum_{n,n1} \prod_{i=1}^{n} \sum_{\underline{m}_i}{}' \prod_{j=1}^{n_1} \sum_{\underline{l}_j}{}' \frac{|\tilde{\psi}^{(n,n_1)}|^2}{N_{\{\underline{m}_i\}} N_{\{\underline{l}_j\}}}. \tag{2.53}$$

We have finished discussing the soluble model at this point. We will present the compression algorithm in the next chapter, in which we will refer back to the soluble model when necessary.

Chapter 3

A Compression Algorithm

Our compression algorithm has two parts: the Singular Value Decompostion (SVD) [10, 11] step and the Lanczos algorithm. Basically, we will continue to use the Lanczos algorithm to solve the eigenvalue problem (2.50). However, we propose to apply SVD to compress the Lanczos vectors in order to significantly reduce the memory cost in the computation. This chapter has two sections, one for the SVD step and the other for the Lanczos algorithm.

3.1 The Singular Value Decomposition Step

In this section, we will give a brief introduction to SVD first [10,11], and then present the way in which SVD is applied to the light-cone wave functions to compress the Lanczos vector. We will also present the SVD projection applied to $H|\psi\rangle$, i.e. the Hamiltonian acting on light-cone wave function, to compress the resulting Lanczos vector.

3.1.1 Introduction to SVD

Given a $m \times n$ real or complex matrix A, a non-negative real number σ is a singular value if and only if there exist unit-length vectors u in $\mathbb{C}^m$ and v in $\mathbb{C}^n$ such that

$$A\vec{v} = \sigma\vec{u} \quad \text{and} \quad A^\dagger\vec{u} = \sigma\vec{v}, \tag{3.1}$$

where $A^\dagger$ is the conjugate transpose of A, and the vectors u and v are called left-singular and right-singular vectors for σ, respectively.

The matrix product $U\Sigma V^\dagger$ is a singular value decomposition for a given matrix A if

- U and V have orthonormal columns respectively.
- Σ has nonnegative elements on its diagonal and zeros elsewhere.
- $A = U\Sigma V^\dagger$.

The diagonal entries of Σ are equal to the singular values of A. The columns of U and V are, respectively, left- and right-singular vectors for the corresponding singular values. If the rank of A is r, then we have a reduced form of SVD, which is $A_{m\times n} = U_{m\times r}\Sigma_{r\times r}V^\dagger_{r\times n}$, where only the r positive singular values are stored in Σ and their corresponding singular vectors are stored in U and V.

The SVD is an important tool in several different applications. One of the most direct applications of the SVD is to the problem of computing the eigenvalue decomposition of a matrix product $A^\dagger A$. Since the SVD can be computed by operating directly on the original matrix A, one can obtain the desired eigenvectors of $A^\dagger A$ and eigenvalues of $A^\dagger A$ without ever explicitly computing $A^\dagger A$. Another application of the SVD is as a numerically reliable estimate of the effective rank of a matrix. Often linear dependencies in data are masked by measurement error, and if there are r_e singular values that are larger than the measurement error, the effective rank of the matrix is found to be r_e. In this thesis, we will apply SVD to wave-function compression with reduced rank approximations. We will illustrate this in the next two subsections.

3.1.2 SVD applied to light-cone wave function

In quantum mechanics, the wave function $|\psi\rangle$ is usually stored as a one-dimensional vector, in which each element of the vector gives the amplitude for a particular quantum state of a particle or a system. Since each quantum state of the system described in equation (2.50) has specified transverse-momentum and longitudinal-momentum integers for particles in that state, it is equivalent to use a matrix to represent a wave function. The idea is to reshape the wave function in a 2D matrix form, in which one dimension is assigned for listings of transverse-momenta integers , and the other

for listings of longitudinal-momenta integers. Consequently, each element of the matrix describes a quantum state with corresponding transverse-momenta integers and longitudinal-momenta integers.

Having done that we expect to naturally apply SVD to the wave function without losing information of any quantum states. Any state $|\tilde{\psi}\rangle$ with n physical bosons, n_1 Pauli-Villars can be represented in a unique SVD form [6] as

$$|\tilde{\psi}\rangle_{\mu\nu} = \sum_j \lambda_j \phi_{\mu j} \phi_{\nu j}, \tag{3.2}$$

where μ and ν are indices for transverse momenta and longitudinal momenta respectively, the λ_j are positive, and $\phi_{\mu j}$ and $\phi_{\nu j}$ are small basis vectors of transverse-momentum space and longitudinal-momentum space respectively. The λ_j are normalized such that

$$\sum_j \lambda_j^2 = 1, \tag{3.3}$$

and the basis vectors are orthonormal,

$$\langle \phi_{\mu j} | \phi_{\mu j'} \rangle = \delta_{jj'} \text{ and } \langle \phi_{\nu j} | \phi_{\nu j'} \rangle = \delta_{jj'}. \tag{3.4}$$

To compress the wave function, we only keep the n_{svd} largest terms of the summation, which defines the SVD projector,

$$|\tilde{\psi}\rangle_{\mu\nu} \simeq P_{svd} |\tilde{\psi}\rangle_{\mu\nu} = \sum_{j=1}^{n_{svd}} \lambda_j \phi_{\mu j} \phi_{\nu j}. \tag{3.5}$$

Here n_{svd} is a chosen number so that for any $j \geq n_{svd}$, $\lambda_j \simeq 0$. Note that the wave function in Equation (2.50) has many sectors, each sector has a different n and a different n_1. The SVD projector is applied to all sectors except the starting sectors and ending sectors, where $n + n_1$ are very small or very large and the rank of matrices are already very small. For the application of the SVD projector to different sectors, we could have a different value for n_{svd} in each sector; however, we will assign the same n_{svd} for all sectors, for computational convenience.

3.1.3 SVD applied to $H|\psi\rangle$

Our compression algorithm economizes on the storage space by applying an SVD projection after each application of the Hamiltonian on the Lanczos vector,

$$\tilde{\psi}'_{\mu\nu} = P_{svd} H \tilde{\psi}_{\mu\nu}. \tag{3.6}$$

Since sectors of $|\tilde{\psi}\rangle$ are stored in SVD form, we need to factorize the Hamiltonian in order to apply it on $|\tilde{\psi}\rangle$. The Hamiltonian is factorized into a sum of products of a transverse operator and a longitudinal operator,

$$H = H^0_\mu \bigotimes I_\nu + I_\mu \bigotimes H^0_\nu + \sum_{\imath=3}^{M} H^i_\mu \bigotimes H^i_\nu. \tag{3.7}$$

where H^0_μ denotes transverse-momenta-dependent operators while H^0_ν denotes longitudinal-momenta-dependent operators. $H^i_\mu \bigotimes H^i_\nu$ is a factorization of operators dependent on both transverse and longitudinal momenta. We will take operators in (2.50) as an example to demonstrate the factorizations.

The factorization of operators in the left side of Equation (2.50) , which involves no interaction between different sectors, is demonstrated as follows:

$$\begin{aligned}
\tilde{M}'_0 \frac{n}{K} &\longrightarrow I_\mu \bigotimes diag \tilde{M}'_0 \frac{n}{K} \\
\sum_i \frac{1 + (m^2_{ix} + m^2_{iy})/\tilde{L}^2_\perp}{m_i/K} &\longrightarrow diag \sum_i (1 + (m^2_{ix} + m^2_{iy})/\tilde{L}^2_\perp) \bigotimes diag \sum_i (K/m_i) \\
\sum_j \frac{\tilde{\mu}^2_1 + (l^2_{jx} + l^2_{jy})/\tilde{L}^2_\perp}{l_j/K} &\longrightarrow diag \sum_j (\tilde{\mu}^2_1 + (l^2_{jx} + l^2_{jy})/\tilde{L}^2_\perp) \bigotimes diag \sum_j (K/l_j).
\end{aligned} \tag{3.8}$$

As we can see, for each sector of $\tilde{\psi}^{(n,n_1)}$, each diagonal operator is factorized into two diagonal matrices; one is the transverse operator and the other being longitudinal. The lengths of the two matrices respectively equal to the number of transverse states and longitudinal states within that sector.

The factorization of operators involving no interactions between different sectors is quite straightforward. However, the factorization of the interaction operators on the right side of equation (2.50) should be taken with some carefully consideration. Before factorizing the interaction operators, we list several facts about the interaction:

1. When the fermion in a state of $\tilde{\psi}^{(n,n_1)}$ absorbs or emits a physical boson, a new state is generated in $\tilde{\psi}^{(n+1,n_1)}$ or $\tilde{\psi}^{(n-1,n_1)}$, respectively; the same is true for absorption or emission of a PV boson, for which a new state is generated in $\tilde{\psi}^{(n,n_1+1)}$ or $\tilde{\psi}^{(n,n_1-1)}$, respectively.

2. Momentum is conserved in the process of absorption and emission of bosons.

3. A state in sector $\tilde{\psi}^{(n,n_1)}$ interacts with a state in $\tilde{\psi}^{(n+1,n_1)}$, $\tilde{\psi}^{(n-1,n_1)}$, $\tilde{\psi}^{(n,n_1+1)}$ or $\tilde{\psi}^{(n,n_1-1)}$ if and only if the momentum of the emitted or absorbed boson equals to difference in the two fermions' momenta.

4. Denote the interaction operators, $\sum_{\underline{m}} \frac{1}{\sqrt{m}} \left(\frac{n-m}{n} \right)^{\gamma}$, $\sum_i \frac{1}{\sqrt{m_i}} \left(\frac{n}{n+m_i} \right)^{\gamma}$, $i \sum_{\underline{l}} \frac{1}{\sqrt{l}} \left(\frac{n-l}{n} \right)^{\gamma}$ and $i \sum_j \frac{1}{\sqrt{l_j}} \left(\frac{n}{n+l_j} \right)^{\gamma}$ as H_n^+, H_n^-, $H_{n_1}^+$ and $H_{n_1}^-$ respectively. It is obvious that these operators are independent of transverse momenta.

To apply the interaction operators on the wave function in SVD form, we define a transverse operator H^T as

$$H^T_{\mu_1\mu_2} = \begin{cases} 1 & \text{if } \mu_1 \text{ and } \mu_2 \text{ differ by only one pair of transverse momentum} \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

Thus, the factorization of interaction operators becomes

$$\begin{aligned} H_n^+ &\longrightarrow H^T \bigotimes H_n^+ \\ H_n^- &\longrightarrow H^T \bigotimes H_n^- \\ H_{n_1}^+ &\longrightarrow H^T \bigotimes H_{n_1}^+ \\ H_{n_1}^- &\longrightarrow H^T \bigotimes H_{n_1}^-. \end{aligned} \tag{3.10}$$

Having discussed factorization of the Hamiltonians, we are ready to apply H on $\tilde{\psi}$ of SVD form:

$$\tilde{\psi}'_{\alpha\beta} \equiv \sum_{\mu,\nu} H_{\alpha\beta\mu\nu} \tilde{\psi}_{\mu\nu}. \tag{3.11}$$

Inserting (3.7) and (3.5) into (3.11), one obtains

$$\tilde{\psi}'_{\alpha\beta} = \sum_{j=1}^{n_{svd}} \lambda_j \sum_{i=1}^{M} \sum_{\mu} H^i_{\alpha\mu} \phi_{\mu j} \sum_{\nu} H^i_{\beta\nu} \phi_{\nu j} \tag{3.12}$$

For the soluble model, M is equal to 7, meaning that we generally have 7 operators to apply on each wave state. Multiply λ_j into one of the sums over μ or ν, and denote $\sum_\mu H^i_{\alpha\mu}\phi_{\mu j}$ and $\lambda_j \sum_\nu H^i_{\beta\nu}\phi_{\nu j}$ as $\phi'_{\mu j}$ and $\phi'_{\nu j}$ respectively. The new state can then be written as

$$\tilde{\psi}'_{\alpha\beta} = \sum_{j=1}^{n_{svd}M} \phi'_{\mu j}\phi'_{\nu j}. \tag{3.13}$$

The small vectors labeled by ϕ' are non-orthonormal. The state in Equation (3.13) lies outside the n_{svd} subspace, and we need to project it back using P_{svd} in order not to further expand the memory cost. The orthonormalization and projection are carried out by following procedures [6]:

- We first orthonormalize $\phi'_{j\mu}$ and $\phi'_{j\nu}$ by diagonalizing the Hermitian overlap matrices (of dimensions $n_{svd}M \times n_{svd}M$)

$$\langle\phi'_j|\phi'_{j'}\rangle_i = (V_i D_i V_i^\dagger) \;\; i = \mu, \nu \tag{3.14}$$

  where D_i are diagonal and positive semidefinite, and V_i are unitary. The new orthormalized vectors are given by

$$|\alpha\rangle_i = \sum_j (D_i^{-\frac{1}{2}} V_i)_{\alpha j} |\phi'_j\rangle_i \;\; i = \mu, \nu, \tag{3.15}$$

  and the action of the Hamiltonian can be written as

$$H|\tilde{\psi}\rangle = \sum_{\alpha\alpha'}^{n_\alpha n_{\alpha'}} C_{\alpha\alpha'}|\alpha\rangle_\mu |\alpha\rangle_\nu, \tag{3.16}$$

  with

$$C = \sqrt{D_\mu} V_\mu^\dagger V_\nu \sqrt{D_\nu}. \tag{3.17}$$

- Perform an SVD on the matrix C,

$$C = U_\mu \Lambda U_\nu^\dagger, \tag{3.18}$$

  where U_μ, U_ν are unitary matrices and Λ is diagonal with entries λ_α.

- After computing Λ, U_μ, U_ν we obtain the SVD form of the state. To project the state in the $\mathbb{C}^{n_{svd}M}$ space into $\mathbb{C}^{n_{svd}}$ subspace, we keep only the n_{svd} largest λ_α

and their corresponding left- and right- singular vectors:

$$P_{svd}H|\tilde{\psi}\rangle = \sum_{\alpha=1}^{n_{svd}} \lambda_\alpha |\alpha'_\mu\rangle\alpha'_\nu\rangle, \tag{3.19}$$

where the small vectors of $i = \mu, \nu$ are,

$$|\alpha'_i\rangle = \sum_{j=1}^{n_{svd}M} (U_i D^{-\frac{1}{2}} V_i)_{\alpha j} |\phi_j)_i. \tag{3.20}$$

This completes the presentation of SVD projection applied to $H|\psi\rangle$. As we can see, by factorizing Hamiltonians into longitudinal operators and transverse operators, we get more small vectors after acting with H on $|\psi\rangle$. However, by applying the SVD projector to $H|\psi\rangle$ we are able to compress the wave-function back to n_{svd} subspace in order not to expand the memory cost.

Note that a potential problem in the compression of $H\psi$ might be that it introduces some compression error by removing the contributions from $n_{svd}(M-1)$ smaller singular values and their corresponding singular vectors. The compressed wave function is just an approximation to the exact wave function. The potential problem brought about by this approximation will be discussed in Section 3.2.2 and 4.3.1.

3.2 Lanczos Algorithm

3.2.1 Introduction to Lanczos Algorithm

In 1950, Lanczos introduced what has since become known as the Lanczos recursion or Lanczos tridiagonalization [4, 5]. Lanczos procedures for computing eigenvalues and eigenvectors of nondefective[1] symmetric matrices are based upon the Lanczos recursion for tridiagonalizing a symmetric matrix.

Let A be a $n \times n$ symmetric matrix and let $\mathbf{v}_1$ be a normalized initial guess for an eigenvector of A. For $N = 1, 2, ..., n$ define the corresponding Lanczos tridiagonal matrix

[1] A complex matrix is nondefective if and only if it is diagonalizable.

T_N using the following recursion. Define $\beta_1 \equiv 0$ and for $i = 1, 2, ..., n$, and define Lanczos vectors $\mathbf{v}_i$ and scalars α_i and β_{i+1} where

$$\begin{aligned} \mathbf{w}_i &= A\mathbf{v}_i \\ \alpha_i &= \mathbf{w}_i\mathbf{v}_i \\ \mathbf{w}_i' &= \mathbf{w}_i - \alpha_i\mathbf{v}_i - \beta_i\mathbf{v}_{i-1} \\ \beta_{i+1} &= \sqrt{\mathbf{w}_i'\mathbf{w}_i'} \\ \mathbf{v}_{i+1} &= \frac{\mathbf{w}_i'}{\beta_{i+1}}. \end{aligned} \tag{3.21}$$

For each N, the corresponding Lanczos matrix T_N is defined as the symmetric tridiagonal matrix with diagonal entries $\alpha_i, 1 \leq i \leq N$, and co-diagonal entries $\beta_{i+1}, 1 \leq i \leq (N-1)$. Therefore for each N,

$$T_N \equiv \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_N \\ & & \beta_N & \alpha_N \end{bmatrix}. \tag{3.22}$$

By definition it is clear that for each i, v_{i+1} is orthogonal to the two most recently-generated Lanczos vectors, v_i and v_{i-1}, and it is easy to prove that each succeeding Lanczos vector is orthogonal with respect to all previously-generated Lanczos vectors. Define the matrix $V_N \equiv \{\mathbf{v}_1, ..., \mathbf{v}_N\}$. We have that

$$V_N^T V_N = I_N. \tag{3.23}$$

Furthermore, for each N,

$$T_N = V_N^T A V_N \tag{3.24}$$

is the orthogonal projection of A onto the subspace spanned by V_N. It tells us that the eigenvalues of T_N end up being the eigenvalues of A restricted to the subspace $v_1, ..., v_N$. It is obvious that T_N is similar to A if $N = n$ and the eigenvalues of T_N are the same as those of A, but it is not useful to compute eigenvalues for large n. Instead, we make use of the Lanczos recursion to approach the approximation of extreme eigenvalues of A using relatively small N. It has been tested that for many matrices, and, for relatively

small N, several of the extreme eigenvalues of A are well approximated by eigenvalues of the corresponding Lanczos matrices, although it is not always true in practice. And it is generally true that at least one end of the eigenvalues is well approximated. In this thesis, we seek a smallest eigenvalue of the Hamiltonian.The following basic iterative, single-vector scheme is a suggested scheme for the computation of smallest eigenvalues (or the largest ones) for a symmetric matrix A of the order n.

1. Select a specified number of steps, $N \ll n$, and a normalized vector $\mathbf{v}_1^0$ and set $k = 0$.

2. For each iteration of k use the basic Lanczos recursion to generate a symmetric tridiagonal Lanczos matrix of order N, T_N^k. For each k, use the norm of the unnormalized second Lanczos vector $\mathbf{p}_1^k \equiv A\mathbf{v}_1^k - \alpha_1^k \mathbf{v}_1^k$ to check for convergence of the iterations. If convergence is observed, terminate.

3. Compute the algebraically-smallest(largest) eigenvalue of T_N^k and the corresponding eigenvector $\mathbf{u}^k$.

4. Compute the corresponding normalized Lanczos vector of A, $\mathbf{y}^k \equiv V_N^k \mathbf{u}^k$ ($V_N^k = \{\mathbf{v}_1^k, ..., \mathbf{v}_N^k\}$) and return to step 2, using this Lanczos vector as the new starting vector for the next iteration. That is, $\mathbf{v}_1^{k+1} \equiv \mathbf{y}^k$.

3.2.2 Modification of the Lanczos Algorithm

As has been indicated in Equation (3.23) and (3.24) that $V_N \equiv \{v_1, \cdots, v_N\}$ has to be an orthogonal basis set such that T_N is an orthogonal projection of A onto the subspace spanned by V_N. However, in our application, the third step of Lanczos iteration (3.21) involves compression, as our Lanczos vectors are in SVD form. This compression leads to a non-orthogonality in the basis set, and this non-orthogonality accumulates at each Lanczos iteration (further discussion will be given in Chapter 4). To solve this problem, we present a modified Lanczos scheme:

1. Select a normalized vector $\mathbf{v}_0$, compute the Krylov subspaces of A, $K_b(A, b) = \text{span}\ \{\mathbf{v}_0, A\mathbf{v}_0, \cdots, A^{b-1}\mathbf{v}_0\}$ [2] .

[2] If memory is scarce, one could set $b = 2$; the rate of convergence increases as b is increased. In our application, we set $b = 3$.

2. Orthonormalize this set of Lanczos vectors by computing the overlap matrix and applying Equation (3.14) and (3.15). This produces an orthonormal basis of dimension b, $U = \{\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_b\}$, in which $\langle \mathbf{u}_i | \mathbf{u}_j \rangle = \delta_{ij}$ $(i, j = 1, \cdots, b)$.

3. Project A orthogonally onto the subspace spanned by U, $H_{ij} = \langle \mathbf{u}_i | A | \mathbf{u}_j \rangle$.

4. Compute the lowest eigenvalue of H and its corresponding eigenvector $\mathbf{c}$. This lowest eigenvalue yields the best approximation to A at this level of iteration.

5. Compute the corresponding normalized Lanczos vector of A, $\mathbf{v}_0 \equiv U\mathbf{c}$ and return to step 1.

As has been indicated in Equation (2.53), the normalization of wave functions for our testing model contains extra factors. We have to include the extra factors in the normalization as well as in the dot products in the above iteration steps.

Note that the Lanczos vectors for the compression algorithm should be in SVD form; thus we have to develop codes for matrix dot products as well as matrix subtractions (or additions) in SVD form.

To calculate the dot product of matrix A and B, which are of SVD form, there is no need to store these two matrices in their original form. Instead, we can compute their elements with same indices, A_{ij} and B_{ij}, as needed. Given two $m \times n$ complex matrices, A and B, suppose the reduced SVD form of A and B are as follows,

$$A = USV^\dagger \quad \text{and} \quad B = XYZ^\dagger; \tag{3.25}$$

then their elements A_{ij} and B_{ij} can be expressed as

$$A_{ij} = \sum_{a=1}^{n_A} U_{ia} S_{aa} V_{aj}^\dagger \quad \text{and} \quad B_{ij} = \sum_{b=1}^{n_B} X_{ib} Y_{bb} Z_{bj}^\dagger, \tag{3.26}$$

where n_A and n_B are the number of singular values of A and B respectively. Thus, the dot product of A and B is as follows,

$$A \cdot B = \sum_{i}^{m} \sum_{j}^{n} \left(\sum_{a=1}^{n_A} U_{ia} S_{aa} V_{aj}^\dagger \right) \left(\sum_{b=1}^{n_B} X_{ib} Y_{bb} Z_{bj}^\dagger \right). \tag{3.27}$$

For the linear combinations (including additions and subtractions) of vectors in SVD form, we can apply Equation (3.13) to Equation (3.20). As these procedures involve truncating smaller singular values and corresponding vectors, there is no guarantee that the resulting SVD forms are exact. However, since the orthogonality holds for small b, and step 5 of the above scheme just generates an initial guess for the next iteration, this iteration scheme is still reliable.

The implementation of this algorithm as well as the descriptions of codes will be given in Chapter 4.

Chapter 4

Application to the Soluble Model

In this chapter, we will implement the compression algorithm into the testing model. Before implementation, we have to construct the basis set with specific cutoffs; this will be presented in section 4.1. We then provide necessary techniques in the application of the algorithm in Section 4.2. We end this chapter by presenting the results and analysis in Section 4.3. The descriptions and function of each code listed in the Appendix will be included throughout this chapter.

4.1 Cutoff and Basis

For regulation of the fermion self energy in light-cone quantization, the most commonly used cutoffs couple p^+ and $\mathbf{p}_\perp$ in some way. For our problem, we will include a cutoff on $\mathbf{p}_\perp$ alone and limit p^+ by the integer resolution K (discussed in section 2.3) naturally. In this way, we can store the basis states into sectors of a 2D matrix. The chosen cutoff on $\mathbf{p}_\perp$ is

$$\mathbf{p}_\perp^2 \leq \Lambda^2 = \frac{\pi^2 N_\perp^2}{{L_\perp}^2}. \tag{4.1}$$

Thus for each elements of $\mathbf{p}_\perp$, we have

$$|p_{\perp x,y}| \leq \frac{\pi N_\perp}{L_\perp} \tag{4.2}$$

Given that $p_{\perp x,y} = \frac{\pi n_{x,y}}{L_\perp}$, it is obvious that the transverse integers $n_{x,y}$ can range from $-N_\perp$ to $N_\perp$.

In the testing model, for a given state $\tilde{\psi}^{(n,n_1)}(\underline{m}_i, \underline{l}_j, \underline{n})$ with n physical bosons, n_1 PV bosons and a fermion, the limitation of the transverse momenta in the list of $(\underline{m}_i, \underline{l}_j, \underline{n})$ is given as follows,

$$\begin{aligned} -N_\perp \leq & m_{ix}, m_{iy}, l_{jx}, l_{jy} \leq N_\perp, \\ -N_\perp \leq n_x = & -(\sum_i^n m_{ix} + \sum_j^{n_1} l_{jx}) \leq N_\perp, \\ -N_\perp \leq n_y = & -(\sum_i^n m_{iy} + \sum_j^{n_1} l_{jy}) \leq N_\perp, \\ & n_x^2 + n_y^2 \leq N_\perp^2. \end{aligned} \tag{4.3}$$

We name these relations as Cutoff-1 (later we will have a modified cutoff for computational convenience). With Cutoff-1 we can generate the basis set with a specific $N_\perp$ and K ($N_\perp$ and K can be assigned in *parameter.h*, see Appendix A.1.1). The basis set is produced by using *basis.h* (See Appendix A.1.2), *AllSumToN2.h* (See Appendix A.1.3), and namespace[1] *Sum2N* (See Appendix A.1.4).

AllsumToN2.h defines necessary functions to create the longitudinal number list for n physical bosons, n_1 PV bosons, and a single fermion such that

- The sum of longitudinal integers for $(n + n_1)$ bosons and the fermion are K;
- Longitudinal integers for bosons are positive and even, while for the fermion the integer is positive and odd.

Namespace *Sum2N* defines necessary functions to generate the transverse number list for n physical bosons, n_1 PV bosons, and a single fermion such that transverse integers satisfy relations (4.3). *basis.h* is used to construct and store the longitudinal basis and transverse basis into vector b and vector d, respectively. The vector s is to store the number of physical bosons and PV bosons, (n, n_1), in each sector.

By assigning different values to $N_\perp$ and K in *parameter.h*, a table of number of states (i.e. basis size) is presented in Table 4.1,

[1] Namespaces in C++ group entities like classes, objects and functions under a name. This way the global scope can be divided into "sub-scopes", each one with its own name.

Table 4.1: Basis sizes for the compression algorithm applied to the soluble model with parameters $M^2 = \mu^2$,$\mu_1^2 = 10\mu^2$, and Cutoff-1.

	K							
$N_\perp$	3	5	7	9	11	13	15	17
1	11	88	506	2490	10530	40007	138085	441356
2	27	516	6958	77626	733018	6043595	44355945	294428852
3	59	2032	51994	1077450	18756390	282494347		
4	99	5484	227990	7620750	213606602			
5	163	12964	807062	39878422	1651610558			
6	227	25084	2165638					
7	299	43940	5021446					
8	395	73736	10836002					

To reach the momentum-space continuum limit, K and $N_\perp$ have to be infinite. In order to get good approximations, we want K and $N_\perp$ to be large. As we can see in Table 4.1, the basis size reaches up to 1.6 billion when $K = 11$ and $N_\perp = 5$, the memory cost to store the Hamiltonian can be as large as 2 Exabyte (EB). As our goal is to investigate the rate of convergence as we include the compression in the Lanczos algorithm, we want to truncate more of the basis set, for computation convenience. Here we introduced Cutoff-2, whose relations are derived from relation (4.3):

$$\begin{aligned} -1 \leq & m_{ix}, m_{iy}, l_{jx}, l_{jy} \leq 1 \\ -\frac{N_\perp}{2} \leq n_x = & -(\sum_i^n m_{ix} + \sum_j^{n_1} l_{jx}) \leq \frac{N_\perp}{2} \\ -\frac{N_\perp}{2} \leq n_y = & -(\sum_i^n m_{iy} + \sum_j^{n_1} l_{jy}) \leq \frac{N_\perp}{2} \\ & n_x^2 + n_y^2 \leq \frac{N_\perp^2}{4}. \end{aligned} \tag{4.4}$$

In Cutoff-2, we limit the transverse momentum number for bosons to be -1, 0 and 1. We also include a factor of 1/2 in the limit of fermion's transverse momentum number,

thus the upper limit for its squared transverse momentum number has a factor of 1/4. A table of the number of states (i.e. basis size) for Cutoff-2 is given in Table 4.2.

Table 4.2: Basis sizes for the compression algorithm applied to the soluble model with parameters $M^2 = \mu^2$,$\mu_1^2 = 10\mu^2$, and Cutoff-2.

	K						
$N_\perp$	5	7	9	11	13	15	17
2	88	506	2490	10530	40007	138085	441356
3	136	834	4106	17746	67823	236413	749420
4	164	1022	5254	23046	89739	316073	1025804
5	196	1326	7142	32310	128619	460633	1514364
6	208	1458	8174	37950	154511	562605	1875768
7	208	1554	8910	42526	175903	650301	2191608
8	208	1602	9494	46478	196631	740133	2533288

For computation convenience, we will use Cutoff-2 and its corresponding basis for the testing of our algorithm.

4.2 Numerical techniques

Having constructed the basis set, we come to build the Hamiltonian matrices using Equation (2.52). The Hamiltonian matrices are generated and stored by *Hamiltonian.h* (See Appendix A.2.1) and *cover.h* (See Appendix A.2.2).

As has been discussed in section 3.1.3, the Hamiltonians are factorized into longitudinal parts and transverse parts. The diagonal matrices, which involve no interaction between different sectors, are created by applying Equation (3.8) and stored in vector A in *Hamiltonian.h*. By applying Equation (3.9) and (3.10), the longitudinal and transverse interaction matrices can be generated. Because of the symmetry of Equation

(2.52), we only need to store H_n^+, $H_{n_1}^+$ (or H_n^-, $H_{n_1}^-$) and their corresponding H^Ts. To specify non-zero elements in the transverse operators H^T and longitudinal operators, H_n^+ and $H_{n_1}^+$, we define a header file *cover.h* to identify if two states differ only by one PV boson or one physical boson. As can be seen in $Hamiltonian.h$, H_n^+ and its corresponding H^T are stored in vector BL and vector BT respectively, while $H_{n_1}^+$ and its corresponding H^T are stored in vector DL and vector DT respectively.

With longitudinal and transverse Hamiltonians created and stored by $Hamiltonian.h$, we can apply these Hamiltonians on wave functions of SVD form by going through the compression procedures from Equation (3.13) to Equation (3.20). Here we define a header file $HDotU.h$ (See Appendix A.2.3) for this application. In this header file, with input Hamiltonians (A, BL, BT, DL, DT), basis vectors (b, d, s) and the SVD form of a wave function($\psi = tv \cdot Sigma \cdot lv^\dagger$), the resulting SVD form of $H|\psi\rangle$ is stored in TV, $Sigma1$ and LV, i.e. $H|\psi\rangle = TV \cdot Sigma1 \cdot LV^\dagger$.

In $HDotU.h$, we include a header file *compression.h* (See appendix A.2.4) to carry out the compression procedures. Vectors stored in A and B lie outside the n_{svd} space and we use this header file to project them back into the n_{svd} space and store the compressed SVD matrices in $Alpha1$, $Alpha2$ and $Lamda$. We will also need to use this header file in the modified Lanczos iterations. As the Lanczos vectors are of SVD form, the compression procedures also apply to the calculations of linear combinations of these Lanczos vectors.

Before we go to the Lanczos iterations, we first introduce four header files $SvdSum.h$, $SvdProduct.h$, $SvdNorm.h$ and *count.h* (See Appendix A.3.1, A.3.2, A.3.3 and A.3.4). Given two matrices of SVD form, $SvdSum.h$ calculates the sum of these two matrices. As has been indicated, the resulting sum of these two matrices has to be in SVD form in order to save memory, thus *compression.h* is included in $SvdSum.h$. $SvdProduct.h$ calculates the dot product of two matrices, and $SvdNorm.h$ calculates the norm of a matrix (these matrices are all in SVD form). *count.h* is included in these two header files to add the normalization factor in Equation (2.53) to the calculations of dot products and norms.

For Lanczos iterations, we tried two schemes to investigate the convergence. First we tried simple Lanczos iterations. We set the iteration number N, generate the transformed tridiagonal matrix T_N, and look for its smallest real eigenvalue. This scheme is presented in the main file *Lanczos.cpp* with the header file *Eigenvalue.h* (See Appendix A.3.5 and A.3.6). Then we tried modified Lanczos iterations; this scheme is presented in the main file *power.cpp*(See Appendix A.3.7).

4.3 Results

In this section, we give the results of the two iteration schemes mentioned above. The drawbacks of simple Lanczos iterations in its application with the compression algorithm is presented in the first subsection. In the second subsection, we present converged results with several tables and figures generated by the second scheme. We end this section with the study of convergence as we change the compression parameter n_{svd}.

4.3.1 Simple Lanczos iterations

In using simple Lanczos iterations, we found that the smallest real eigenvalue disappears at some iteration steps. Then we studied the orthogonality of the Lanczos vectors as we change the iteration number N.

Table 4.3 gives the biggest dot product of three most recently generated Lanczos vectors and the eigenvalue with smallest norm at each level of iteration. As we can see, the orthogonality in the Lanczos vectors is lost, as with more iterations. Since we only stored three Lanczos vectors at each level of iteration, the table only shows the non-orthogonality between these three vectors. The non-orthogonality between recently-generated Lanczos vectors and most previous Lanczos vectors could be even larger. Due to the non-orthogonality in the Lanczos vectors, the smallest real eigenvalue disappears at the fifth iteration, which fails the Lanczos iterative scheme.

Table 4.4 uses the same parameters as Table 4.3 except that we changed the compression

parameter n_{svd} from 3 to 10. With less compression in the case of Table 4.4, the non-orthogonality shows up later than the case in Table 4.3, which confirms our conjecture that the non-orthogonality comes from the compression procedures.

Table 4.3: Non-orthogonality of Lanczos vectors and smallest eigenvalues in simple Lanczos iterations with $n_{svd} = 3$. The biggest dot product of the three most recently generated Lanczos vectors is presented as the non-orthogonality at each level of iteration N. The real eigenvalue disappears at some iteration steps, so the presented eigenvalues are those with the smallest norm at each N.

N	Non-orthogonality	Smallest eigenvalue
3	$2.9560 \times 10^{-15} - 2.1528 \times 10^{-18}i$	$13.593 + 1.9797 \times 10^{-05}i$
4	$2.1391 \times 10^{-14} + 2.6916 \times 10^{-17}i$	$11.546 - 0.0053472i$
5	$9.1150 \times 10^{-12} + 5.0992 \times 10^{-10}i$	$636.74 + 0.00064955i$
6	$0.079480 + 0.19038i$	$-13368 + 7794.5i$
7	$-0.0035725 + 0.00041781i$	$-0.88631 - 0.50269i$
8	$-0.0029595 - 0.0023103i$	$45.301 - 3.4532i$
9	$-0.0068843 - 0.0040854i$	$10.842 - 6.3222i$
10	$0.0040387 - 0.0076722i$	$11.383 + 2.9360i$
11	$0.044876 - 0.088772i$	$7.78532 + 3.07853i$

With the orthogonality lost in the application of the compression procedures, we are unable to get converged smallest real eigenvalues. But because the non-orthogonality is reasonably acceptable at the early stage of iterations, the compression algorithm will still be useful if we avoid generating a large nonorthogonal basis set. This is also the motivation of our modified Lanczos algorithm discussed in Section 3.2.2, which avoids the subtraction of SVD vectors in (3.21).

Table 4.4: Same as Table 4.3 but with $n_{svd} = 10$.

N	Non-orthogonality	Smallest eigenvalue
3	$4.7462 \times 10^{-15} + 1.6514 \times 10^{-19}i$	$13.593 + 1.9797 \times 10^{-05}i$
4	$2.6227 \times 10^{-14} + 8.0194 \times 10^{-18}i$	$11.532 - 0.16084i$
5	$-1.3200 \times 10^{-10} + 4.8772 \times 10^{-8}i$	$1.2956 - 0.031825i$
6	$-2.6626 \times 10^{-9} - 7.1829 \times 10^{-7}i$	$0.97915 - 0.11113i$
7	$1.7506 \times 10^{-11} + 4.2362 \times 10^{-10}i$	$-0.22355 + 0.0030527i$
8	$1.2013 \times 10^{-9} - 1.7260 \times 10^{-9}i$	$7.8732 + 0.29231i$
9	$0.0041738 - 8.00175 \times 10^{-5}i$	$37.881 - 41.913i$
10	$0.0037151 - 0.004258i$	$0.76936 - 0.61314i$
11	$3.6027 \times 10^{-6} - 2.2633 \times 10^{-6}i$	$220.46 + 21.951i$

4.3.2 Modified Lanczos iterations

We have solved the discrete eigenvalue problem (2.52) for various cases. We use the basis sets generated by Cutoff-2, and the physical parameter values chosen were $\gamma = 1/2$, $\mu_1^2 = 10\mu^2$. The value of the compression parameter, n_{svd}, we have chosen is 3. The parameters that control the numerical approximation, namely K, $N_\perp$ and M_0^2, were varied to study convergence with basis size up to $\sim$ 320 000. These parameter values are chosen in order to be as close as possible to those used by Brodsky and his coworkers' [7], to have some basis for comparison.

Given g, μ, M, μ_1 and a cutoff, Brodsky and his coworkers solved for the wave function ψ and M_0^2. The cutoff was used to fix $L_\perp$, and the expectation value for the square of the scalar field ϕ was used to fix the bare coupling g. The value of M they used is the analytic value, $(M/\mu)^2 = 1$. For computational convenience, we choose to use the values of $L_\perp$, g, and M_0^2 in their work and solve for $(M/\mu)^2$. The range of these numerical parameters and our results are shown in Table 4.5.

Table 4.5: Numerical parameter values and results from solving the model eigenvalue problem for M^2. Here $n_{svd} = 3$.

K	$N_\perp$	$\mu L_\perp/\pi$	$(M_0/\mu)^2$	g/μ	$(M/\mu)^2$
11	7	0.8165	0.8547	13.293	1.48160
11	6	0.8165	0.8547	13.293	1.49232
11	5	0.8165	0.8547	13.293	1.49470
11	4	0.8165	0.8547	13.293	1.51021
13	4	0.8165	0.8518	13.230	1.49476

Note that we use the same parameter values for cases when $K = 11$, $N_\perp = 7,\ 6,\ 5,\ 4$ since the basis size for these cases vary little with Cutoff-2. Because we use a cutoff which is different from that of Brodsky and his coworkers' work, these parameter values are just approximations for our calculations. However, we are able to obtain approximate converged real eigenvalues for $(M/\mu)^2$. These approximations are close to the analytic value, $(M/\mu)^2 = 1$.

Figure 4.1: Eigenvalue convergence as a function of iteration, for the case with $K = 11$, $N_\perp = 4$ and $n_{svd} = 3$.

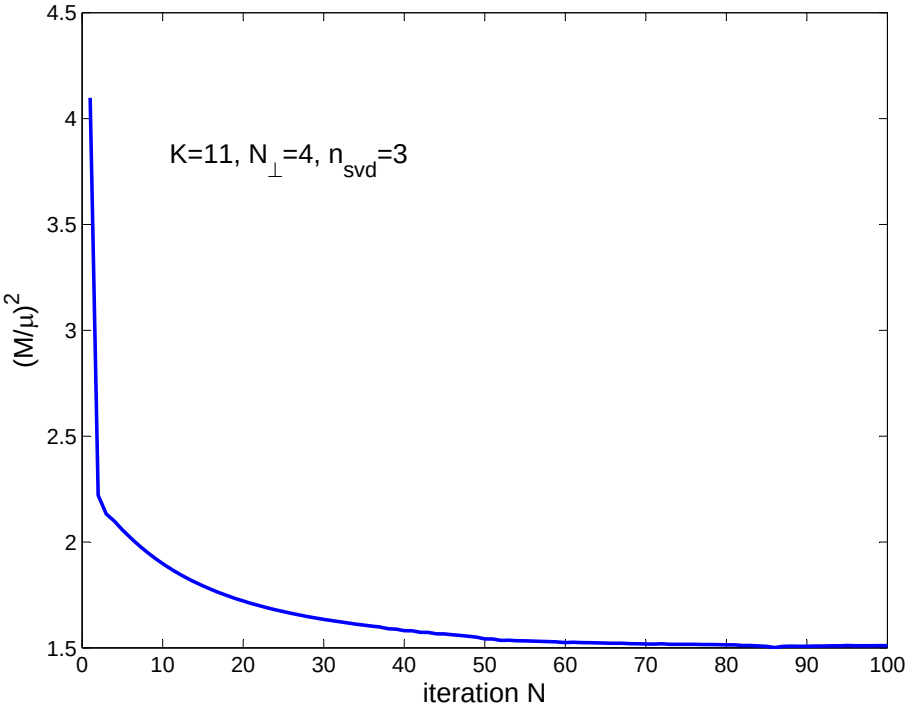
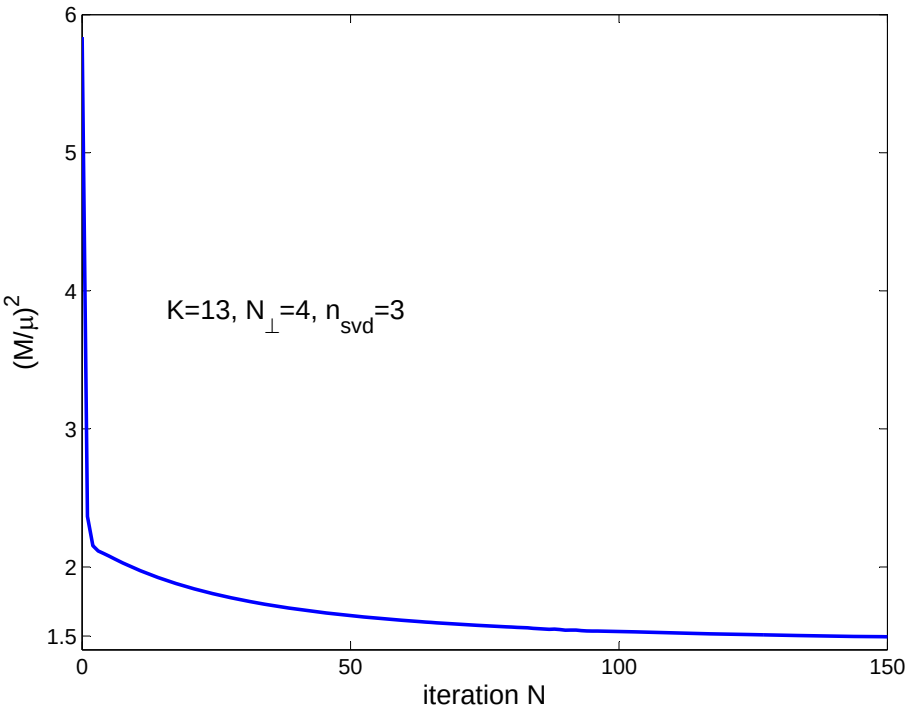


Figure 4.2: Eigenvalue convergence as a function of iteration, for the case with $K = 13$, $N_\perp = 4$ and $n_{svd} = 3$.



Figures 4.1 and 4.2 show the convergence of $(M/\mu)^2$ versus iteration for the case with $K = 11$, $N_\perp = 4$ (Case 1) and the case with $K = 13$, $N_\perp = 4$ (Case 2) respectively. We use $n_{svd} = 3$ and the modified Lanczos iterations. As we can see, there is some difference between our computed values and the analytic value. This is due to the fact that our work is performed in the framework of DLCQ, which makes it impossible to get better results than DLCQ approximations. However, as our work is to study the compression algorithm, it is more important to see how much compression is obtained and how good the convergence is with respect to the compression parameter, n_{svd}.

To study how much memory is reduced by the compression algorithm, we define a compression ratio,

$$\Theta(n_{svd}) = \frac{M_{svd}}{M_{1d}} \times 100\% \tag{4.5}$$

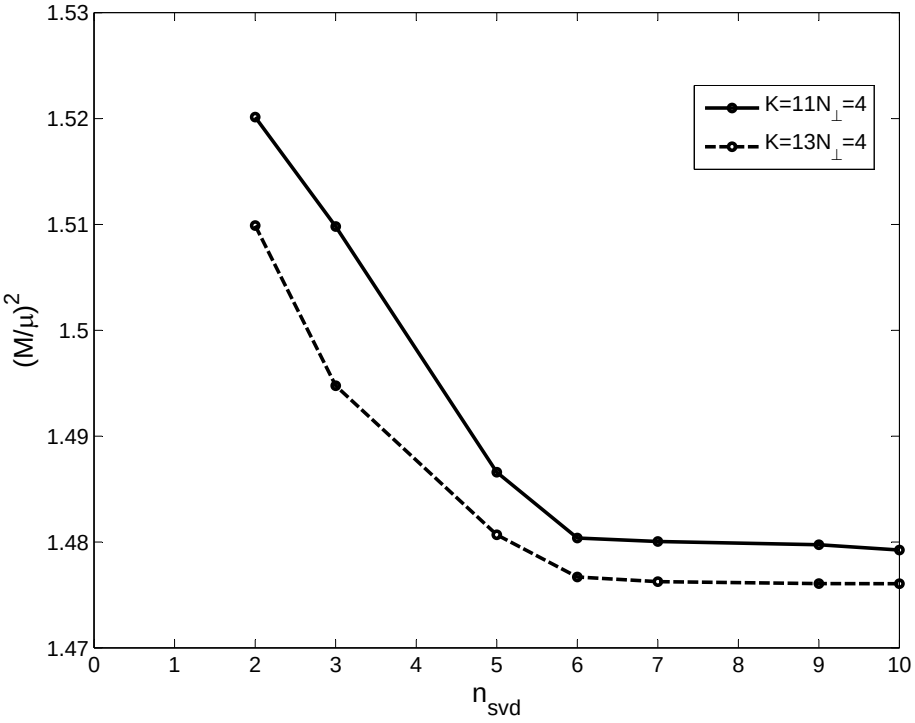
where M_{svd} is the memory used for a single vector in SVD form for the given n_{svd} value, and M_{1d} is the memory used to store the vector as a one-dimensional array.

We compute the converged smallest eigenvalues and the compression ratios for various cases, as presented in Table 4.6.

Table 4.6: Converged smallest eigenvalues for $(M/\mu)^2$ and compression factors Θ for the cases with $K = 11$, $N_\perp = 4$ (Case 1) and $K = 13$, $N_\perp = 4$ (Case 2) at each level of compression, n_{svd}.

	$\Theta(\%)$		$(M/\mu)^2$	
n_{svd}	Case 1	Case 2	Case 1	Case 2
2	75.06	68.86	1.520135	1.509895
3	88.86	85.47	1.509800	1.494760
5	95.61	88.58	1.486590	1.480679
6	98.49	92.26	1.480375	1.476694
7	100.99	95.90	1.480044	1.476255
9	101.67	98.57	1.479744	1.476064
10	102.02	99.23	1.479235	1.476062

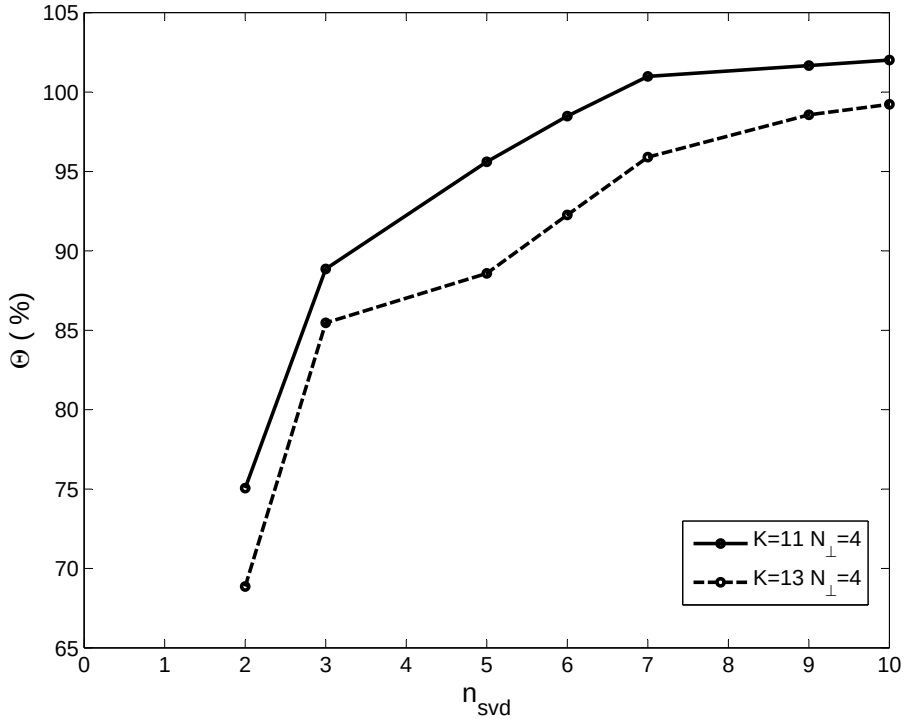
Figure 4.3: Converged eigenvalues as a function of n_{svd}, for the cases with $K = 11$, $N_\perp = 4$ and $K = 13$, $N_\perp = 4$.



In Figure 4.3 we show the converged smallest eigenvalue for $(M/\mu)^2$, as a function of n_{svd}. As we increase n_{svd}, the smallest eigenvalues converge to the last eigenvalue with no compression or little compression.[2] In Figure 4.4 we show the compression ratio Θ as a function of n_{svd}. As we increase n_{svd}, Θ increases.

[2] The largest numbers of longitudinal states for all sectors in Case 1 is 10, so the last eigenvalue in Figure 4.3 is the one with no compression; while for Case 2, the last eigenvalue in figure is with little compression, since the no-compression limit for this case is when $n_{svd} = 15$.

Figure 4.4: Compression ratio Θ as a function of n_{svd}, for the cases with $K = 11$, $N_\perp = 4$ and $K = 13$, $N_\perp = 4$.



The compression ratio at small n_{svd} is not as small as we have expected. This is due to the slimness of the matrices in the Lanczos vectors for small values of K. As has been indicated in Section 4.1, the longitudinal momentum is limited by K. In cases with small K, there are fewer longitudinal states compared with transverse states. The largest numbers of longitudinal states for all sectors in Case 1 and Case 2 are 10 and 15, respectively; while the numbers of transverse momenta states have reached up to 2917 and 9093, respectively. With this slimness in cases with small K, the compression algorithm does not reduce the memory requirements very much. However, as we increase K, we can expect much more compression. This can be implied by the comparison between the compression ratios in Case 1 and Case 2 shown in Table 4.6 and Figure 4.4. When K approaches to infinity and $N_\perp$ is fixed to a specific value, we can expect the compression ratios to be the smallest.

Chapter 5

Conclusion

In this thesis we have developed a compression algorithm for quantum field theoretic calculations. Given a quantum-field theoretic eigenvalue problem, we can apply this algorithm to reduce memory requirements for eigenvectors. Unlike traditional one-dimensional basis vectors, we store basis states as two-dimensional matrices, in which one index stands for longitudinal momenta and the other represents transverse momenta. With this adjustment, we are able to compress the wave functions by SVD. The Lanczos algorithm is used to approach the smallest eigenstates. Our compression algorithm reduces memory cost, but relative to standard Lanczos, it demands longer computation time due to additional matrix manipulations.

We tried to use simple Lanczos iterations to compute the smallest eigenvalues, but we found that the SVD projection interferes with the Lanczos convergence. The SVD projection introduces a truncation error in generating Lanczos vectors and thus induces non-orthogonality in the Lanczos vectors. This non-orthogonality is quite small at the early stage of iterations, but becomes significant as the iterations progress. With non-orthogonality introduced, the smallest eigenvalues of the transformed tridiagonal matrices no longer converge and even disappear at some iteration levels.

We modified the Lanczos iterations by only keeping b[1] Lanczos vectors and orthonormalizing these vectors. By projecting the Hamiltonian onto this vector subspace, we are

[1] b should be a small number so that the non-orthogonality induced is not significant.

able to get a reduced Hamiltonian. Its smallest eigenvalue yields the best approximation to the ground state, and the resulting wave function becomes the initial state for the next iteration.

By applying our compression algorithm to a model theory, we are able to approximate the ground state. The converged eigenvalue generated by the modified Lanczos iterations is close to the analytic value. With this test model, we then studied the convergence of the ground state with respect to the compression parameter n_{svd}. Small n_{svd} saves more memory but also introduces a computational error; however, this error is quite small. This matches our expectation that the compression algorithm reduces memory cost with little sacrifice in the accuracy of calculations.

We also studied how much the memory cost is reduced by the compression algorithm. Our results show that the compression ratio at small n_{svd} is not as small as expected. This is due to the slimness of the matrices in the Lanczos vectors. To avoid the slimness, a potential solution might be to include an appropriate cutoff so that the number of longitudinal states are comparable with that of transverse states in each sector. Another idea is to separate the transverse momentum, p_x and p_y, into two dimensions. With p_x and p_y separated into two dimensions, one can obtain nearly square matrices in all sectors of the Lanczos vectors, which avoids the slimness and reduces memory requirements.

This work could be extended in various ways. For computation convenience, this work stores the Hamiltonian matrix in static memory, but if the Hamiltonian matrix is computed as needed, as must be done in a real application, additional tests could be done at larger values of K and $N_\perp$. Theories in $2+1$ dimensions, including a lower-dimensional version of this model, might be the more natural place for this algorithm, because having one less transverse dimension would bring the wave function matrices closer to the square shape that is optimal for compression. A first application to a real, (3+1)-dimensional theory would naturally be to Yukawa theory, where, unlike the present model, the fermion would have its own dynamics and the interactions would include spin.

References

[1] Hans-Christian Pauli and Stanley J Brodsky. Solving field theory in one space and one time dimension. *Phys. Rev. D*, 32(8):1993, 1985.

[2] Hans-Christian Pauli and Stanley J Brodsky. Discretized light-cone quantization: Solution to a field theory in one space and one time dimension. *Physical Review D*, 32(8):2001, 1985.

[3] Toshihide Maskawa and Koichi Yamawaki. The problem of $p^+ = 0$ mode in the null-plane field theory and dirac's method of quantization. *Progress of Theoretical Physics*, 56(1):270–283, 1976.

[4] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office, 1950.

[5] Jane K Cullum and Ralph A Willoughby. *Lanczos algorithms for large symmetric eigenvalue computations. Vol I: Theory, Vol II: programs*. Birkhäuser. Reprinted by SIAM in the series Classics in Applied Mathematics, 1985.

[6] Marvin Weinstein, Assa Auerbach, and V. Ravi Chandra. Reducing memory cost of exact diagonalization using singular value decomposition. *Phys. Rev. E*, 84:056701, Nov 2011.

[7] Stanley J. Brodsky, John R. Hiller, and Gary McCartor. Pauli-villars regulator as a nonperturbative ultraviolet regularization scheme in discretized light-cone quantization. *Phys. Rev. D*, 58:025005, Jun 1998.

[8] Paul AM Dirac. Forms of relativistic dynamics. *Reviews of Modern Physics*, 21(3):392, 1949.

[9] OW Greenberg and SS Schweber. Clothed particle operators in simple models of quantum field theory. *Il Nuovo Cimento*, 8(3):378–406, 1958.

[10] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970.

[11] Leslie Hogben. *Handbook of linear algebra*. Chapman & Hall, 2007.

Appendix A

Code listings

A.1 Basis

A.1.1 parameter.h

```
1  #ifndef PARAMETER_H
2  #define PARAMETER_H
3  int nperp=4; double K=13.0;
4  double g_over_mu=13.230;
5  long double M0_prime= pow(g_over_mu,2)*log(10)/(32*pow(M_PI,
    2));
6  double M0=0.8518;
7  double L_perp=0.8165;
8  double mu1=10;
9  double coeff=g_over_mu/(L_perp*sqrt(8*pow(M_PI,3)));
10 int n_svd=2;
11 #endif
12
```

A.1.2 basis.h

```
#ifndef BASIS_H
#define BASIS_H
#include <iostream>
#include <iomanip>
#include <string.h>
#include <algorithm>
#include <vector>
#include <cassert>
#include "Sum2N.h"
#include "AllSumToN2.h"
#include "parameter.h"
using std::vector;
using namespace std;
using namespace Sum2N;
void basis(vector<vector<vector<int> > > &b,vector<vector<
vector<pair<int,int> > > > &d,vector<vector<int> > &s)
{
 int Ns=1;//# states, initialize it to 1 to include the
 state(5,0,0),which only has one fermion.
 int N=0;//N is # bosons
 int Nb=0;//Nb is # physical bosons
 int Nv=0;//Nv is # pauli-villars
 vector<int> s1(2);// s1 is the sub-vector inside s
 vector<vector<int> >b1;// b1 is the first layer of b
 vector<int>b2;//b2 is the second layer of b
 for (N=1;N<=(K-1)/2;N++)// The total number can range from
 1 up to (K-1)/2
 {
 for (Nb=0;Nb<=N;Nb++)
 {
  Nv=N-Nb;
  s1[0]=Nb;s1[1]=Nv;//cout<<"Nb="<<Nb<<" "<<"Nv="<<Nv<<endl;
  vector<vector<pair<int,int> > > d1=case1(N,nperp,Nv);
  d.push_back(d1);//d is the 3D vector to store the
  transverse momentum,
  d1.clear();
  int T=d1.size();// for each row of s, how many possible
  transverse states are there?
  s.push_back(s1);//s is s n by 2 matrix to store data for
  # of p-b and p-v
  if(Nv==0)
  {
  for(int m=2*Nb;m<=K-1;m+=2)
  {vector<vector<int> > a = getAllSums2(m/2, Nb);
  for (vector<vector<int> >::const_iterator it = a.begin();
  it != a.end(); ++it)
  {
```

```
41      for (vector<int>::const_iterator it2 = it->begin(); it2 !=
         it->end(); ++it2)
42      b2.push_back(*it2*2);
43      b1.push_back(b2);
44      b2.clear();
45      }
46      }
47      }
48      else if(Nb==0)
49      {
50      for(int l=2*Nv;l<=K-1;l+=2)
51      {vector<vector<int> > a = getAllSums2(l/2, Nv);
52      for (vector<vector<int> >::const_iterator it = a.begin();
        it != a.end(); ++it)
53      {
54      for (vector<int>::const_iterator it2 = it->begin(); it2 !=
         it->end(); ++it2)
55      b2.push_back(*it2*2);
56      b1.push_back(b2);
57      b2.clear();
58      }
59      }
60      }
61      else
62      {
63      for (int m=2*Nb;m<=K-1-2*Nv;m+=2)
64      {
65      for(int l=2*Nv;l<=K-1-m;l+=2)
66      { vector<vector<int> > a = getAllSums2(m/2, Nb);
67        vector<vector<int> > aa = getAllSums2(l/2, Nv);
68      for (vector<vector<int> >::const_iterator it = a.begin();
        it != a.end(); ++it)
69      {
70      for(vector<vector<int> >::const_iterator it1 = aa.begin();
         it1 != aa.end(); ++it1)
71      {
72      for (vector<int>::const_iterator it2 = it->begin(); it2 !=
         it->end(); ++it2)
73      b2.push_back(*it2*2);
74      for (vector<int>::const_iterator it3 = it1->begin(); it3
        != it1->end(); ++it3)
75      b2.push_back(*it3*2);
76      b1.push_back(b2);
77      b2.clear();
78      //b2.erase (b2.end()-(*it1).size(),myvector.end());
79      }
80      }
```

```
81      }
82      }
83      }
84      Ns=Ns+T*b1.size();
85      b.push_back(b1);// b is the 3D vector to store
        longitudinal momentum,
86      b1.clear();
87      }
88     }
89    }
90    #endif
91
```

A.1.3 AllSumToN2.h

```
#ifndef ALLSUMTON2_H
#define ALLSUMTON2_H
/*
 *
 * AllSumToN2
 *
 * getAllSums returns an ordering of positive integers (a1, a2, a3, ... ar)
 *   s.t. a1 + a2 + a3 + .. + ar = n
 * Example:
 *   getAllSums(5, 3) returns 1 1 3, 1 2 2, 1 3 1, 2 1 2, 2 2 1, 3 1 1
 *
 * getAllSums2 returns a set of positive integers (a1, a2, a3, ... ar)
 *   s.t. a1 + a2 + a3 + .. + ar = n
 * Example:
 *   getAllSums(5, 3) returns 1 1 3, 1 2 2
 *
 */

#include <vector>

std::vector<std::vector<int> > getAllSums(int n, int r);
std::vector<std::vector<int> > getAllSums2(int n, int r);

#endif

```

```
1   /*
2    *  Generate combinations:
3    *
     http://stackoverflow.com/questions/9430568/generating-combin
     ations-in-c
4    */
5
6   #include "AllSumToN2.h"
7   #include <iostream>
8   #include <algorithm>
9   #include <iterator>
10  #include <cassert>
11  #include <stdexcept>
12  #include <map>
13
14  using namespace std;
15
16  vector<vector<bool> > generateCombinations(int n, int r)
17  {
18      vector<vector<bool> > combinations;
19      vector<bool> v(n);
20      fill(v.begin() + r, v.end(), true);
21
22      do {
23          vector<bool> c(n);
24          for (int i = 0; i < n; ++i)
25              c[i] = !v[i];
26          combinations.push_back(c);
27      } while (next_permutation(v.begin(), v.end()));
28
29      return combinations;
30  }
31
32  vector<vector<int> > getAllSums(int n, int r) {
33      vector<vector<int> > vec;
34      if (n < r || n < 0 || r <0||r==0) {
35          throw range_error("Invalid input in getAllSums");
36      }
37
38      vector<vector<bool> > combinations =
        generateCombinations(n-1, r-1);
39      vector<int> a(r);
40      for (vector<vector<bool> >::const_iterator c =
        combinations.begin();
41           c != combinations.end(); ++c) {
42          for (int m = 0; m < r; m++)
43              a[m] = 0;
```

```
44          a[0] = 1;
45          int i = 0;
46          for (int sep = 0; sep < n - 1; sep++)
47              if (!(*c)[sep])
48                  a[i]++;
49              else
50                  a[++i]++;
51
52          vec.push_back(a);
53      }
54
55      return vec;
56  }
57
58  vector<vector<int> > getAllSums2(int n, int r)
59  {
60      vector<vector<int> > a = getAllSums(n, r);
61      vector<vector<int> > vec;
62      for (vector<vector<int> >::const_iterator it = a.begin
        (); it != a.end(); ++it)
63      {
64          bool add = true;
65          for (vector<int>::const_iterator it2 = it->begin();
            it2 != it->end() - 1; ++it2)
66              if (*it2 > (*(it2 + 1)))
67              {
68                  add = false;
69                  break;
70              }
71          if (add)
72              vec.push_back(*it);
73      }
74      return vec;
75  }
76
```

A.1.4 Sum2N.h

```
1  /* SumToN
2   * generateCombinationsTest(5, 3) returns
3   *   123, 124, 125, 134, 135, 145, 234, 235, 245, 345
4   * getAllSums returns all lists of n positive integers (a1,
    a2, a3, ... an)
5   *   s.t. a1 + a2 + a3 + .. + an = sum
6   *   getAllSums(3, 5) returns 1 1 3, 1 2 2, 1 3 1, 2 1 2, 2
    2 1, 3 1 1
7   * getAllSumBetween returns all lists of integers (a1, a2,
    a3, ... an)
8   *   s.t. loBound <= ai <= hiBound, and loSum <= a1 + a2 +
    ... + an <= hiSum
9   * getAllSquaredSumBetween returns all lists of integers
    (a1, a2, a3, ... an)
10  *   s.t. loBound <= ai <= hiBound, and loSum <= a1^2 +
    a2^2 + ... + an^2 <= hiSum*/
11 #ifndef SUM2N_H
12 #define SUM2N_H
13 #include <vector>
14 #include <iostream>
15 #include <algorithm>
16 #include <stdexcept>
17 #include <cassert>
18 #include <cmath>
19 #include <map>
20 #include <set>
21 namespace Sum2N
22 {   int binomialCoefficient(int n, int k); // n choose k
23     std::vector<std::vector<bool> > generateCombinations(int
       n, int r);
24     std::vector<std::vector<int> > getAllSums(int n, int sum
      );
25     std::vector<std::vector<int> > getAllSumBetween(int n,
      int loSum, int hiSum, int loBound, int hiBound);
26     std::vector<std::vector<int> > getAllSquaredSumBetween(
      int n, int loSum, int hiSum, int loBound, int hiBound);
27     void binomialCoefficientTest(); // assert
28     void generateCombinationsTest(int n, int r); // print
29     void getAllSumsTest(int n, int sum); // print
30     std::vector<std::vector<std::pair<int, int> > > case1(
      int n,int m,int p);// all possible transverse momentum
      combinations, serving as the transverse basis
31     int case2(int n, int m, int p);// (a1 + a2 + ... +
      an)^2 + (b1 + b2 + ... + bn)^2 <= m^2; |ai|, |bi| <= m
32     //n is # of bosons,p is # of pauli-vilars and m=nperp
33 };
34 #endif
```

```cpp
/* Sum2N.cpp
 */
#include "Sum2N.h"
#include <iostream>
#include <algorithm>
#include <stdexcept>
#include <cassert>
#include <cmath>
#include <map>
#include <set>
using namespace std;
using namespace Sum2N;
inline int square(int x) {
    return x * x;
}
//
http://stackoverflow.com/questions/9330915/number-of-combinat
ions-n-choose-r-in-c
int Sum2N::binomialCoefficient(int n, int k)
{
    if (k > n) return 0;
    if (k * 2 > n) k = n - k;
    if (k == 0) return 1;

    int result = n;
    for (int i = 2; i <= k; ++i) {
        result *= n - i + 1;
        result /= i;
    }
    return result;
}
double factor(int n )
{
    double product = 1;
    for (int i = 2; i <= n; i++)
        product *= i;
    return product;
}
void Sum2N::binomialCoefficientTest() {
    for (int n = 0; n <= 5; n++)
        for (int k = 0; k <= n; k++)
            assert (static_cast<int>(factor(n)/factor(k)/
            factor(n-k))
                    == binomialCoefficient(n, k));
}
//
http://stackoverflow.com/questions/9430568/generating-combina
```

```
    tions-in-c
44  vector<vector<bool> > Sum2N::generateCombinations(int n, int
     r)
45  {
46      vector<vector<bool> > combinations;
47      vector<bool> v(n);
48      fill(v.begin() + r, v.end(), true);
49
50      do {
51          vector<bool> c(n);
52          for (int i = 0; i < n; ++i)
53              c[i] = !v[i];
54          combinations.push_back(c);
55      } while (next_permutation(v.begin(), v.end()));
56
57      assert(static_cast<int>(combinations.size()) ==
        binomialCoefficient(n, r));
58
59      return combinations;
60  }
61  void Sum2N::generateCombinationsTest(int n, int r)
62  {
63      vector<vector<bool> > combinations =
        generateCombinations(n, r);
64      for (vector<vector<bool> >::const_iterator c =
        combinations.begin();
65              c != combinations.end(); ++c) {
66          assert(static_cast<int>(c->size()) == n);
67          for (int i = 0; i < n; i++)
68              if ((*c)[i])
69                  cout << i + 1;
70          cout << (c != combinations.end() - 1 ? ", " : "\n");
71      }
72  }
73  vector<vector<int> > Sum2N::getAllSums(int n, int sum) {
74      vector<vector<int> > vec;
75      if (sum < n || sum < 0 || n < 0) {
76          throw range_error("Invalid input in getAllSums");
77      }
78
79      vector<vector<bool> > combinations =
        generateCombinations(sum-1, n-1);
80      vector<int> a(n);
81      for (vector<vector<bool> >::const_iterator c =
        combinations.begin();
82           c != combinations.end(); ++c) {
83          for (int m = 0; m < n; m++)
```

```
84              a[m] = 0;
85          a[0] = 1;
86          int i = 0;
87          for (int sep = 0; sep < sum - 1; sep++)
88              if (!(*c)[sep])
89                  a[i]++;
90              else
91                  a[++i]++;
92
93          vec.push_back(a);
94      }
95
96      return vec;
97  }
98  void Sum2N::getAllSumsTest(int n, int sum)
99  {
100     vector<vector<int> > a = getAllSums(n, sum);
101     for (vector<vector<int> >::const_iterator it = a.begin
        (); it != a.end(); ++it) {
102         for (vector<int>::const_iterator it2 = it->begin();
            it2 != it->end(); ++it2)
103             cout << *it2;
104         cout << (it != a.end() - 1 ? ", " : "\n");
105     }
106
107 }
108 vector<vector<int> >
109 Sum2N::getAllSumBetween(int n, int loSum, int hiSum, int
    loBound, int hiBound)
110 {
111     vector<vector<int> > a;
112
113     int lo = max(loSum, loBound * n);
114     int hi = min(hiSum, hiBound * n);
115     if (lo > hi)
116         return a;
117
118     if (n == 1) {
119         for (int a1 = lo; a1 <= hi; a1++) {
120             vector<int> t(1, a1);
121             a.push_back(t);
122         }
123         return a;
124     }
125
126     for (int a1 = loBound; a1 <= hiBound; a1++) {
127         vector<vector<int> >
```

```
128             rest = getAllSumBetween(n-1, loSum - a1, hiSum - a1,
                 loBound, a1);
129
130             for (vector<vector<int> >::iterator it = rest.begin
                ();
131                     it != rest.end(); ++it) {
132                 it->push_back(a1);
133                 a.push_back(*it);
134             }
135         }
136
137         return a;
138     }
139     vector<vector<int> >
140     Sum2N::getAllSquaredSumBetween(int n, int loSum, int hiSum,
        int loBound, int hiBound)
141     {
142         vector<vector<int> > a;
143
144         if (hiSum < 0)
145             return a;
146
147         if (loSum < 0)
148             loSum = 0;
149
150         if (n == 1) {
151             for (int a1 = loBound; a1 <= hiBound; a1++) {
152                 if (loSum <= square(a1) && square(a1) <= hiSum) {
153                     vector<int> t(1, a1);
154                     a.push_back(t);
155                 }
156             }
157             return a;
158         }
159
160         for (int a1 = loBound; a1 <= hiBound; a1++) {
161             vector<vector<int> >
162             rest = getAllSquaredSumBetween(n-1, loSum - a1*a1,
                hiSum - a1*a1, loBound, a1);
163
164             for (vector<vector<int> >::iterator it = rest.begin
                ();
165                     it != rest.end(); ++it) {
166                 it->push_back(a1);
167                 a.push_back(*it);
168             }
169         }
```

```
170
171         return a;
172     }
173     void print(const set<vector<pair<int, int> > >& x, const
        char* name)
174     {
175         cout << name << x.size() << endl;
176         for (set<vector<pair<int, int> > >::const_iterator it =
            x.begin(); it != x.end(); ++it) {
177             for (vector<pair<int, int> >::const_iterator it2 =
                it->begin(); it2 != it->end(); ++it2)
178                 cout << "(" << it2->first << ", " << it2->second
                     << ") ";
179             cout << endl;
180         }
181         cout << endl;
182     }
183     // make pairs (a1, b1), (a2, b2), ..., (an, bn)
184     set<vector<pair<int, int> > > makePairs(const vector<vector<
        int> >& a,
185             const vector<vector<int> >& b) {
186         set<vector<pair<int, int> > > c;
187         for (vector<vector<int> >::const_iterator it = a.begin
            (); it != a.end(); ++it) {
188             for (vector<vector<int> >::const_iterator it2 = b.
                begin(); it2 != b.end(); ++it2) {
189                 vector<pair<int, int> > t;
190                 for (size_t i = 0; i < it->size(); i++)
191                     t.push_back(make_pair((*it)[i], (*it2)[i]));
192                 sort(t.begin(), t.end());
193                 c.insert(t); // cannot use vector, e.g. (0 0)
                    (0 1) | (0 1) (0 0)
194             }
195         }
196         return c;
197     }
198     // choose p from all permutations of (a1, b1), (a2, b2),
        ..., (an, bn)
199     vector<vector<pair<int, int> > > chooseFromPermutations
200     (const set<vector<pair<int, int> > >& c, const int p) {
201         vector<vector<pair<int, int> > > d;
202         int n = 0;
203         for (set<vector<pair<int, int> > >::const_iterator it =
            c.begin(); it != c.end(); ++it) {
204             if (it == c.begin())
205                 n = static_cast<int>(it->size());
206
```

```
207             vector<pair<int, int> > t = *it;
208
209             set<vector<pair<int, int> > > uniqueChoices;
210             vector<vector<bool> > combinations =
                generateCombinations(n, p);
211             for (vector<vector<bool> >::const_iterator c =
                combinations.begin();
212                     c != combinations.end(); ++c) {
213                 vector<pair<int, int> > ps;
214                 vector<pair<int, int> > vs;
215                 for (int i = 0; i < n; i++)
216                     if ((*c)[i])
217                         ps.push_back(t[i]);
218                     else
219                         vs.push_back(t[i]);
220                 sort(ps.begin(), ps.end());
221                 sort(vs.begin(), vs.end());
222                 ps.insert(ps.end(), vs.begin(), vs.end());
223                 uniqueChoices.insert(ps);
224             }
225
226             d.insert(d.end(), uniqueChoices.begin(),
                uniqueChoices.end());
227         }
228         return d;
229     }
230     vector<vector<int> > allPermutations(const vector<vector<int
    > >& a) {
231         vector<vector<int> > b; // all permutations of a
232         for (vector<vector<int> >::const_iterator it = a.begin
        (); it != a.end(); ++it) {
233             vector<int> t = *it;
234             do {
235                 b.push_back(t);
236             } while (next_permutation(t.begin(), t.end()));
237         }
238         return b;
239     }
240
241     int Sum2N::case2(int n, int m, int p)
242     {
243         set<vector<pair<int, int> > > c;
244
245         for (int x = -m/2; x <= m/2; x++) {
246             vector<vector<int> > ta = getAllSumBetween(n, x, x,
                -1, 1);
247             int ymax = sqrt(static_cast<double>(0.25*m*m - x*x));
```

```
248             vector<vector<int> > tb = allPermutations(
                getAllSumBetween(n,
249                     -ymax, ymax, -1, 1));
250             set<vector<pair<int, int> > > tc = makePairs(ta, tb);
251             c.insert(tc.begin(), tc.end());
252         }
253         vector<vector<pair<int, int> > > d =
                chooseFromPermutations(c, p);
254         return d.size();
255
256     }
257     vector<vector<pair<int,int> > > Sum2N::case1(int n, int m,
        int p)
258     {set<vector<pair<int, int> > > c;
259
260         for (int x = -m/2; x <= m/2; x++) {
261             vector<vector<int> > ta = getAllSumBetween(n, x, x,
                -1, 1);
262             int ymax = sqrt(static_cast<double>(0.25*m*m - x*x));
263             vector<vector<int> > tb = allPermutations(
                getAllSumBetween(n,
264                     -ymax, ymax, -1, 1));
265             set<vector<pair<int, int> > > tc = makePairs(ta, tb);
266             c.insert(tc.begin(), tc.end());
267         }
268         vector<vector<pair<int, int> > > d =
                chooseFromPermutations(c, p);
269         return d;
270     }
271
```

A.2 $H|\psi\rangle$

A.2.1 Hamiltonian.h

```
#ifndef HAMILTONIAN_H
#define HAMILTONIAN_H
#include <iostream>
#include <vector>
#include <stdexcept>
#include <cmath>
#include <lapackpp.h>
#include "parameter.h"
#include "cover.h"
#include "basis.h"
using namespace std;
using namespace Sum2N;
// A:To store the diagonal matrices
//BL:To store the Longitudinal parts of off-diagonal Hamiltonian B
//BT:To store the Transverse parts of off-diagonal Hamiltonian B
//DL:To store the Longitudinal parts of off-diagonal Hamiltonian D
//DT:To store the Transverse parts of off-diagonal Hamiltonian D
void Hamiltonian(vector<vector<LaVectorDouble> > &A,vector<LaGenMatDouble > &BL, vector<LaGenMatDouble > &BT,vector<LaGenMatDouble > &DL,
vector<LaGenMatDouble > &DT,vector<vector<vector<int> > > &b,vector<vector<vector<pair<int,int> > > > &d,vector<vector<int> > &s)
{
  basis(b,d,s);//generate basis sets,both the longitudinal basis and the transverse basis
//---------Generate diagonal matrix A---
   for(size_t i=0;i<d.size();i++)
 {  vector<LaVectorDouble> A1(5);
    int v=b[i].size();
    int u=d[i].size();
    LaVectorDouble A10(v);//submatrix a
    LaVectorDouble A11(v);//submatrix b
    LaVectorDouble A12(v);//submatrix c
    LaVectorDouble A13(u);//submatrix d
    LaVectorDouble A14(u);//submatrix e

    for(int j=0;j<v;j++)
    {  double n=K;double Kmi=0; double Klj=0; //necessary variables for submatrix a,b,c
       for(int k=0;k<b[i][j].size();k++)
          n-=b[i][j][k];
       A10(j)=M0_prime*n/K;
```

```
38          for(int m=0;m<s[i][0];m++)
39             Kmi+=K/b[i][j][m];
40          A11(j)=Kmi;
41
42          for(int l=s[i][0];l<b[i][j].size();l++)
43             Klj+=K/b[i][j][l];
44          A12(j)=Klj;
45       }//the above is the definition for Transverse Hamiltonian
46
47       for(int j1=0;j1<u;j1++)
48       {  double trp=0; double trv=0;//necessary variables for
         submatrix d and e.
49          for(int k=0;k<s[i][0];k++)
50             trp+=1+(pow(d[i][j1][k].first,2.0)+pow(d[i][j1][k
               ].second,2.0))/pow(L_perp,2.0);
51          A13(j1)=trp;
52          for(int l=s[i][0];l<d[i][j1].size();l++)
53             trv+=mu1+(pow(d[i][j1][l].first,2.0)+pow(d[i][j1][
               l].second,2.0))/pow(L_perp,2.0);
54          A14(j1)=trv;
55        }
56        A1[0]=A10; A1[1]=A11;A1[2]=A12;A1[3]=A13;A1[4]=A14;
57       //cout<<A10(0)<<endl;
58       A.push_back(A1);
59    }
60   //----Generate matrix B(BL,BT)----------
61   //NEED TO CHECK pv's LONGITUDINAL MOMENTUM at s[i] and
     s[i+I] ARE THE SAME.
62     for(size_t i=0;i<d.size()-(K+1)/2;i++)
63   {
64     int I=s[i][0]+s[i][1]+2;// B interaction happens between
       s[i] and s[i+I];
65     if (s[i+I][0]!=s[i][0]+1||s[i+I][1]!=s[i][1]) {
66           throw range_error("Incorrect location for pb
             interaction");
67     }
68     int x1=b[i].size();int y1=b[i+I].size();int x2=d[i+I].size
       ();int y2=d[i].size();
69     LaGenMatDouble BL1(x1,y1); LaGenMatDouble BT1(x2,y2);
70     for(int j=0;j<x1;j++)
71     {
72       for(int k=0;k<y1;k++)
73       { BL1(j,k)=0;
74         bool are_equal=true;
75         if(s[i][1]!=0)
76         {
77         vector<int> pv(b[i+I][k].end()-s[i+I][1],b[i+I][k].end
```

```
        ());
78      vector<int> pv1(b[i][j].end()-s[i][1],b[i][j].end());
79      are_equal=cover1(pv,pv1);
80      }
81      if(are_equal)
82     {
83      vector<int> c(b[i+I][k].begin(),b[i+I][k].begin()+s[i+
        I][0]);
84      vector<int> cc(b[i][j].begin(),b[i][j].begin()+s[i][0
        ]);
85      bool answer=cover1(c,cc);
86      if(answer)
87       {
88         double n=K;
89         for(size_t l=0;l<b[i][j].size();l++)
90         n-=b[i][j][l];
91         double nm=K;
92         for(size_t m=0;m<b[i+I][k].size();m++)
93         nm-=b[i+I][k][m];
94         BL1(j,k)=coeff*sqrt(nm/(n*(n-nm)));
95       }
96     }
97    }
98
99   }
100  for(int j=0;j<x2;j++)
101  {
102    for(int k=0;k<y2;k++)
103    { BT1(j,k)=0;
104      bool are_equal=true;
105      if(s[i][1]!=0)
106      {
107      vector<pair<int,int> > pv(d[i+I][j].end()-s[i+I][1],d[
        i+I][j].end());
108      vector<pair<int,int> > pv1(d[i][k].end()-s[i][1],d[i][
        k].end());
109      are_equal=cover(pv,pv1);
110      }
111      if(are_equal)
112      {
113      vector<pair<int,int> > c(d[i+I][j].begin(),d[i+I][j].
        begin()+s[i+I][0]);
114      vector<pair<int,int> > cc(d[i][k].begin(),d[i][k].
        begin()+s[i][0]);
115      bool answer=cover(c,cc);
116      if(answer) BT1(j,k)=1;
117      }
```

```
118       }
119     }
120     BL.push_back(BL1);BT.push_back(BT1);
121   }
122   //------Define D(DL,DT)--------
123     for(size_t i=0;i<d.size()-(K+1)/2;i++)
124   {
125     int I=s[i][0]+s[i][1]+1;// D interaction happens between
        s[i] and s[i+I];
126     if (s[i+I][1]!=s[i][1]+1||s[i+I][0]!=s[i][0]){
127         throw range_error("Incorrect location for pv
            interaction");
128       }
129     int x1=b[i].size(); int y1=b[i+I].size(); int x2=d[i+I].
        size();int y2=d[i].size();
130     LaGenMatDouble DL1(x1,y1);LaGenMatDouble DT1(x2,y2);
131     for(int j=0;j<x1;j++)
132     {
133       for(int k=0;k<y1;k++)
134       { DL1(j,k)=0;
135         bool are_equal=true;
136         if(s[i][0]!=0)
137         {
138         vector<int> pb(b[i+I][k].begin(),b[i+I][k].begin()+s[i
            +I][0]);
139         vector<int> pb1(b[i][j].begin(),b[i][j].begin()+s[i][0
            ]);
140         are_equal=cover1(pb,pb1);
141         }
142         if(are_equal)
143         {
144           bool answer=true;
145           if(s[i][1]!=0)
146           {
147            vector<int> a(b[i+I][k].end()-s[i+I][1],b[i+I][k].
               end());
148            vector<int> aa(b[i][j].end()-s[i][1],b[i][j].end());
149            answer=cover1(a,aa);
150           }
151           if(answer)
152           {
153             double L=K;
154             for(size_t l=0;l<b[i][j].size();l++)
155             L-=b[i][j][l];
156             double NL=K;
157             for(size_t m=0;m<b[i+I][k].size();m++)
158             NL-=b[i+I][k][m];
```

```
159             DL1(j,k)=coeff*sqrt(NL/(L*(L-NL)));
160           }
161         }
162       }
163     }
164     for(int j=0;j<x2;j++)
165     {
166       for(int k=0;k<y2;k++)
167       { DT1(j,k)=0;
168         bool are_equal=true;
169         if(s[i][0]!=0)
170         {
171         vector<pair<int,int> > pb(d[i+I][j].begin(),d[i+I][j].
          begin()+s[i+I][0]);
172         vector<pair<int,int> > pb1(d[i][k].begin(),d[i][k].
          begin()+s[i][0]);
173         are_equal=cover(pb,pb1);
174         }
175         if(are_equal)
176         {
177         bool answer=true;
178         if(s[i][1]!=0)
179         {
180         vector<pair<int,int> > a(d[i+I][j].end()-s[i+I][1],d[i
          +I][j].end());
181         vector<pair<int,int> > aa(d[i][k].end()-s[i][1],d[i][k
          ].end());
182         answer=cover(a,aa);
183         }
184         if(answer) DT1(j,k)=1;
185         }
186       }
187     }
188     DL.push_back(DL1);DT.push_back(DT1);
189   }
190   }
191   #endif
192
```

A.2.2 cover.h

```
1   #ifndef COVER_H
2   #define COVER_H
3   #include <map>
4   #include <vector>
5   using std::vector;
6   using namespace std;
7   bool cover(vector<pair<int,int> > &a,vector<pair<int,int> >
    &b)
8   {
9     map<pair<int,int>,int> acount;
10    for(vector<pair<int,int> >::const_iterator it=a.begin();it
      !=a.end();++it)
11    acount[*it]++;
12    bool inc=true;
13    for(vector<pair<int,int> >::const_iterator it=b.begin();it
      !=b.end();++it)
14    {
15       if(acount[*it]==0)
16       { inc=false;
17         break;
18       }
19       acount[*it]--;
20    }
21    return inc;
22  }
23  bool cover1(vector<int > &a,vector<int > &b)
24  {
25    map<int,int> acount;
26    for(vector<int >::const_iterator it=a.begin();it!=a.end
      ();++it)
27    acount[*it]++;
28    bool inc=true;
29    for(vector<int >::const_iterator it=b.begin();it!=b.end
      ();++it)
30    {
31       if(acount[*it]==0)
32       { inc=false;
33         break;
34       }
35       acount[*it]--;
36    }
37    return inc;
38  }
39  #endif
40
```

A.2.3 HDotU.h

```
#ifndef HDOTU_H
#define HDOTU_H
#define LA_COMPLEX_SUPPORT
#include <iostream>
#include <stdexcept>
#include <vector>
#include <lapack++.h>
#include <math.h>
#include <complex>
#include "basis.h"
#include "parameter.h"
#include "Hamiltonian.h"
#include "Compression.h"
using namespace std;
using namespace Sum2N;
void HDotU(vector<vector<LaVectorDouble> > &A,vector<LaGenMatDouble > &BL, vector<LaGenMatDouble > &BT,vector<LaGenMatDouble > &DL, vector<LaGenMatDouble > &DT, vector<vector<vector<int> > > &b,vector<vector<vector<pair<int,int> > > > &d,vector<vector<int> > &s,vector<LaGenMatComplex> &tv,vector<LaGenMatComplex> &lv, vector<LaVectorDouble> &Sigma,vector<LaGenMatComplex> &TV, vector<LaGenMatComplex> &LV,vector<LaVectorDouble> &Sigma1)
{ //------calculate \psi'^(0,0)----------
    complex<double> tv0(tv[0](0,0).r*M0_prime,tv[0](0,0).i*M0_prime);
    complex<double> a(lv[0](0,0).r,lv[0](0,0).i);
    a=a*tv0;
    for(int k=0;k<2;k++){
    LaGenMatComplex M(d[k].size(),Sigma[k+1].size());
    M=tv[k+1];
      for(int j=0;j<M.size(1);j++)
     {COMPLEX comp;comp.r=Sigma[k+1](j);
      M(LaIndex(0,M.size(0)-1),j)*=comp;
     }
    LaGenMatComplex C(d[k].size(),b[k].size());
    Blas_Mat_Mat_Trans_Mult(M,lv[k+1], C);//Calculate \psi^(0,1)and \psi^(1,0), which equal to tv[k+1]*Sigma[k+1]*lv[k+1]'
    for (int i=0;i<b[k].size();i++)
    {  double bki=b[k][i][0];
       complex<double> x=coeff*sqrt((K-bki)/(K*bki));
       if(k==0){complex<double> COM(0,1);x*=COM;}
       for(int j=0;j<d[k].size();j++)
          { complex<double> a1(C(j,i).r,C(j,i).i);
            a+=x*a1;}
    }// calculate D*\psi^(0,1),B*\psi^(1,0) and add them to
```

```
      \psi'^(0,0)
40    }
41    LaGenMatComplex TV0(1,1);TV0(0,0).r=a.real();TV0(0,0).i=
      a.imag();
42    //cout<<TV0(0,0)<<endl;
43    TV[0]=TV0;
44    LaGenMatComplex t1(1,1);
45    t1(0,0).r=1.0;
46    LV[0]=t1;
47    LaVectorDouble vd(1);vd(0)=1.0;
48    Sigma1[0]=vd;
49    //calculate the SVD form of \psi'{(n,n1)}
50    //How to deal with psi^{(0,1)} or psi^(1,0) interact
      with psi^(0,0);
51    vector<LaGenMatComplex> lv1(lv.size());
52      for(int i=0;i<lv1.size();i++)
53    { lv1[i]=lv[i];
54      for(int j=0;j<lv1[i].size(1);j++)
55     {COMPLEX comp;comp.r=Sigma[i](j);comp.i=0;
56      lv1[i](LaIndex(0,lv1[i].size(0)-1),j)*=comp;
57     }
58    }
59
60    for(size_t i=1;i<s.size()+1;i++)
61    {
62      if (s[i-1][0]!=0&s[i-1][1]!=0&s[i-1][0]+s[i-1][1]!=(K-
      1)/2)//case 1
63      { int ds=d[i-1].size(); int tvs=tv[i].size(1);
64        int I=s[i-1][0]+s[i-1][1]+2;//B interaction happens
        between tv[i] and tv[i+I]
65        int tb=tv[i+I].size(1);
66        int td=tv[i+I-1].size(1);//D interaction happens
        between tv[i] and tv[i+I-1]
67        int tc=tv[i-I+1].size(1);//C interaction happens
        between tv[i] and tv[i-I+1]
68        int te=tv[i-I+2].size(1);//E interaction happens
        between tv[i] and tv[i-I+2]
69        LaGenMatComplex tti(ds,tvs*3+tb+td+tc+te);
70        //I*\psi
71        tti(LaIndex(0,ds-1),LaIndex(0,tvs-1)).inject(tv[i]);
72        //A13*\psi
73        LaGenMatComplex a10(ds,tvs);a10=tv[i];
74        for(int j=0;j<ds;j++)
75        {COMPLEX a0;a0.r=A[i-1][3](j);
76        a10(j,LaIndex(0,tvs-1))*=a0;}
77        tti(LaIndex(0,ds-1),LaIndex(tvs,2*tvs-1)).inject(a10
          );
```

```
78          //A14*\psi
79          a10=tv[i];
80          for(int j=0;j<ds;j++)
81          {COMPLEX a0;a0.r=A[i-1][4](j);
82          a10(j,LaIndex(0,tvs-1))*=a0;}
83          tti(LaIndex(0,ds-1),LaIndex(2*tvs,3*tvs-1)).inject(
            a10);
84          //BT1'\psi(n+1,n1) B interaction
85          LaGenMatComplex a20(ds,tb);
86          Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BT[i-1]),tv[
            i+I],a20);
87          tti(LaIndex(0,ds-1),LaIndex(3*tvs,3*tvs+tb-1)).
            inject(a20);
88          //DT1'\psi(n,n1+1) D interaction
89          LaGenMatComplex a30(ds,td);
90          Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(DT[i-1]),tv[
            i+I-1],a30);
91          COMPLEX d0;d0.r=0;d0.i=1;
92          a30*=d0;
93          tti(LaIndex(0,ds-1),LaIndex(3*tvs+tb,3*tvs+tb+td-1
            )).inject(a30);
94          //C interaction happens between tv[i] with tv[i-I+1];
95          LaGenMatComplex a40(ds,tc);
96          Blas_Mat_Mat_Mult(LaGenMatComplex(BT[i-1-I+1]),tv[i-
            I+1],a40);
97          tti(LaIndex(0,ds-1),LaIndex(3*tvs+tb+td,3*tvs+tb+td+
            tc-1)).inject(a40);
98          //E interaction happens between tv[i] with tv[i-I+2];
99          LaGenMatComplex a50(ds,te);
100         Blas_Mat_Mat_Mult(LaGenMatComplex(DT[i-1-I+2]),tv[i-
            I+2],a50);
101         a50*=d0;
102         tti(LaIndex(0,ds-1),LaIndex(3*tvs+tb+td+tc,3*tvs+tb+
            td+tc+te-1)).inject(a50);
103 //Transverse vectors has been generated above, the
    following gives Longitudinal vectors
104         int bs=b[i-1].size();
105         LaGenMatComplex lvi(bs,tvs*3+tb+td+tc+te);
106         //A10*\psi
107         LaGenMatComplex b10(bs,tvs);b10=lv1[i];
108         for(int j=0;j<bs;j++)
109         { COMPLEX b0;b0.r=A[i-1][0](j);
110           b10(j,LaIndex(0,tvs-1))*=b0;}
111           lvi(LaIndex(0,bs-1),LaIndex(0,tvs-1)).inject(b10);
112         //A11*\psi
113         b10=lv1[i];
114         for(int j=0;j<bs;j++)
```

```
115           { COMPLEX b0;b0.r=A[i-1][1](j);
116             b10(j,LaIndex(0,tvs-1))*=b0;}
117             lvi(LaIndex(0,bs-1),LaIndex(tvs,2*tvs-1)).inject(
                b10);
118           //A12*\psi
119           b10=lv1[i];
120           for(int j=0;j<bs;j++)
121           { COMPLEX b0;b0.r=A[i-1][2](j);
122             b10(j,LaIndex(0,tvs-1))*=b0;}
123             lvi(LaIndex(0,bs-1),LaIndex(2*tvs,3*tvs-1)).inject
                (b10);
124         //BL1'\psi(n+1,n1) B interaction
125         LaGenMatComplex b20(bs,tb);
126         Blas_Mat_Mat_Mult(LaGenMatComplex(BL[i-1]),lv1[i+I],
            b20);
127         lvi(LaIndex(0,bs-1),LaIndex(3*tvs,3*tvs+tb-1)).inject
            (b20);
128         //BL1'\psi(n,n1+1) D interaction
129         LaGenMatComplex b30(bs,td);
130         Blas_Mat_Mat_Mult(LaGenMatComplex(DL[i-1]),lv1[i+I-1
            ],b30);
131         lvi(LaIndex(0,bs-1),LaIndex(3*tvs+tb,3*tvs+tb+td-1)).
            inject(b30);
132         //C interaction happens between lv1[i] with lv1[i-I+1]
133         LaGenMatComplex b40(bs,tc);
134         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BL[i-1-I+1]),
            lv1[i-I+1],b40);
135         lvi(LaIndex(0,bs-1),LaIndex(3*tvs+tb+td,3*tvs+tb+td+
            tc-1)).inject(b40);
136         //E interaction happens between lv1[i] with
            lv1[i-I+2];
137         LaGenMatComplex b50(bs,te);
138         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(DL[i-1-I+2]),
            lv1[i-I+2],b50);
139         lvi(LaIndex(0,bs-1),LaIndex(3*tvs+tb+td+tc,3*tvs+tb+
            td+tc+te-1)).inject(b50);
140         //-------------compression----
141         int m=min(n_svd,bs);
142         LaGenMatComplex TVi(tti.size(0),m);
143         LaGenMatComplex LVi(lvi.size(0),m);
144         LaVectorDouble Lamda(m);
145       Compression(tti, lvi, TVi,LVi, Lamda);
146       TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;
147       }
148       else if (s[i-1][0]==0&s[i-1][1]!=0&s[i-1][1]!=(K-1)/2)
          //case 2
149       { int ds=d[i-1].size(); int tvs=tv[i].size(1);
```

```
150         int I=s[i-1][0]+s[i-1][1]+2;//B interaction happens
            between tv[i] and tv[i+I]
151         int tb=tv[i+I].size(1); // diagonal part
152         int td=tv[i+I-1].size(1);//D interaction happens
            between tv[i] and tv[i+I-1]
153         int te=tv[i-I+2].size(1);//E interaction happens
            between tv[i] and tv[i-I+2]
154         LaGenMatComplex tti(ds,tvs*2+tb+td+te);
155         tti(LaIndex(0,ds-1),LaIndex(0,tvs-1)).inject(tv[i]);
156         //A14*\psi
157         LaGenMatComplex a10(ds,tvs);
158         a10=tv[i];
159         for(int j=0;j<ds;j++)
160         {COMPLEX a0;a0.r=A[i-1][4](j);
161         a10(j,LaIndex(0,tvs-1))*=a0;}
162         tti(LaIndex(0,ds-1),LaIndex(tvs,2*tvs-1)).inject(a10
            );
163         //BT1'\psi(n+1,n1) B interaction
164         LaGenMatComplex a20(ds,tb);
165         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BT[i-1]),tv[
            i+I],a20);
166         tti(LaIndex(0,ds-1),LaIndex(2*tvs,2*tvs+tb-1)).
            inject(a20);
167         //DT1'\psi(n,n1+1) D interaction
168         LaGenMatComplex a30(ds,td);
169         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(DT[i-1]),tv[
            i+I-1],a30);
170         COMPLEX d0;d0.r=0;d0.i=1;
171         a30*=d0;
172         tti(LaIndex(0,ds-1),LaIndex(2*tvs+tb,2*tvs+tb+td-1
            )).inject(a30);
173         //E interaction happens between tv[i] with tv[i-I+2];
174         LaGenMatComplex a50(ds,te);
175         if(i!=1){
176         Blas_Mat_Mat_Mult(LaGenMatComplex(DT[i-1-I+2]),tv[i-
            I+2],a50);
177         a50*=d0;}
178         else {a50=d0;a50(LaIndex(0,ds-1),LaIndex(0,te-1))*=
            tv[0](0,0);}//E interaction with \psi^{(0,0)}
179         tti(LaIndex(0,ds-1),LaIndex(2*tvs+tb+td,2*tvs+tb+td+
            te-1)).inject(a50);
180         int bs=b[i-1].size();
181         LaGenMatComplex lvi(bs,tvs*2+tb+td+te);
182         //A10*\psi
183         LaGenMatComplex b10(bs,tvs);b10=lv1[i];
184         for(int j=0;j<bs;j++)
185         { COMPLEX b0;b0.r=A[i-1][0](j);
```

```
186             b10(j,LaIndex(0,tvs-1))*=b0;}
187             lvi(LaIndex(0,bs-1),LaIndex(0,tvs-1)).inject(b10);
188           //A12*\psi
189           b10=lv1[i];
190           for(int j=0;j<bs;j++)
191           { COMPLEX b0;b0.r=A[i-1][2](j);
192             b10(j,LaIndex(0,tvs-1))*=b0;}
193             lvi(LaIndex(0,bs-1),LaIndex(tvs,2*tvs-1)).inject(
              b10);
194         //BL1'\psi(n+1,n1) B interaction
195         LaGenMatComplex b20(bs,tb);
196         Blas_Mat_Mat_Mult(LaGenMatComplex(BL[i-1]),lv1[i+I],
            b20);
197         lvi(LaIndex(0,bs-1),LaIndex(2*tvs,2*tvs+tb-1)).inject
            (b20);
198         //BL1'\psi(n,n1+1) D interaction
199         LaGenMatComplex b30(bs,td);
200         Blas_Mat_Mat_Mult(LaGenMatComplex(DL[i-1]),lv1[i+I-1
            ],b30);
201         lvi(LaIndex(0,bs-1),LaIndex(2*tvs+tb,2*tvs+tb+td-1)).
            inject(b30);
202         //E interaction happens between lv1[i] with
            lv1[i-I+2];
203         LaGenMatComplex b50(bs,te);
204          if(i!=1){
205          Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(DL[i-1-I+2
             ]),lv1[i-I+2],b50);}
206          else {
207          for(int j=0;j<bs;j++)
208          { double bki=b[i-1][j][0];
209            double x=coeff*sqrt((K-bki)/(K*bki));
210            COMPLEX X;X.r=x;
211          b50(j,0)=X;}}//E interaction with
             \psi^{(0,0)},lv[0](0,0)=1,so there's no need to
             multiply lv1[0].
212          lvi(LaIndex(0,bs-1),LaIndex(2*tvs+tb+td,2*tvs+tb+td+
             te-1)).inject(b50);
213         //-------------compression----------------
214         int m=min(n_svd,bs);
215         LaGenMatComplex TVi(tti.size(0),m);
216         LaGenMatComplex LVi(lvi.size(0),m);
217         LaVectorDouble Lamda(m);
218       Compression(tti,lvi, TVi,LVi, Lamda);
219       TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;
220       }
221       else if (s[i-1][1]==0&s[i-1][0]!=0&s[i-1][0]!=(K-1)/2)
          //case 3
```

```
222         {
223           int ds=d[i-1].size(); int tvs=tv[i].size(1);
224           int I=s[i-1][0]+s[i-1][1]+2;//B interaction happens
              between tv[i] and tv[i+I]
225           int tb=tv[i+I].size(1);
226           int td=tv[i+I-1].size(1);//D interaction happens
              between tv[i] and tv[i+I-1]
227           int tc=tv[i-I+1].size(1);//C interaction happens
              between tv[i] and tv[i-I+1]
228           LaGenMatComplex tti(ds,tvs*2+tb+td+tc);
229           tti(LaIndex(0,ds-1),LaIndex(0,tvs-1)).inject(tv[i]);
230           //A13*\psi
231           LaGenMatComplex a10(ds,tvs);a10=tv[i];
232           for(int j=0;j<ds;j++)
233           {COMPLEX a0;a0.r=A[i-1][3](j);
234           a10(j,LaIndex(0,tvs-1))*=a0;}
235           tti(LaIndex(0,ds-1),LaIndex(tvs,2*tvs-1)).inject(a10
              );
236           //BT1'\psi(n+1,n1) B interaction
237           LaGenMatComplex a20(ds,tb);
238           Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BT[i-1]),tv[
              i+I],a20);
239           tti(LaIndex(0,ds-1),LaIndex(2*tvs,2*tvs+tb-1)).
              inject(a20);
240           //DT1'\psi(n,n1+1) D interaction
241           LaGenMatComplex a30(ds,td);
242           Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(DT[i-1]),tv[
              i+I-1],a30);
243           COMPLEX d0;d0.r=0;d0.i=1;
244           a30*=d0;
245           tti(LaIndex(0,ds-1),LaIndex(2*tvs+tb,2*tvs+tb+td-1
              )).inject(a30);
246           //C interaction happens between tv[i] with tv[i-I+1];
247           LaGenMatComplex a40(ds,tc);
248           if(i!=2){
249           Blas_Mat_Mat_Mult(LaGenMatComplex(BT[i-1-I+1]),tv[i-
              I+1],a40);}
250           else {a40=1;a40(LaIndex(0,ds-1),LaIndex(0,tc-1))*=tv
              [0](0,0);}
251           tti(LaIndex(0,ds-1),LaIndex(2*tvs+tb+td,2*tvs+tb+td+
              tc-1)).inject(a40);
252           int bs=b[i-1].size();
253           LaGenMatComplex lvi(bs,tvs*2+tb+td+tc);
254           //A10*\psi
255           LaGenMatComplex b10(bs,tvs);b10=lv1[i];
256           for(int j=0;j<bs;j++)
257           { COMPLEX b0;b0.r=A[i-1][0](j);
```

```
258             b10(j,LaIndex(0,tvs-1))*=b0;}
259             lvi(LaIndex(0,bs-1),LaIndex(0,tvs-1)).inject(b10);
260           //A11*\psi
261           b10=lv1[i];
262           for(int j=0;j<bs;j++)
263           { COMPLEX b0;b0.r=A[i-1][1](j);
264             b10(j,LaIndex(0,tvs-1))*=b0;}
265             lvi(LaIndex(0,bs-1),LaIndex(tvs,2*tvs-1)).inject(
              b10);
266         //BL1'\psi(n+1,n1) B interaction
267         LaGenMatComplex b20(bs,tb);
268         Blas_Mat_Mat_Mult(LaGenMatComplex(BL[i-1]),lv1[i+I],
            b20);
269         lvi(LaIndex(0,bs-1),LaIndex(2*tvs,2*tvs+tb-1)).inject
            (b20);
270         //BL1'\psi(n,n1+1) D interaction
271         LaGenMatComplex b30(bs,td);
272         Blas_Mat_Mat_Mult(LaGenMatComplex(DL[i-1]),lv1[i+I-1
            ],b30);
273         lvi(LaIndex(0,bs-1),LaIndex(2*tvs+tb,2*tvs+tb+td-1)).
            inject(b30);
274         //C interaction happens between lv1[i] with lv1[i-I+1]
275         LaGenMatComplex b40(bs,tc);
276         if(i!=2){
277         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BL[i-1-I+1]),
            lv1[i-I+1],b40);}
278         else {
279          for(int j=0;j<bs;j++)
280          { double bki=b[i-1][j][0];
281            double x=coeff*sqrt((K-bki)/(K*bki));
282            COMPLEX X;X.r=x;
283          b40(j,0)=X;}}//C interaction with
            \psi^{(0,0)},lv[0](0,0)=1,so there's no need to
            multiply lv1[0].
284          lvi(LaIndex(0,bs-1),LaIndex(2*tvs+tb+td,2*tvs+tb+td+
            tc-1)).inject(b40);
285         //-------compression----
286         int m=min(n_svd,bs);
287         LaGenMatComplex TVi(tti.size(0),m);
288         LaGenMatComplex LVi(lvi.size(0),m);
289         LaVectorDouble Lamda(m);
290       Compression(tti, lvi, TVi,LVi, Lamda);
291       TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;
292       }
293       else if (s[i-1][0]!=0&s[i-1][1]!=0&s[i-1][0]+s[i-1][1
          ]==(K-1)/2)//case 4
294       { int ds=d[i-1].size(); int tvs=tv[i].size(1);
```

```
295         int I=s[i-1][0]+s[i-1][1]+2;//there is no B
            interaction
296         int tc=tv[i-I+1].size(1);//C interaction happens
            between tv[i] and tv[i-I+1]
297         int te=tv[i-I+2].size(1);//E interaction happens
            between tv[i] and tv[i-I+2]
298         LaGenMatComplex tti(ds,tvs*3+tc+te);
299         //I*\psi
300         tti(LaIndex(0,ds-1),0).inject(tv[i]);
301         //A13*\psi
302         LaGenMatComplex a10(ds,tvs);a10=tv[i];
303         for(int j=0;j<ds;j++)
304         {COMPLEX a0;a0.r=A[i-1][3](j);
305         a10(j,LaIndex(0,tvs-1))*=a0;}
306         tti(LaIndex(0,ds-1),1).inject(a10);
307         //A14*\psi
308         a10=tv[i];
309         for(int j=0;j<ds;j++)
310         {COMPLEX a0;a0.r=A[i-1][4](j);
311         a10(j,LaIndex(0,tvs-1))*=a0;}
312         tti(LaIndex(0,ds-1),2).inject(a10);
313         //C interaction happens between tv[i] with tv[i-I+1];
314         LaGenMatComplex a40(ds,tc);
315         Blas_Mat_Mat_Mult(LaGenMatComplex(BT[i-1-I+1]),tv[i-
            I+1],a40);
316         tti(LaIndex(0,ds-1),LaIndex(3*tvs,3*tvs+tc-1)).
            inject(a40);
317         //E interaction happens between tv[i] with tv[i-I+2];
318         LaGenMatComplex a50(ds,te);
319         Blas_Mat_Mat_Mult(LaGenMatComplex(DT[i-1-I+2]),tv[i-
            I+2],a50);
320         COMPLEX d0;d0.r=0;d0.i=1;
321         a50*=d0;
322         tti(LaIndex(0,ds-1),LaIndex(3*tvs+tc,3*tvs+tc+te-1
            )).inject(a50);
323         int bs=b[i-1].size();
324         LaGenMatComplex lvi(bs,tvs*3+tc+te);
325         //A10*\psi
326         LaGenMatComplex b10(bs,tvs);b10=lv1[i];
327         for(int j=0;j<bs;j++)
328         { COMPLEX b0;b0.r=A[i-1][0](j);
329           b10(j,LaIndex(0,tvs-1))*=b0;}
330           lvi(LaIndex(0,bs-1),0).inject(b10);
331         //A11*\psi
332         b10=lv1[i];
333         for(int j=0;j<bs;j++)
334         { COMPLEX b0;b0.r=A[i-1][1](j);
```

```
335             b10(j,LaIndex(0,tvs-1))*=b0;}
336             lvi(LaIndex(0,bs-1),1).inject(b10);
337           //A12*\psi
338           b10=lv1[i];
339           for(int j=0;j<bs;j++)
340           { COMPLEX b0;b0.r=A[i-1][2](j);
341             b10(j,LaIndex(0,tvs-1))*=b0;}
342             lvi(LaIndex(0,bs-1),2).inject(b10);
343         //C interaction happens between lv1[i] with lv1[i-I+1]
344         LaGenMatComplex b40(bs,tc);
345         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BL[i-1-I+1]),
          lv1[i-I+1],b40);
346         lvi(LaIndex(0,bs-1),LaIndex(3*tvs,3*tvs+tc-1)).inject
          (b40);
347         //E interaction happens between lv1[i] with
          lv1[i-I+2];
348         LaGenMatComplex b50(bs,te);
349         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(DL[i-1-I+2]),
          lv1[i-I+2],b50);
350         lvi(LaIndex(0,bs-1),LaIndex(3*tvs+tc,3*tvs+tc+te-1)).
          inject(b50);
351             //----------compression------
352         int m=min(n_svd,bs);
353         LaGenMatComplex TVi(tti.size(0),m);
354         LaGenMatComplex LVi(lvi.size(0),m);
355         LaVectorDouble Lamda(m);
356       Compression(tti, lvi, TVi,LVi, Lamda);
357       TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;
358       }
359        else if(s[i-1][0]==0&s[i-1][1]==(K-1)/2)//case 5
360       { int ds=d[i-1].size(); int tvs=tv[i].size(1);
361         int I=s[i-1][0]+s[i-1][1]+2;//there is no B
          interaction
362         int te=tv[i-I+2].size(1);//E interaction happens
          between tv[i] and tv[i-I+2]
363         LaGenMatComplex tti(ds,tvs*2+te);
364         //I*\psi
365         tti(LaIndex(0,ds-1),0).inject(tv[i]);
366         //A14*\psi
367         LaGenMatComplex a10(ds,tvs);
368         for(int j=0;j<ds;j++)
369         {COMPLEX a00;a00.r=A[i-1][4](j)*tv[i](j,0).r;a00.i=A
          [i-1][4](j)*tv[i](j,0).i;
370          a10(j,0)=a00;
371         }
372         tti(LaIndex(0,ds-1),1).inject(a10);
373         //E interaction happens between tv[i] with tv[i-I+2];
```

```
374           LaGenMatComplex a50(ds,te);
375           Blas_Mat_Mat_Mult(LaGenMatComplex(DT[i-1-I+2]),tv[i-
              I+2],a50);
376           COMPLEX d0;d0.r=0;d0.i=1;
377           a50*=d0;
378           tti(LaIndex(0,ds-1),LaIndex(2*tvs,2*tvs+te-1)).
              inject(a50);
379           //int bs=b[i-1].size();
380           LaGenMatComplex lvi(1,tvs*2+te);
381           //A10*\psi
382             COMPLEX b0;b0.r=A[i-1][0](0)*lv1[i](0,0).r;b0.i=A[
                i-1][0](0)*lv1[i](0,0).i;
383             lvi(0,0)=b0;
384           //A12*\psi
385             b0.r=A[i-1][2](0)*lv1[i](0,0).r;b0.i=A[i-1][2](0)*
                lv1[i](0,0).i;
386             lvi(0,1)=b0;
387         //E interaction happens between lv1[i] with
          lv1[i-I+2];
388         LaGenMatComplex b50(1,te);
389         LaGenMatComplex dl(2,1);dl(0,0).r=DL[i-I+1](0,0);dl(0
          ,0).i=0;dl(1,0).r=DL[i-I+1](1,0);dl(1,0).i=0;
390         Blas_Mat_Trans_Mat_Mult(dl,lv1[i-I+2],b50);
391         lvi(0,2)=b50(0,0);
392         lvi(0,3)=b50(0,1);
393         //-------------compression------------
394         int m=min(n_svd,1);
395         LaGenMatComplex TVi(tti.size(0),m);
396         LaGenMatComplex LVi(lvi.size(0),m);
397         LaVectorDouble Lamda(m);
398         complex<double> lvi0(lvi(0,0).r,lvi(0,0).i);
399         complex<double> lvi1(lvi(0,1).r,lvi(0,1).i);
400         complex<double> lvi2(lvi(0,2).r,lvi(0,2).i);
401         complex<double> lvi3(lvi(0,3).r,lvi(0,3).i);
402         if(abs(lvi0)<1.0e-50&abs(lvi1)<1.0e-50&abs(lvi2)<
          1.0e-50&abs(lvi3)<1.0e-50)
403         {
404         cout<<"yes"<<endl;
405         TV[i]=tv[i]; LV[i]=lv1[i]; Sigma1[i]=Sigma[i];Sigma1[
          i]=1;}
406          else
407          {
408           Compression(tti, lvi, TVi,LVi, Lamda);
409           TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;
410          }
411        }
412        else
```

```
413         { if (s[i-1][1]!=0||s[i-1][0]!=(K-1)/2)
414           throw range_error("Incorrect location for pv
              interaction");   //case 6
415           int ds=d[i-1].size(); int tvs=tv[i].size(1);
416           int I=s[i-1][0]+s[i-1][1]+2;//there is no B
              interaction
417           int tc=tv[i-I+1].size(1);//C interaction happens
              between tv[i] and tv[i-I+1]
418           LaGenMatComplex tti(ds,tvs*2+tc);
419           //I*\psi
420           tti(LaIndex(0,ds-1),LaIndex(0,tvs-1)).inject(tv[i]);
421           //A13*\psi
422           LaGenMatComplex a10(ds,tvs);a10=tv[i];
423           for(int j=0;j<ds;j++)
424           {COMPLEX a0;a0.r=A[i-1][3](j);
425           a10(j,LaIndex(0,tvs-1))*=a0;}
426           tti(LaIndex(0,ds-1),LaIndex(tvs,2*tvs-1)).inject(a10
              );
427           //C interaction happens between tv[i] with tv[i-I+1];
428           LaGenMatComplex a40(ds,tc);
429           Blas_Mat_Mat_Mult(LaGenMatComplex(BT[i-1-I+1]),tv[i-
              I+1],a40);
430           tti(LaIndex(0,ds-1),LaIndex(2*tvs,2*tvs+tc-1)).
              inject(a40);
431           int bs=b[i-1].size();
432           LaGenMatComplex lvi(bs,tvs*2+tc);
433           //A10*\psi
434           LaGenMatComplex b10(bs,tvs);b10=lv1[i];
435           for(int j=0;j<bs;j++)
436           { COMPLEX b0;b0.r=A[i-1][0](j);
437             b10(j,LaIndex(0,tvs-1))*=b0;}
438             lvi(LaIndex(0,bs-1),LaIndex(0,tvs-1)).inject(b10);
439           //A11*\psi
440           b10=lv1[i];
441           for(int j=0;j<bs;j++)
442           { COMPLEX b0;b0.r=A[i-1][1](j);
443             b10(j,LaIndex(0,tvs-1))*=b0;}
444             lvi(LaIndex(0,bs-1),LaIndex(tvs,2*tvs-1)).inject(
                b10);
445         //C interaction happens between lv1[i] with lv1[i-I+1]
446         LaGenMatComplex b40(bs,tc);
447         Blas_Mat_Trans_Mat_Mult(LaGenMatComplex(BL[i-1-I+1]),
            lv1[i-I+1],b40);
448         lvi(LaIndex(0,bs-1),LaIndex(2*tvs,2*tvs+tc-1)).inject
            (b40);
449             //-------------compression----------
450         int m=min(n_svd,bs);
```

```
451           LaGenMatComplex TVi(tti.size(0),m);
452           LaGenMatComplex LVi(lvi.size(0),m);
453           LaVectorDouble Lamda(m);
454         // Compression(tti, lvi, TVi,LVi, Lamda,v2);
455
            //TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;norm_const1[i]=v2
            ;
456           complex<double> lvi0(lvi(0,0).r,lvi(0,0).i);
457           complex<double> lvi1(lvi(0,1).r,lvi(0,1).i);
458           complex<double> lvi2(lvi(0,2).r,lvi(0,2).i);
459           complex<double> lvi3(lvi(0,3).r,lvi(0,3).i);
460           if(abs(lvi0)<1.0e-50&abs(lvi1)<1.0e-50&abs(lvi2)<
              1.0e-50&abs(lvi3)<1.0e-50)
461           {
462           cout<<"yes"<<endl;
463           TV[i]=tv[i]; LV[i]=lv1[i]; Sigma1[i]=Sigma[i];Sigma1[
              i]=1;}
464           else
465           {
466            Compression(tti, lvi, TVi,LVi, Lamda);
467            TV[i]=TVi;LV[i]=LVi;Sigma1[i]=Lamda;
468           }
469          }
470   }
471   }
472   #endif
473
```

A.2.4 compression.h

```
#ifndef COMPRESSION_H
#define COMPRESSION_H
#define LA_COMPLEX_SUPPORT
#include <iostream>
#include <lapack++.h>
#include <vector>
#include <math.h>
#include <complex>
using namespace std;
//void Compression(LaGenMatComplex &A1, LaGenMatComplex
&A2, LaGenMatComplex &Alpha1, LaGenMatComplex
&Alpha2,LaVectorDouble &Lamda,double &v2)
void Compression(LaGenMatComplex &A1, LaGenMatComplex &A2,
LaGenMatComplex &Alpha1, LaGenMatComplex &Alpha2,
LaVectorDouble &Lamda)
{
 int x=A1.size(1);
LaGenMatComplex C1(x,x);
Blas_Mat_Trans_Mat_Mult(A1,A1,C1);//involve conjugate
transpose,it is correct!
LaVectorDouble Sigma1(x);
LaGenMatComplex U1(x,x);
LaGenMatComplex VT1(x,x);
LaSVD_IP(C1,Sigma1,U1,VT1);// element v'v=U1*Sigma1*VT1,
U1(i,:) and VT1(:,i) are conjugate transpose. Column of U1
are eigenvectors of C1
std::vector<double> sigma1;
for(int i=0;i<Sigma1.size();i++)
{ if (Sigma1(i)!=0) sigma1.push_back(Sigma1(i));}
int y=sigma1.size();
LaGenMatComplex X1(x,y);
for(int i=0;i<x;i++)
{ for (int j=0;j<y;j++)
   X1(i,j)=U1(i,j);
}
LaGenMatComplex XX1(x,y);
XX1=X1;
for(int i=0;i<y;i++)
{COMPLEX a;a.r=pow(sigma1[i],-0.5);XX1(LaIndex(0,x-1),i)*=a;}
LaGenMatComplex B1(A1.size(0),y);
Blas_Mat_Mat_Mult(A1,XX1,B1);
//---------------------B1 is obtained from the above steps
and the followings are for
B2-------------------------------------------
LaGenMatComplex C2(x,x);
Blas_Mat_Trans_Mat_Mult(A2,A2,C2);//involve conjugate
transpose,it is correct!
```

```
38   LaVectorDouble Sigma2(x);
39   LaGenMatComplex U2(x,x);
40   LaGenMatComplex VT2(x,x);
41   LaSVD_IP(C2,Sigma2,U2,VT2);// element v'v=U1*Sigma1*VT1,
     U1(i,:) and VT1(:,i) are conjugate transpose. Column of U1
     are eigenvectors of C1.
42   vector<double> sigma2;
43   for(int i=0;i<Sigma2.size();i++)
44   { if(Sigma2(i)!=0) sigma2.push_back(Sigma2(i));}
45   int z=sigma2.size();
46   LaGenMatComplex X2(x,z);
47   for(int i=0;i<x;i++)
48   { for(int j=0;j<z;j++)
49     X2(i,j)=U2(i,j);
50   }
51   LaGenMatComplex XX2(x,z);
52   XX2=X2;
53   for(int i=0;i<z;i++)
54   {COMPLEX a;a.r=pow(sigma2[i],-0.5);XX2(LaIndex(0,x-1),i)*=a;}
55   LaGenMatComplex B2(A2.size(0),z);
56   Blas_Mat_Mat_Mult(A2,XX2,B2);
57   //Orthonomalization of A1 and A2 is done!
58   LaGenMatComplex C(y,z);
59   Blas_Mat_Trans_Mat_Mult(X1,X2,C);
60   for(int i=0;i<y;i++)
61   { COMPLEX s1;s1.r=pow(sigma1[i],0.5);
62     C(i,LaIndex(0,z-1))*=s1;
63   }
64   for(int i=0;i<z;i++)
65   { COMPLEX s2;s2.r=pow(sigma2[i],0.5);
66    C(LaIndex(0,y-1),i)*=s2;
67   }
68   int mi=min(y,z);
69   LaVectorDouble lamda(mi);
70   LaGenMatComplex X(y,y);
71   LaGenMatComplex Y(z,z);
72   LaSVD_IP(C,lamda,X,Y);
73   LaGenMatComplex XX(y,mi);
74   if(y==mi) XX=X;
75   else{
76    for(int i=0;i<y;i++)
77    { for(int j=0;j<mi;j++)
78      XX(i,j)=X(i,j);}}
79   LaGenMatComplex YY(z,mi);
80   for(int i=0;i<z;i++)
81   { for(int j=0;j<mi;j++)
82     { YY(i,j).r= Y(j,i).r;
```

```
83        YY(i,j).i= -Y(j,i).i;
84      }
85    }
86
87    /*v2=Blas_Norm2(lamda);
88    for(int i=0;i<mi;i++)
89    lamda(i)/=v2;*/
90    int m=Lamda.size();
91    for(int i=0;i<m;i++)
92    Lamda(i)=lamda(i);
93    LaGenMatComplex alpha1(A1.size(0),mi);
94    LaGenMatComplex alpha2(A2.size(0),mi);
95    Blas_Mat_Mat_Mult(B1,XX,alpha1);
96    Blas_Mat_Mat_Mult(B2,YY,alpha2);
97    for(int i=0;i<A1.size(0);i++)
98    { for(int j=0;j<m;j++)
99       Alpha1(i,j)=alpha1(i,j);
100   }
101   for(int i=0;i<A2.size(0);i++)
102   { for(int j=0;j<m;j++)
103      Alpha2(i,j)=alpha2(i,j);
104   }
105   }
106   #endif
107
```

A.3 Lanczos Iteration

A.3.1 SvdSum.h

```
1   #ifndef SVDSUM_H
2   #define SVDSUM_H
3   #define LA_COMPLEX_SUPPORT
4   #include <iostream>
5   #include <lapack++.h>
6   #include <math.h>
7   #include <complex>
8   #include "parameter.h"
9   #include "Compression.h"
10  using namespace std;
11  void SvdSum(LaGenMatComplex &u1, LaVectorDouble &Sigma1,
    LaGenMatComplex &v1, LaGenMatComplex &u2, LaVectorDouble &
    Sigma2, LaGenMatComplex &v2)
12  { LaGenMatComplex mu1(u1);
13   LaGenMatComplex mu2(u2);
14   int m=u1.size(0);int n=u1.size(1);
15   for(int i=0;i<n;i++)
16   { COMPLEX a1;a1.r=Sigma1(i);
17     mu1(LaIndex(0,m-1),i)*=a1;
18     COMPLEX a2;a2.r=Sigma2(i);
19     mu2(LaIndex(0,m-1),i)*=a2;
20   }
21   LaGenMatComplex B1(m,2*n);
22   B1(LaIndex(0,m-1),LaIndex(0,n-1)).inject(mu1);
23   B1(LaIndex(0,m-1),LaIndex(n,2*n-1)).inject(mu2);
24   int x=v1.size(0);int y=v1.size(1);
25   LaGenMatComplex B2(x,2*y);
26   B2(LaIndex(0,x-1),LaIndex(0,y-1)).inject(v1);
27   B2(LaIndex(0,x-1),LaIndex(y,2*y-1)).inject(v2);
28   LaGenMatComplex Alpha1(m,n);
29   LaGenMatComplex Alpha2(x,n);
30   LaVectorDouble Lamda(n);
31   Compression(B1, B2, Alpha1,Alpha2,Lamda);
32   u1=Alpha1;v1=Alpha2;Sigma1=Lamda;
33  }
34  #endif
35
```

A.3.2 SvdProduct.h

```
#define LA_COMPLEX_SUPPORT
#include <iostream>
#include <lapack++.h>
#include <math.h>
#include "count.h"
#include <complex>
using namespace std;
complex<double> SvdProduct(LaGenMatComplex &u1,
LaVectorDouble &Sigma1,LaGenMatComplex &v1, LaGenMatComplex
&u2,LaVectorDouble &Sigma2, LaGenMatComplex &v2,
vector<vector<int> > &bi, vector<vector<pair<int,int> > > &
di, vector<int> &si)
{ complex<double> product=0.0;
 int a=u1.size(0);int b=v1.size(0); int m=Sigma1.size();
for(int i=0;i<a;i++)
{ for(int j=0;j<b;j++)
  { complex<double> Aij=0.0;//Calculate Aij
    complex<double> Bij=0.0;//Calculate Bij
    for(int x=0;x<m;x++)
    { complex<double> uu1(u1(i,x).r,u1(i,x).i);
      complex<double> uu2(u2(i,x).r,u2(i,x).i);
      complex<double> vv1(v1(j,x).r,-v1(j,x).i);
      complex<double> vv2(v2(j,x).r,-v2(j,x).i);
     Aij+=uu1*Sigma1(x)*vv1;
     Bij+=uu2*Sigma2(x)*vv2;}

  if(si[0]>1)
  { vector<int> bi0(bi[j].begin(),bi[j].begin()+si[0]);
    vector<int> repeat_b=count(bi0);
    if(repeat_b.size()>0)
    {vector<pair<int,int> > di0(di[i].begin(),di[i].begin()+
    si[0]);
     vector<int> repeat_d=countpair(di0);
     if(repeat_d.size()>0)
      {for(int m=0;m<repeat_b.size();m++)
       { for(int n=0;n<repeat_d.size();n++)
         {Aij/=sqrt(factor1(min(repeat_b[m],repeat_d[n])));
         Bij/=sqrt(factor1(min(repeat_b[m],repeat_d[n])));}
       }
      }
    }
  }

   if(si[1]>1)
  { vector<int> bi0(bi[j].end()-si[1],bi[j].end());
    vector<int> repeat_b=count(bi0);
    if(repeat_b.size()>0)
```

```
44        {vector<pair<int,int> > di0(di[i].end()-si[1],di[i].end
          ());
45         vector<int> repeat_d=countpair(di0);
46         if(repeat_d.size()>0)
47          {for(int m=0;m<repeat_b.size();m++)
48           { for(int n=0;n<repeat_d.size();n++)
49             {Aij/=sqrt(factor1(min(repeat_b[m],repeat_d[n])));
50             Bij/=sqrt(factor1(min(repeat_b[m],repeat_d[n])));}
51           }
52          }
53        }
54      }
55
56      product+=Aij*Bij;
57    }
58    }
59     return product;
60    }
61
```

A.3.3 SvdNorm.h

```
#define LA_COMPLEX_SUPPORT
#include <iostream>
#include <lapack++.h>
#include <math.h>
#include "count.h"
#include <complex>
using namespace std;
complex<double>
SvdNorm(LaGenMatComplex &u1, LaVectorDouble &Sigma1,
LaGenMatComplex &v1, vector<vector<int> > &bi, vector<vector
<pair<int,int> > > &di, vector<int> &si)
{ complex<double> product=0.0;
 int a=u1.size(0);int b=v1.size(0); int m=Sigma1.size();
for(int i=0;i<a;i++)
{ for(int j=0;j<b;j++)
  { complex<double> Aij=0.0;//Calculate Aij
    for(int x=0;x<m;x++)
    { complex<double> uu1(u1(i,x).r,u1(i,x).i);
      complex<double> vv1(v1(j,x).r,-v1(j,x).i);
     Aij+=uu1*Sigma1(x)*vv1;}
  if(si[0]>1)
  { vector<int> bi0(bi[j].begin(),bi[j].begin()+si[0]);
    vector<int> repeat_b=count(bi0);
    if(repeat_b.size()>0)
    {vector<pair<int,int> > di0(di[i].begin(),di[i].begin()+
    si[0]);
     vector<int> repeat_d=countpair(di0);
     if(repeat_d.size()>0)
      {for(int m=0;m<repeat_b.size();m++)
       { for(int n=0;n<repeat_d.size();n++)
         Aij/=sqrt(factor1(min(repeat_b[m],repeat_d[n])));
       }
      }
    }
  }
   if(si[1]>1)
  { vector<int> bi0(bi[j].end()-si[1],bi[j].end());
    vector<int> repeat_b=count(bi0);
    if(repeat_b.size()>0)
    {vector<pair<int,int> > di0(di[i].end()-si[1],di[i].end
    ());
     vector<int> repeat_d=countpair(di0);
     if(repeat_d.size()>0)
      {for(int m=0;m<repeat_b.size();m++)
       { for(int n=0;n<repeat_d.size();n++)
         Aij/=sqrt(factor1(min(repeat_b[m],repeat_d[n])));
       }
```

```
44          }
45        }
46      }
47      product+=Aij*conj(Aij);
48    }
49    }
50     return product;
51    }
52
```

A.3.4 count.h

```
1   /* count.h
2      input: vector<int> a;
3      output: vector<int> d; count the number of a repeating
       element and push the number back into d;
4      for example a={1,1,1,2,3,4,5,5,4};
5      the output d={3,2,2};
6
7      coutpair()
8      input: vector<pair<int,int>> a;
9      output: vector<int> d; count the number of a repeating
       pair and push_back the number into d;
10     for example a={(1,1),(1,2),(1,1),(5,5),(2,1),(5,5)};
11     the output d={2,2};
12  */
13  #ifndef COUNT_H
14  #define COUNT_H
15  # include <stdio.h>
16  # include <vector>
17  # include <iostream>
18  # include <stdlib.h>
19  using namespace std;
20  vector<int> count(vector<int> &a)
21  {int i,j;
22   int MAX=a.size();
23   vector<bool> A(MAX);
24   vector<int> b(MAX,0);
25   vector<int> d;
26
27  for(i=0;i<MAX-1;i++)
28  for(j=i+1;j<MAX;j++)
29  if(A[i]!=true &&a[j]==a[i])
30  { b[i]++;
31    A[j]=true;
32  }
33  for(i=0;i<MAX;i++)
34  if(A[i]!=true)
35  b[i]++;
36  for(i=0;i<MAX;i++)
37  if(b[i]>1)
38  d.push_back(b[i]);
39   return d;
40  }
41
42  vector<int> countpair(vector<pair<int,int> > &a)
43  {int i,j;
44   int MAX=a.size();
45   vector<bool> A(MAX);
```

```
46    vector<int> b(MAX,0);
47    vector<int> d;
48
49   for(i=0;i<MAX-1;i++)
50   for(j=i+1;j<MAX;j++)
51   if(A[i]!=true &&a[j].first==a[i].first&&a[j].second==a[i].
     second)
52   { b[i]++;
53     A[j]=true;
54   }
55   for(i=0;i<MAX;i++)
56   if(A[i]!=true)
57   b[i]++;
58   for(i=0;i<MAX;i++)
59   if(b[i]>1)
60   d.push_back(b[i]);
61    return d;
62   }
63
64   double factor1(int n )
65   {
66       double product = 1;
67       for (int i = 2; i <= n; i++)
68           product *= i;
69       return product;
70   }
71   #endif
72
```

A.3.5 Lanczos.cpp

```
#define LA_COMPLEX_SUPPORT
#include <iostream>
#include <stdexcept>
#include <vector>
#include <complex>
#include <lapack++.h>
#include <math.h>
#include "basis.h"
#include "parameter.h"
#include "Hamiltonian.h"
#include "HDotU.h"
#include "SvdSum.h"
#include "SvdProduct.h"
#include "SvdSum.h"
#include "Eigenvalue.h"
using namespace std;
using namespace Sum2N;
int main()
{
    vector<vector<LaVectorDouble> > A; vector<LaGenMatDouble
     > BL;
    vector<LaGenMatDouble > BT; vector<LaGenMatDouble > DL;
    vector<LaGenMatDouble > DT;
    vector<vector<vector<int> > > b;
    vector<vector<vector<pair<int,int> > > > d;
    vector<vector<int> > s;
    Hamiltonian(A,BL,BT,DL,DT,b,d,s);
    int ss=s.size()+1;
    //To declare v_i=tv*Sigma*lv'
    vector<LaGenMatComplex> tv(ss);//transverse vectors, +1
    refers to \psi^{(0,0)}
    vector<LaGenMatComplex> lv(ss);//longitudinal vectors
    vector<LaVectorDouble> Sigma(ss);//Diagonal Matrices
    //To declare w1=Hv_i=TV*Sigma1*LV'
    vector<LaGenMatComplex> TV(ss);//new transverse vectors
    vector<LaGenMatComplex> LV(ss);//new longitudinal vectors
    vector<LaVectorDouble> Sigma1(ss);//new diagonal Matrices
    //To declare v_{i+1}=tv_1*Sigma_1*lv_1'
    vector<LaGenMatComplex> tv_1(ss);//new transverse vectors
    vector<LaGenMatComplex> lv_1(ss);//new longitudinal
    vectors
    vector<LaVectorDouble> Sigma_1(ss);//new diagonal
    Matrices
    //initialize tv,lv and Sigma such that \psi is
    normalized; Also initialize tv_1,lv_1 and Sigma_1
    LaGenMatComplex t1(1,1);
    t1(0,0).r=0.9999999999;
```

```
42      tv[0]=t1;tv_1[0]=t1;
43      LaGenMatComplex t2(1,1);
44      t2(0,0).r=1.0;
45      lv[0]=t2;lv_1[0]=t2;
46      LaVectorDouble sigma0(1);
47      sigma0(0)=1.0;
48      Sigma[0]=sigma0;Sigma_1[0]=sigma0;
49      for(size_t i=1;i<ss;i++)
50      { int x=b[i-1].size();
51        int m=min(x,n_svd);
52        LaGenMatComplex ti(d[i-1].size(),m);
          //ti.rand(d[i-1].size(),m);
53        LaGenMatComplex li(b[i-1].size(),m);
          //li.rand(b[i-1].size(),m);
54        LaVectorDouble sigmai(m);
55        ti=0;li=0;sigmai=1;
56        tv[i]=ti;lv[i]=li;Sigma[i]=sigmai;
57        tv_1[i]=ti;lv_1[i]=li;Sigma_1[i]=sigmai;
58        COMPLEX comp;comp.r=1;comp.i=0;
59        COMPLEX com;com.r=sqrt((1-pow(t1(0,0).r,2))*2/(K+1));
          com.i=0;// To avoid 0s for SVD form and the normality
          should be guranteed
60        tv[i](0,0)=comp;lv[i](lv[i].size(0)-1,lv[i].size(1)-1
          )=com;
61
          //tv[i](0,0)=comp;lv[i](lv[i].size(0)-1,lv[i].size(1)-1
          )=com;
62      }
63      int n=30;
64      LaVectorComplex a(n);
65      LaVectorComplex beta(n+1);
66      HDotU(A,BL,BT,DL,DT,b,d,s,tv,lv,Sigma,TV,LV,Sigma1);
67      complex<double> aa(TV[0](0,0).r,TV[0](0,0).i);
68      complex<double> tt(tv[0](0,0).r,tv[0](0,0).i);
69      //v1=tv*Sigma*lv' w1=TV*Sigma1*LV'
70      complex<double> a0; a0=aa*tt;
71      //a0 is initialized to be \psi^(0,0)*\'psi^(0,0)
72      for(int i=1;i<ss;i++)
73      { complex<double> a1; a1=SvdProduct(TV[i],Sigma1[i],LV[i
        ],tv[i],Sigma[i],lv[i]);
74        a0+=a1;
75      }
76      a(0).r=real(a0);a(0).i=imag(a0);
77      beta(0).r=0; beta(0).i=0;
78      cout<<"a(0)="<<a(0)<<endl;
79      //First calculate w1' and store 'it' in
        tv_1*Sigma_1*lv_1, calculate b1=beta(1);
```

```
80
81        tv_1[0](0,0).r=real(aa-tt*a0);tv_1[0](0,0).i=imag(aa-tt*
          a0);
82        Sigma_1[0]=1;lv_1[0]=1;
83        complex<double> b1(pow(tv_1[0](0,0).r,2.0)-pow(tv_1[0](0
          ,0).i,2.0),2*tv_1[0](0,0).r*tv_1[0](0,0).i);
84        for(int i=1;i<ss;i++)
85        { lv_1[i]=lv[i];tv_1[i]=tv[i];Sigma_1[i]=Sigma[i];
86          COMPLEX ao;ao.r=-real(a0);ao.i=-imag(a0);
87          tv_1[i](LaIndex(0,tv_1[i].size(0)-1),LaIndex(0,tv_1[i
            ].size(1)-1))*=ao;
88          SvdSum(tv_1[i],Sigma_1[i],lv_1[i],TV[i],Sigma1[i],LV[i
            ]);
89          complex<double> bb;bb=SvdProduct(tv_1[i],Sigma_1[i],
            lv_1[i],tv_1[i],Sigma_1[i],lv_1[i]);
90          b1+=bb;
91        }
92        b1=sqrt(b1);
93        beta(1).r=real(b1);beta(1).i=imag(b1);
94        cout<<"beta(1)="<<beta(1)<<endl;
95        //Normalize w1', devide w1' by beta(1);
96        complex<double> inv(1,0);
97        b1=inv/b1;
98        COMPLEX bb; bb.r=real(b1);bb.i=imag(b1);
99        for(int i=1;i<ss;i++)
100       tv_1[i](LaIndex(0,tv_1[i].size(0)-1),LaIndex(0,tv_1[i].
          size(1)-1))*=bb;
101       tv_1[0]*=bb;
102       //First iteration is finished, the following loops are
          for following interations
103       for(int m=1;m<n;m++)
104       {
105       HDotU(A,BL,BT,DL,DT,b,d,s,tv_1,lv_1,Sigma_1,TV,LV,Sigma1
          );
106       LaGenMatComplex C(1,1);
107       Blas_Mat_Mat_Mult(TV[0],tv_1[0],C);
108       a0.real()=C(0,0).r;a0.imag()=C(0,0).i;
109       for(int i=1;i<ss;i++)
110       { complex<double> a1; a1=SvdProduct(TV[i],Sigma1[i],LV[i
          ],tv_1[i],Sigma_1[i],lv_1[i]);
111         a0+=a1;
112       }
113       a(m).r=real(a0);a(m).i=imag(a0);
114       cout<<"a"<<m<<"="<<a(m)<<endl;
115       complex<double> tv1(tv_1[0](0,0).r,tv_1[0](0,0).i);
116       complex<double> b_1(beta(m).r,beta(m).i);
117       complex<double> tv0(tv[0](0,0).r,tv[0](0,0).i);
```

```
118       TV[0](0,0).r=TV[0](0,0).r-real(a0*tv1)-real(b_1*tv0);TV[
          0](0,0).i=TV[0](0,0).i-imag(tv1*a0)-imag(b_1*tv0);
119       complex<double> b1(pow(TV[0](0,0).r,2.0)-pow(TV[0](0,0).
          i,2.0),2*TV[0](0,0).r*TV[0](0,0).i);
120       for(int i=1;i<ss;i++)
121       { LaGenMatComplex llv(tv_1[i].size(0),tv_1[i].size(1));
122         llv=tv_1[i];
123         COMPLEX am;am.r=-real(a0);am.i=-imag(a0);
124         llv(LaIndex(0,llv.size(0)-1),LaIndex(0,llv.size(1)-1
          ))*=am;
125         SvdSum(TV[i],Sigma1[i],LV[i],llv,Sigma_1[i],lv_1[i]);
126         llv=tv[i];
127         COMPLEX bm;bm.r=-real(b_1);bm.i=-imag(b_1);
128         llv(LaIndex(0,llv.size(0)-1),LaIndex(0,llv.size(1)-1
          ))*=bm;
129         SvdSum(TV[i],Sigma1[i],LV[i],llv,Sigma[i],lv[i]);
130         complex<double> bb;bb=SvdProduct(TV[i],Sigma1[i],LV[i
          ],TV[i],Sigma1[i],LV[i]);
131         //cout<<bb<<endl;
132         b1+=bb;
133       }
134       b1=sqrt(b1);
135       beta(m+1).r=real(b1);beta(m+1).i=imag(b1);
136       cout<<"beta"<<m+1<<"="<<beta(m+1)<<endl;
137       //Normalize w1', devide w1' by beta(1)
138       b1=inv/b1;
139       COMPLEX bb; bb.r=real(b1);bb.i=imag(b1);
140       TV[0]*=bb;
141       for(int i=1;i<ss;i++)
142       TV[i](LaIndex(0,TV[i].size(0)-1),LaIndex(0,TV[i].size(1
          )-1))*=bb;
143
144       cout<<"check TV with tv"<<endl;
145       complex<double> c(tv[0](0,0).r,tv[0](0,0).i);
146       complex<double> o(TV[0](0,0).r,TV[0](0,0).i);
147       c=o*c;
148       for(int i=1;i<ss;i++)
149       { complex<double> a1; a1=SvdProduct(TV[i],Sigma1[i],LV[i
          ],tv[i],Sigma[i],lv[i]);
150         c+=a1;
151       }
152        cout<<"c="<<c<<endl;
153       cout<<"check TV with tv_1"<<endl;
154       complex<double> ccc(tv_1[0](0,0).r,tv_1[0](0,0).i);
155       complex<double> ooo(TV[0](0,0).r,TV[0](0,0).i);
156       ccc=ooo*ccc;
157       for(int i=1;i<ss;i++)
```

```
158       { complex<double> a1; a1=SvdProduct(TV[i],Sigma1[i],LV[i
          ],tv_1[i],Sigma_1[i],lv_1[i]);
159         ccc+=a1;
160       }
161        cout<<"ccc="<<ccc<<endl;
162       cout<<"check tv_1 with tv"<<endl;
163       complex<double> cc(tv_1[0](0,0).r,tv_1[0](0,0).i);
164       complex<double> oo(tv[0](0,0).r,tv[0](0,0).i);
165       cc=oo*cc;
166       for(int i=1;i<ss;i++)
167       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],tv_1[i],Sigma_1[i],lv_1[i]);
168         cc+=a1;
169       }
170        cout<<"cc="<<cc<<endl;
171       // pass the value of tv_1[i],lv_1[i] and Sigma_1[i] to
          tv[i],lv[i] and Sigma[i], respectively;
172       // and pass the value of TV[i],LV[i] and Sigma1[i] to
          tv_1[i],lv_1[i] and Sigma_1[i], respectively;
173           for(int i=0;i<ss;i++)
174       {tv[i]=tv_1[i];lv[i]=lv_1[i];Sigma[i]=Sigma_1[i];
175        tv_1[i]=TV[i];lv_1[i]=LV[i];Sigma_1[i]=Sigma1[i];
176        }
177     }
178     cout<<"n iteration"<<endl;
179     LaVectorComplex W5(n);
180     Eigenvalue(a, beta, W5);
181   }
182
```

A.3.6 Eigenvalue.h

```
#ifndef EIGENVALUE_H
#define EIGENVALUE_H
#define LA_COMPLEX_SUPPORT
#include <iostream>
#include <stdexcept>
#include <vector>
#include <complex>
#include <lapack++.h>
#include <math.h>
using namespace std;
void Eigenvalue(LaVectorComplex &a, LaVectorComplex &beta,
LaVectorComplex &W)
{  int n=a.size();
  LaGenMatComplex A(n,n);
  for(int i=1;i<n-1;i++)
  { A(i,i)=a(i);
    A(i,i+1)=beta(i+1);
    A(i,i-1)=beta(i);
  }
  A(0,0)=a(0);A(0,1)=beta(1);
  A(n-1,n-1)=a(n-1);A(n-1,n-2)=beta(n-1);
  LaGenMatComplex V(n,n);
  LaEigSolve(A,W,V);
  for(int j=0;j<n;j++)
  cout<<W(j)<<endl;
}
#endif

```

A.3.7 power.cpp

```
1   #define LA_COMPLEX_SUPPORT
2   #include <iostream>
3   #include <stdexcept>
4   #include <vector>
5   #include <complex>
6   #include <lapack++.h>
7   #include <math.h>
8   #include "basis.h"
9   #include "SvdNorm.h"
10  #include "parameter.h"
11  #include "Hamiltonian.h"
12  #include "HDotU.h"
13  #include "SvdProduct.h"
14  #include "SvdSum.h"
15  using namespace std;
16  using namespace Sum2N;
17  int main()
18  {   vector<vector<LaVectorDouble> > A; vector<LaGenMatDouble
     > BL;
19      vector<LaGenMatDouble > BT; vector<LaGenMatDouble > DL;
        vector<LaGenMatDouble > DT;
20      vector<vector<vector<int> > > b;
21      vector<vector<vector<pair<int,int> > > > d;
22      vector<vector<int> > s;
23      Hamiltonian(A,BL,BT,DL,DT,b,d,s);
24      int ss=s.size()+1;
25      //To declare v_i=tv*Sigma*lv'
26      vector<LaGenMatComplex> tv(ss);//transverse vectors, +1
        refers to \psi^{(0,0)}
27      vector<LaGenMatComplex> lv(ss);//longitudinal vectors
28      vector<LaVectorDouble> Sigma(ss);//Diagonal Matrices
29      //To declare v_{i+1}=tv_1*Sigma_1*lv_1'
30      vector<LaGenMatComplex> tv_1(ss);//new transverse vectors
31      vector<LaGenMatComplex> lv_1(ss);//new longitudinal
        vectors
32      vector<LaVectorDouble> Sigma_1(ss);//new diagonal
        Matrices
33      //initialize tv,lv and Sigma such that \psi is
        normalized; Also initialize tv_1,lv_1 and Sigma_1
34      LaGenMatComplex t1(1,1);
35      t1(0,0).r=0.8;
36      tv[0]=t1;
37      LaGenMatComplex t2(1,1);
38      t2(0,0).r=1.0;
39      lv[0]=t2;
40      LaVectorDouble sigma0(1);
41      sigma0(0)=1.0;
```

```
42        Sigma[0]=sigma0;
43        for(size_t i=1;i<ss;i++)
44        { int x=b[i-1].size();
45          int m=min(x,n_svd);
46          LaGenMatComplex ti(d[i-1].size(),m);
            //ti.rand(d[i-1].size(),m);
47          LaGenMatComplex li(b[i-1].size(),m);
            //li.rand(b[i-1].size(),m);
48          LaVectorDouble sigmai(m);
49          ti=0;li=0;sigmai=1;
50          tv[i]=ti;lv[i]=li;Sigma[i]=sigmai;
51          COMPLEX comp;comp.r=1;comp.i=0;
52          COMPLEX com;com.r=sqrt((1-pow(t1(0,0).r,2))*2/(K+1));
            com.i=0;// To avoid 0s for SVD form and the normality
            should be guranteed
53          tv[i](0,0)=comp;lv[i](lv[i].size(0)-1,lv[i].size(1)-1
            )=com;
54        }
55        complex<double> inv(1,0);
56        //HDotU(A,BL,BT,DL,DT,b,d,s,tv,lv,Sigma,TV,LV,Sigma1);
57
          //HDotU(A,BL,BT,DL,DT,b,d,s,TV,LV,Sigma1,tv_1,lv_1,Sigma_
          1);
58        for(int k=0;k<200;k++)
59        { //include the factor and normalize u;
60         complex<double> normfactor(pow(tv[0](0,0).r,2.0)+pow(tv
          [0](0,0).i,2.0),0.0);
61         for(int i=1;i<ss;i++)
62         { complex<double> bb;bb=SvdNorm(tv[i],Sigma[i],lv[i],b[
          i-1],d[i-1],s[i-1]);
63           normfactor+=bb;
64         }
65         normfactor=sqrt(normfactor);
66         normfactor=inv/normfactor;
67         COMPLEX NF; NF.r=real(normfactor);NF.i=imag(normfactor);
68         for(int i=0;i<ss;i++)
69         tv[i](LaIndex(0,tv[i].size(0)-1),LaIndex(0,tv[i].size(1
          )-1))*=NF;
70        //To declare w1=Hv_i=TV*Sigma1*LV'
71        vector<LaGenMatComplex> TV(ss);//new transverse vectors
72        vector<LaGenMatComplex> LV(ss);//new longitudinal vectors
73        vector<LaVectorDouble> Sigma1(ss);//new diagonal Matrices
74         // u=tv*Sigma*lv';u1=TV*Sigma1*LV';u2=tv_1*Sigma_1*lv_1'
75          HDotU(A,BL,BT,DL,DT,b,d,s,tv,lv,Sigma,TV,LV,Sigma1);
76          complex<double> b1(pow(TV[0](0,0).r,2.0)+pow(TV[0](0,0
            ).i,2.0),0);
77          for(int i=1;i<ss;i++)
```

```
78        { complex<double> bb;bb=SvdNorm(TV[i],Sigma1[i],LV[i],b
          [i-1],d[i-1],s[i-1]);
79           b1+=bb;
80        }
81        b1=sqrt(b1);
82        b1=inv/b1;
83        COMPLEX bb; bb.r=real(b1);bb.i=imag(b1);
84        for(int i=0;i<ss;i++)
85        TV[i](LaIndex(0,TV[i].size(0)-1),LaIndex(0,TV[i].size(1
          )-1))*=bb;
86
87        HDotU(A,BL,BT,DL,DT,b,d,s,TV,LV,Sigma1,tv_1,lv_1,
          Sigma_1);
88        complex<double> b2(pow(tv_1[0](0,0).r,2.0)+pow(tv_1[0](
          0,0).i,2.0),0.0);
89         for(int i=1;i<ss;i++)
90        { complex<double> bb;bb=SvdNorm(tv_1[i],Sigma_1[i],lv_1
          [i],b[i-1],d[i-1],s[i-1]);
91           b2+=bb;
92        }
93        b2=sqrt(b2);
94        b2=inv/b2;
95        bb.r=real(b2);bb.i=imag(b2);
96        for(int i=0;i<ss;i++)
97        tv_1[i](LaIndex(0,tv_1[i].size(0)-1),LaIndex(0,tv_1[i].
          size(1)-1))*=bb;
98        LaGenMatComplex B(3,3);
99        B(0,0).r=1;B(0,0).i=0;B(1,1).r=1;B(1,1).i=0;B(2,2).r=1;
          B(2,2).i=0;
100       //Calculate B(0,1)=conj(B(1,0))=sum(SvdProduct(u1,u))
101      complex<double> c(tv[0](0,0).r,tv[0](0,0).i);
102      complex<double> o(TV[0](0,0).r,TV[0](0,0).i);
103      c=o*conj(c);
104      for(int i=1;i<ss;i++)
105      { complex<double> a1; a1=SvdProduct(TV[i],Sigma1[i],LV[i
         ],tv[i],Sigma[i],lv[i],b[i-1],d[i-1],s[i-1]);
106        c+=a1;
107      }
108      B(0,1).r=real(c);B(0,1).i=imag(c);B(1,0).r=real(c);B(1,0
         ).i=-imag(c);
109       //calculate B(0,2)=conj(B(2,0))=sum(SvdProduct(u2,u))
110      complex<double> cc(tv_1[0](0,0).r,tv_1[0](0,0).i);
111      complex<double> oo(tv[0](0,0).r,tv[0](0,0).i);
112      cc=cc*conj(oo);
113      for(int i=1;i<ss;i++)
114      { complex<double> a1; a1=SvdProduct(tv_1[i],Sigma_1[i],
         lv_1[i],tv[i],Sigma[i],lv[i],b[i-1],d[i-1],s[i-1]);
```

```
115         cc+=a1;
116       }
117       B(0,2).r=real(cc);B(0,2).i=imag(cc);B(2,0).r=real(cc);B(
          2,0).i=-imag(cc);
118       //calculate B(1,2)=conj(B(2,1))=sum(SvdProduct(u2,u1))
119       complex<double> ccc(tv_1[0](0,0).r,tv_1[0](0,0).i);
120       complex<double> ooo(TV[0](0,0).r,TV[0](0,0).i);
121       ccc=ccc*conj(ooo);
122       for(int i=1;i<ss;i++)
123       { complex<double> a1; a1=SvdProduct(tv_1[i],Sigma_1[i],
          lv_1[i],TV[i],Sigma1[i],LV[i],b[i-1],d[i-1],s[i-1]);
124         ccc+=a1;
125       }
126       B(1,2).r=real(ccc);B(1,2).i=imag(ccc);B(2,1).r=real(ccc
          );B(2,1).i=-imag(ccc);
127       LaGenMatComplex U1(3,3);
128       LaVectorDouble S(3);
129       LaGenMatComplex V1(3,3);
130       LaSVD_IP(B,S,U1,V1);
131
132       for(int j=0;j<U1.size(1);j++)
133        {COMPLEX comp;comp.r=pow(S(j),-0.5);comp.i=0;
134         U1(LaIndex(0,U1.size(0)-1),j)*=comp;
135        }
136        //orthonomalization:
          new_u=ttv*SSigma*llv'=U1(0,0)*u+U1(1,0)*u1+U1(2,0)*u2;
137       complex<double> a(tv[0](0,0).r,tv[0](0,0).i);
138       complex<double> a1(TV[0](0,0).r,TV[0](0,0).i);
139       complex<double> a2(tv_1[0](0,0).r,tv_1[0](0,0).i);
140       vector<LaGenMatComplex> ttv(ss);//transverse vectors,
          +1 refers to \psi^{(0,0)}
141       vector<LaGenMatComplex> llv(ss);//longitudinal vectors
142       vector<LaVectorDouble> SSigma(ss);//Diagonal Matrices
143       for(int i=0;i<ss;i++)
144       {ttv[i]=tv[i];llv[i]=lv[i];SSigma[i]=Sigma[i];
145        ttv[i](LaIndex(0,ttv[i].size(0)-1), LaIndex(0,ttv[i].
          size(1)-1))*=U1(0,0);}
146        complex<double> aa(ttv[0](0,0).r,ttv[0](0,0).i);
147        //ttv[0]=ttv[0]+U1(1,0)*TV[0]+U(2,0)*tv_1[0];
148       complex<double> u10(U1(1,0).r,U1(1,0).i);
149       complex<double> u20(U1(2,0).r,U1(2,0).i);
150        ttv[0](0,0).r=real(aa+u10*a1+u20*a2);ttv[0](0,0).i=imag
          (aa+u10*a1+u20*a2);
151        for(int i=1;i<ss;i++)
152        { LaGenMatComplex sub(TV[i].size(0),TV[i].size(1));
153          sub=TV[i];
154          sub(LaIndex(0,sub.size(0)-1),LaIndex(0,sub.size(1)-1
```

```
          ))*=U1(1,0);
155       SvdSum(ttv[i],SSigma[i],llv[i],sub,Sigma1[i],LV[i]);
156       sub=tv_1[i];
157       sub(LaIndex(0,sub.size(0)-1),LaIndex(0,sub.size(1)-1
          ))*=U1(2,0);
158       SvdSum(ttv[i],SSigma[i],llv[i],sub,Sigma_1[i],lv_1[i
          ]);
159      }
160     //new_u1=U1(0,1)*u+U1(1,1)*u1+U1(2,1)*u2;
161     vector<LaGenMatComplex> TTV(ss);//transverse vectors,
        +1 refers to \psi^{(0,0)}
162     vector<LaGenMatComplex> LLV(ss);//longitudinal vectors
163     vector<LaVectorDouble> SSigma1(ss);//Diagonal Matrices
164     for(int i=0;i<ss;i++)
165     {TTV[i]=TV[i];LLV[i]=LV[i];SSigma1[i]=Sigma1[i];
166      TTV[i](LaIndex(0,TTV[i].size(0)-1), LaIndex(0,TTV[i].
        size(1)-1))*=U1(1,1);}
167     complex<double> aaa(TTV[0](0,0).r,TTV[0](0,0).i);
168     //TTV[0]=TTV[0]+U1(0,1)*tv[0]+U(2,1)*tv_1[0];
169     complex<double> u01(U1(0,1).r,U1(0,1).i);
170     complex<double> u21(U1(2,1).r,U1(2,1).i);
171     TTV[0](0,0).r=real(aaa+u01*a+u21*a2);TTV[0](0,0).i=imag(
        aaa+u01*a+u21*a2);
172     for(int i=1;i<ss;i++)
173      { LaGenMatComplex sub(tv[i].size(0),tv[i].size(1));
174        sub=tv[i];
175        sub(LaIndex(0,sub.size(0)-1),LaIndex(0,sub.size(1)-1
          ))*=U1(0,1);
176        SvdSum(TTV[i],SSigma1[i],LLV[i],sub,Sigma[i],lv[i]);
177        sub=tv_1[i];
178        sub(LaIndex(0,sub.size(0)-1),LaIndex(0,sub.size(1)-1
          ))*=U1(2,1);
179        SvdSum(TTV[i],SSigma1[i],LLV[i],sub,Sigma_1[i],lv_1[i
          ]);
180      }
181      //new_u2=U1(0,2)*u+U1(1,2)*u1+U1(2,2)*u2;
182      for(int i=0;i<ss;i++)
183      tv_1[i](LaIndex(0,tv_1[i].size(0)-1), LaIndex(0,tv_1[i
         ].size(1)-1))*=U1(2,2);
184      //tv_1[0]=tv_1[0]+U1(0,2)*u+U1(1,2)*u1
185     complex<double> u02(U1(0,2).r,U1(0,2).i);
186     complex<double> u12(U1(1,2).r,U1(1,2).i);
187      a2.real()=tv_1[0](0,0).r; a2.imag()=tv_1[0](0,0).i;
188      tv_1[0](0,0).r=real(a2+u02*a+u12*a1);tv_1[0](0,0).i=
         imag(a2+u02*a+u12*a1);
189      for(int i=1;i<ss;i++)
190      { tv[i](LaIndex(0,tv[i].size(0)-1),LaIndex(0,tv[i].size
```

```
          (1)-1))*=U1(0,2);
191         SvdSum(tv_1[i],Sigma_1[i],lv_1[i],tv[i],Sigma[i],lv[i
            ]);
192         TV[i](LaIndex(0,TV[i].size(0)-1),LaIndex(0,TV[i].size
            (1)-1))*=U1(1,2);
193         SvdSum(tv_1[i],Sigma_1[i],lv_1[i],TV[i],Sigma1[i],LV[
            i]);
194        }
195        //clear the storage of TV,Sigma1,LV
196        vector<LaGenMatComplex>().swap(TV);
197        vector<LaGenMatComplex>().swap(LV);
198        vector<LaVectorDouble>().swap(Sigma1);
199       //new u,u1,u2 are ttv*SSigma*llv', TTV*SSigma1*LLV,
          tv_1*Sigma_1*lv_1' respectively.
200       LaGenMatComplex Y(3,3);
201
          //Y00=SvdProduct(Au,u),Y01=SvdProduct(Au1,u),Y02=SvdProdu
          ct(Au2,u);
202
          //Y10=SvdProduct(Au,u1),Y11=SvdProduct(Au1,u1),Y12=SvdPro
          duct(Au2,u1);
203
          //Y20=SvdProduct(Au,u2),Y21=SvdProduct(Au1,u2),Y22=SvdPro
          duct(Au2,u2);
204       //calculate the first column of Y
205       HDotU(A,BL,BT,DL,DT,b,d,s,ttv,llv,SSigma,tv,lv,Sigma);
206       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
207       o.real()=ttv[0](0,0).r;o.imag()=ttv[0](0,0).i;
208       c=c*conj(o);
209       for(int i=1;i<ss;i++)
210       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],ttv[i],SSigma[i],llv[i],b[i-1],d[i-1],s[i-1]);
211         c+=a1;
212       }
213       Y(0,0).r=c.real();Y(0,0).i=c.imag();
214       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
215       o.real()=TTV[0](0,0).r;o.imag()=TTV[0](0,0).i;
216       c=c*conj(o);
217       for(int i=1;i<ss;i++)
218       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],TTV[i],SSigma1[i],LLV[i],b[i-1],d[i-1],s[i-1]);
219         c+=a1;
220       }
221       Y(1,0).r=c.real();Y(1,0).i=c.imag();
222       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
223       o.real()=tv_1[0](0,0).r;o.imag()=tv_1[0](0,0).i;
224       c=c*conj(o);
```

```
225       for(int i=1;i<ss;i++)
226       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],tv_1[i],Sigma_1[i],lv_1[i],b[i-1],d[i-1],s[i-1]);
227         c+=a1;
228       }
229       Y(2,0).r=c.real();Y(2,0).i=c.imag();
230      //calculate the second column of Y
231       HDotU(A,BL,BT,DL,DT,b,d,s,TTV,LLV,SSigma1,tv,lv,Sigma);
232       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
233       o.real()=ttv[0](0,0).r;o.imag()=ttv[0](0,0).i;
234       c=c*conj(o);
235       for(int i=1;i<ss;i++)
236       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],ttv[i],SSigma[i],llv[i],b[i-1],d[i-1],s[i-1]);
237         c+=a1;
238       }
239       Y(0,1).r=c.real();Y(0,1).i=c.imag();
240       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
241       o.real()=TTV[0](0,0).r;o.imag()=TTV[0](0,0).i;
242       c=c*conj(o);
243       for(int i=1;i<ss;i++)
244       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],TTV[i],SSigma1[i],LLV[i],b[i-1],d[i-1],s[i-1]);
245         c+=a1;
246       }
247       Y(1,1).r=c.real();Y(1,1).i=c.imag();
248       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
249       o.real()=tv_1[0](0,0).r;o.imag()=tv_1[0](0,0).i;
250       c=c*conj(o);
251       for(int i=1;i<ss;i++)
252       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],tv_1[i],Sigma_1[i],lv_1[i],b[i-1],d[i-1],s[i-1]);
253         c+=a1;
254       }
255       Y(2,1).r=c.real();Y(2,1).i=c.imag();
256      // Calculate the third column of Y
257       HDotU(A,BL,BT,DL,DT,b,d,s,tv_1,lv_1,Sigma_1,tv,lv,Sigma
          );
258       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
259       o.real()=ttv[0](0,0).r;o.imag()=ttv[0](0,0).i;
260       c=c*conj(o);
261       for(int i=1;i<ss;i++)
262       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],ttv[i],SSigma[i],llv[i],b[i-1],d[i-1],s[i-1]);
263         c+=a1;
264       }
265       Y(0,2).r=c.real();Y(0,2).i=c.imag();
```

```
266       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
267       o.real()=TTV[0](0,0).r;o.imag()=TTV[0](0,0).i;
268       c=c*conj(o);
269       for(int i=1;i<ss;i++)
270       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],TTV[i],SSigma1[i],LLV[i],b[i-1],d[i-1],s[i-1]);
271         c+=a1;
272       }
273       Y(1,2).r=c.real();Y(1,2).i=c.imag();
274       c.real()=tv[0](0,0).r;c.imag()=tv[0](0,0).i;
275       o.real()=tv_1[0](0,0).r;o.imag()=tv_1[0](0,0).i;
276       c=c*conj(o);
277       for(int i=1;i<ss;i++)
278       { complex<double> a1; a1=SvdProduct(tv[i],Sigma[i],lv[i
          ],tv_1[i],Sigma_1[i],lv_1[i],b[i-1],d[i-1],s[i-1]);
279         c+=a1;
280       }
281       Y(2,2).r=c.real();Y(2,2).i=c.imag();
282       LaVectorComplex W(3);
283       LaGenMatComplex VR(3,3);
284       LaEigSolve(Y,W,VR);
285       cout<<W(0)<<" "<<W(1)<<" "<<W(2)<<endl;
286       //find the smallest eigenvalue and its corresponding
          eigenvector
287      complex<double> x(W(0).r,W(0).i);
288      complex<double> y(W(1).r,W(1).i);
289      complex<double> z(W(2).r,W(2).i);
290      int n=2;
291      if (abs(z)>=abs(y)) {z=y; n=1;}
292      if (abs(z)>=abs(x)) {z=x; n=0;}
293      //calculate the resulting wavefunction for the next
         iteration. (u,u1,u2)*VR(n);
294      //new u,u1,u2 are ttv*SSigma*llv', TTV*SSigma1*LLV,
         tv_1*Sigma_1*lv_1' respectively.
295        //new_u=VR(0,n)*u+VR(1,n)*u1+VR(2,n)*u2;
296        //VR(2,n).i=0;
297        for(int i=0;i<ss;i++)
298        tv_1[i](LaIndex(0,tv_1[i].size(0)-1), LaIndex(0,tv_1[i
           ].size(1)-1))*=VR(2,n);
299        //tv_1[0]=tv_1[0]+VR(0,n)*u+VR(1,n)*u1
300       complex<double> VR0(VR(0,n).r,VR(0,n).i);
301       complex<double> VR1(VR(1,n).r,VR(2,n).i);
302       //complex<double> VR0(VR(0,n).r,0);
303       //complex<double> VR1(VR(1,n).r,0);
304       a.real()=ttv[0](0,0).r; a.imag()=ttv[0](0,0).i;
305       a1.real()=TTV[0](0,0).r; a1.imag()=TTV[0](0,0).i;
306       a2.real()=tv_1[0](0,0).r; a2.imag()=tv_1[0](0,0).i;
```

```
307         tv_1[0](0,0).r=real(a2+VR0*a+VR1*a1);tv_1[0](0,0).i=
            imag(a2+VR0*a+VR1*a1);
308         for(int i=1;i<ss;i++)
309         { ttv[i](LaIndex(0,ttv[i].size(0)-1),LaIndex(0,ttv[i].
            size(1)-1))*=VR(0,n);
310           SvdSum(tv_1[i],Sigma_1[i],lv_1[i],ttv[i],SSigma[i],
              llv[i]);
311           TTV[i](LaIndex(0,TTV[i].size(0)-1),LaIndex(0,TTV[i].
              size(1)-1))*=VR(1,n);
312           SvdSum(tv_1[i],Sigma_1[i],lv_1[i],TTV[i],SSigma1[i],
              LLV[i]);
313         }
314         for(int i=0;i<ss;i++)
315         {tv[i]=tv_1[i];Sigma[i]=Sigma_1[i];lv[i]=lv_1[i];}
316         Sigma[0]=1;lv[0]=1;
317       }
318     }
319
```