上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目 原子分子在强场作用下的数值研究方法

学生姓名 辛 麟

学生学号 5120309206

指导教师 何峰特别研究员

专 业 物理学(交大理科班)

学院（系） 致远学院

Submitted in total fulfilment of the requirements for the degree of

Bachelor

in Physics

The Numerical Simulation of Atoms and Molecules in Strong Laser Fields

Lin Xin

Supervisor

Prof. Feng He

Zhiyuan College

Shanghai Jiao Tong University

Shanghai, P.R.China

June. 8th, 2016

原子分子在强场作用下的数值研究方法

摘　　要

原子物理是物理研究方向中非常大的一个研究方向，因为从原子和分子尺度开始，量子现象变的非常重要。经典牛顿方程不再能够描述整个体系，取而代之的是以波函数来描述物质出现概率的薛定谔方程。通过研究原子分子的本征态，人们理解了自然界中稳定元素存在的原因。通过研究波函数含时演化的过程，化学反应中的机制被逐渐理解。然而即使在现在，显微镜所能实现的也只是到原子层面的成像，电子的成像仍然无法实现。简单的来说，成像就如拍照，只有在快门时间很短的时候才能捕捉到一掠而过的飞鸟，同样的道理，由于电子运动的速度很快 (光速的量级)，因此我们需要通过超快激光脉冲来实现对其的成像，这也是研究在这个时间尺度上物理学现象的意义之一。同时在非常强的外场下通常会发生很多与已知规律不同的物理现象，例如在强电场下发生击穿，欧姆定律不再适用。所以通过超强激光与原子分子相互作用我们可以探寻新的规律。通过数值方法求解薛定谔方程，我们可以研究超快超强动力学中具体演化过程，分析波函数来得到直观的理解，以及通过计算得出不同的本征态对最后电离率的贡献。

在本文中，我们首先通过虚时间演化的方法得到原子和分子的本征态，然后通过傅立叶变换法和劈裂算符法求解含时演化方程。我们将这个方法用到两个物理问题上：一，氢原子的电离率与共旋和逆旋圆偏振激光之间的关系，二，氢分子离子中的电荷共振增强电离现象中出现的类弗朗霍菲衍射现象。其次，我们通过使用 B 样条函数，优化变分法来获得原子和分子的所有本征态，并提

升由 Muller 提出的分轨道求解法，做为一个快速准确的新算法，更好的求解和分析具体的物理过程。

关键词： 原子物理，超快激光，数值方法，氢原子，氢分子离子，电荷增强电离，B 样条

The Numerical Simulation of Atoms and Molecules in Strong Laser Fields

ABSTRACT

The Atomic, Molecular and Optical Physics is a large field of physics researches, since starting form the size of atoms and molecules, the quantum phenomena play a more important role. The traditional Newton equation can not describe the entire system any more. Instead the Schrödinger equation which uses wave functions to describe matters works. Through researching on eigenstates of atoms and molecules, people understood the reason for the stable elements existing in nature. Through researching on the time-dependent evolution of wave functions, the mechanisms of chemical reactions were revealed. However, microscopes can only observe images at the atom level even nowadays and can not catch images of electrons. In a word, it's just like taking a photograph. Only when the speed of a shutter is fast enough can it catch the image of birds flying away swiftly. For the same reason, we need to use the ultrafast laser pulses to image electrons since they move at the speed level of light. This is one of the meaning that people do researches on physics phenomena at this time scale. At the same time, there usually exist many phenomena against rules that we already know when under ultra strong external fields. For instance, the Ohm Rule no longer be applicable since breakdown happens under a strong electric field. As a result, we can discover new rules by the interaction between ultrastrong laser pulses and atoms or molecules.

Via solving the Schrödinger equations through numerical methods, we can figure out the time evolution in the ultrafast and ultrastrong dynamics, analyse wave functions to get direct comprehension and get contributions of ionization from different eigenstates through data procession.

In this article, we first get eigenstates of atoms and molecules by image time propagation method (ITPM) and then solve the time-dependent Schrödinger equation (TDSE) through the Fourier transformation method (FFT) and the Crank-Nicolson method (C-N). We applied this method to two different physics problems: First, the ionization rate relation of a hydrogen atom under corotating and counterrotating circular polarized laser pulses. Second, the Fraunhofer-like Diffraction phenomenon in the charge-resonance-enhanced Ionisation (CREI) in a hydrogen molecular ion. Next, we use the B-spline functions in variational method to solve eigenstates of atoms and molecules, and improve the Split-orbital Algorithm raised by Muller, HG *et al* [1] to achieve a fast and accurate new algorithm, which help to solve TDSE and analyse physics mechanisms better.

KEY WORDS: Atomic Molecule and Optical Physics, Ultrafast laser pulses, Strong laser pulses, Numerical Method, Hydrogen atom, Hydrogen Molecular Ion, Charge-resonance-enhanced Ionisation, B-spline Function

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The term "laser" originated as an acronym for "light amplification by stimulated emission of radiation". The first laser was built in 1960 by Theodore H. Maiman at Hughes Research Laboratories, based on theoretical work by Charles Hard Townes and Arthur Leonard Schawlow. Since then, this technology has developed rapidly and made great change to people's life such as optical disk drives, laser printers, and bar-code scanners. There are two main directions for the development of this technology: compressing the length of laser pulses and enhancing the strength of laser intensity. With the chirped pulse amplification technology (CPA), we can now achieve the short length and strong intensity laser pulses. With these laser pulses, we can study the interaction between them and atoms, molecules or plasmas. These fields are what we usually called ultrafast physics and ultrastrong physics. The electromagnetic pulses in the ultrafast physics are of the order of a picosecond (10^{-12} second) or less. It's a very important technology for observing procedures at the quantum level. For instance, the 1999 Nobel Prize in Chemistry was awarded to Ahmed H. Zewail for using ultrashort pulses to observe chemical reactions on the timescales they occur on [6]. With the ultrastrong laser pulses, the human-controlled fusion becomes possible, which is named as inertial confinement fusion.

During the study of laser-matter interactions, ionization is a very important aspect for understanding physical phenomena, which measures the electron freed from bound states around nucleuses. There are several already-known processes such as the tunnel ionization, the single-photon ionization and the multiphoton ionization. However,

the experimental techniques can only give measured results but not more detailed processes. With the computer technologies rocketing in recent ten years, the simulation of the time-dependent Schrödinger equation through numerical methods contributes mainly for phenomenological explanations of the evolution process in wave functions.

In this article, we first use some very popular numerical methods to study the interesting quantum phenomena such as ionization dependence of corotating or counterrotating laser pulses and charge-resonance-enhanced ionization (CREI). Next, we construct a faster and more accurate algorithm based on the existing ideas and algorithms. With this more powerful tool, we do research to explain the results of some fantastic quantum phenomena.

Chapter 2 Methods Based on Finite Grids

The time-dependent Schrödinger equation (TDSE) which we use is under the atomic unit. It sets $\hbar, c, m_e = 1$

$$i\frac{\partial}{\partial t}\Psi = (-\frac{\nabla^2}{2} + V)\Psi \tag{2–1}$$

2.1 Imaginary Time Propagation Method [2]

We need an initial state at the beginning, in order to calculate the time revolution of the wave function. For a general physics system, we usually take the initial state as the ground state of atoms or molecules. In numerical simulations, ground states is not normally calculated through theoretical formulas. There are two reasons: 1) Many complicate atoms and molecules don't have theoretical formula for their ground states 2) the eigenstates strongly depend on the boundary condition which are different from theoretical values.

The TDSE can be changed into the form

$$\Psi(t + \Delta t) = exp(-iH\Delta t)\Psi(t) \tag{2–2}$$

Imaginary time propagation method (ITPM) is that we give an arbitrary nonzero wave function, and then replace the Δt with $\tau = -i\Delta t$ then free propagate and renormalize it, the final stable state is the ground state of the numerical simulation. The mechanism is that the arbitrary initial state can be viewed as the combination of eigen-

states with different energies.

$$\Psi_{initial} = \sum_n a_n \Psi_n \tag{2–3}$$

After the imaginary time propagation, $exp(-iH\Delta t)$ turn into as a decay or divergent factor.

$$\Psi_{new} = \sum_n exp(-E_n \Delta t) a_n \Psi_n \tag{2–4}$$

Since the ground state always decay slowest or diverge fastest $E_0 < E_{n\neq 0}$, $exp(-E_0\Delta t) > exp(-E_{n\neq 0}\Delta t)$. For this reason, unifying the wave function after each time step can leads a larger proportion of the ground state in the new wave function. After many time steps when the energy of the wave function converges we achieve the ground state.

2.2 Real Time Propagation

I have already mentioned the propagation in the section above, but only replace Δt with τ. View from the aspect of the algorithm, these two procedure are the same. The propagation of the time dependent Schrödinger equation (TDSE) is the core of the entire procedure. There are two major methods for the time propagation of wave function: the Fourier transformation method (FFT) and the Crank-Nicolson method (C-N).

2.2.1 Fourier Transformation Method [3]

The TDSE can be expressed as

$$\begin{aligned}\Psi(t+\Delta t) =& exp[-i(T+V)\Delta t]\Psi(t) \\ =& exp[-iT\Delta t]exp[-iV\Delta t]\Psi(t)\end{aligned} \tag{2–5}$$

For the wave function in the real space $\Psi(x,t)$, we can multiply $exp(iV\Delta t)$ directly and get $\Psi_{tmp}(x,t)$. Then we transfer the new wave function into momentum space $\tilde{\Psi}_{tmp}(p,t)$, which in the math operation is Fourier transformation. Now, we can multiply the factor $exp(-iT\Delta t) = exp(-iP^2/2\Delta t)$ directly since we are now in momentum space and get $\tilde{\Psi}_{tmp}(p,t+\Delta t)$. With the inverse Fourier transformation, we get the wave function of next time step in real space $\Psi(x,t+\Delta t)$. In a different coordinates, we can use different transformation to achieve the same effect. For instance, the transformation in cylindrical coordinates is the Bessel transformation.

2.2.2 Crank-Nicholson Method [4]

This method is different from the manipulation of the Hamiltonian operator $exp(-iH\Delta t)$. In this method we can approximate the Schrödinger equation with

$$\begin{aligned}\Psi(t+\Delta t) =& exp(-iH\Delta t)\Psi(t) \\ =& \frac{1-iH\Delta t/2}{1+iH\Delta t/2}\Psi(t)\end{aligned} \tag{2–6}$$

Which can expressed in

$$(1+iH\Delta t/2)\Psi(t+\Delta t) = (1-iH\Delta t/2)\Psi(t) \tag{2–7}$$

The equation is in the $Ax = b$ model and the A matrix is in the tridiagonal format. We can easily apply the mature algorithm of the Chasing Method (CM) to solve these linear equations. [7]

2.2.3 Boundary Condition

We need to limit the calculation in a finite space since the cost of entire space is large and inaccessible. As a result, the boundary of the space is equivalent to an infinite higher potential barrier. The wave function which touches this boundary will be definitely reflected, which is different from reality. The general solution to avoid this reflection is that we force the wave function that approaches the boundary slowly converge to zero, which can be achieved by multiplying a function slowly converging to zero. Based on this idea, people raised many methods. The mostly used method is the mask function.

Suppose the length of the simulation is L. In the 10% range of the entire length close to both ends, I multiply a function

$$M(x) = \cos^{1/6}[\frac{x - x_0}{0.1L}\pi/2] \tag{2–8}$$

x_0 is the start point of the mask function. The function certain the condition $M(x = x_0) = 1$, $M(x = L) = 0$, which satisfies the requirement of the function mentioned above.

2.3 Researches Based on the Finite Grids Method

By applying the methods we mentioned above, we conduct the following research.

2.3.1 Ionization of a Hydrogen Atom in Two Sequentially Circular Polarized Strong Laser Pulses

In this research, we study the ionization of a hydrogen atom in time-delayed two circularly polarized strong laser pulses, which is inspired by [8, 9].

Ionization has been one of the most fundamental physical processes in laser-matter interaction. For atoms and molecules in strong laser fields, the ionization can be divided into multiphoton and tunneling scenarios according to the Keldysh parameter [10] $\gamma = Z\omega/nF$ with ω being the frequency, F is the amplitude of laser field and n being the main quantum number (of the nuclear charge Z). When $\gamma << 1$, the procedure is adiabatic and the tunnel ionization play a major role. When $\gamma > 1$ means that multiphoton ionization is dominant. Recently, there is theoretical and experimental results showed the strong-field ionization rate has dependence on magnetic quantum number [8, 9]. As a result, we are interesting in whether the ionization rate of counter-rotating laser pulses can exceed the ionization rate of corotating laser pulses, which is against the classical viewpoint.

The laser-atoms/molecules interaction is governed by TDSE in length gauge (atomic units are used throughout unless indicated otherwise)

$$i\frac{\partial\Psi(x,y,t)}{\partial t} = [-\frac{1}{2}(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}) + V_C(x,y) \\ +(xE\sin(\omega t) + \epsilon yE\cos(\omega t))f(t)]\Psi(x,y,t) \tag{2–9}$$

where V_C is the Coulomb potential and $f(t)$ represents the pulse envelope.

First, we use a hydrogen atom and apply a certain circular polarized laser pulse so that we can see the stable wave function after the laser is rotating clock wise. The

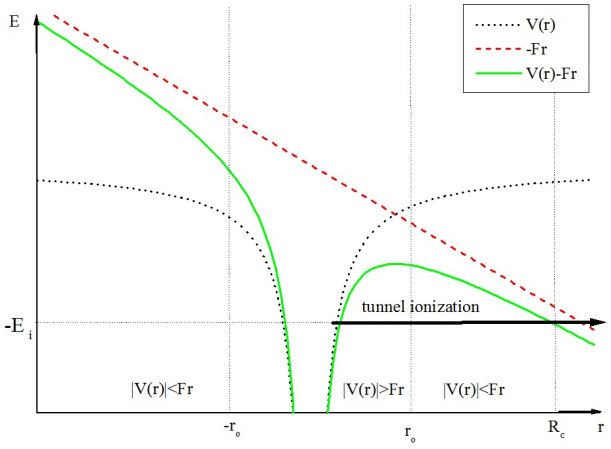


Figure 2–1: The tunnel ionization mechanism

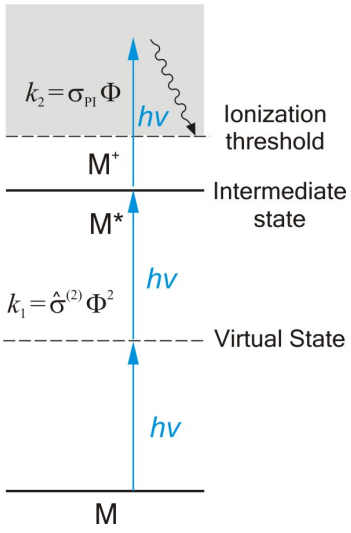


Figure 2–2: The multiphoton ionization mechanism

Coulomb potential is

$$V_{C,H} = -\frac{1}{\sqrt{x^2 + y^2 + s}} \tag{2–10}$$

Here $s = 0.64$ is the soft core parameter which is used to adjust the two-dimensional system in accord with the real ionization potential. With the laser intensity $I = 1e13W/cm^2$ and circular frequency $\omega = 0.08$ we can get the state we want.

Basically, the laser pulse excites the electron, generating a superposition of several eigenstates. A clockwise rotating electric field will induce a clockwise electron rotation, as shown in Fig. 2–3.

After we get the stable state, we used another circular polarized laser which is counterrotating or corotating. From the classical viewpoint, of course the corotating laser will lead to the higher ionization for the fast rotating frequency. Nevertheless, the phenomenon in quantum region displays different scenarios. When we varied the intensity of laser pulses, we can get a different result. The ionization rates of counterrotating laser pulses exceed those of corotating laser pulses when the laser intensity is higher

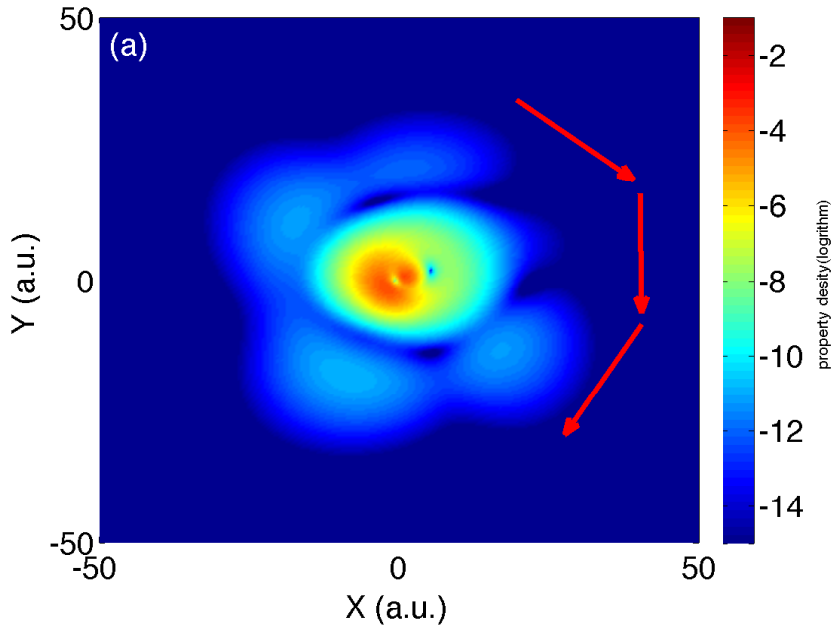


Figure 2–3: A snapshot of the probability density in ln scale at a time point, the red arrow is the direction that the wave function rotates, coordinates (X,Y) are in atomic units

than $5 \times 10^{13} W/cm^2$ and lower than $1 \times 10^{13} W/cm^2$ as showed in Fig. 2–4. To better understand the mechanism, we further diagnosed the energy spectrum for the ionization signal when the laser intensities are $2 \times 13 W/cm^2$ and $7 \times 13 W/cm^2$, as shown in Fig. 2–5 (a) and (b) . According to the Keldysh parameter $\gamma_{2\times13} = 2.9325, \gamma_{7\times13} = 1.5675$, the ionization induced by these two laser pulses may be regarded as multiphoton ionization and tunnel ionization. Indeed, Fig. 2–5 (a) shows clear discrete energy peaks. Whereas in Fig. 2–5 (b), the energy spectrum looks more continuous, indicating tunneling characters. The enhancement of ionization in multiphoton and tunnel regime are clearly different, indicating different mechanisms. By looking into the electron wave packet evolution in laser fields, we give an intuitive phenomenological picture here. In the tunneling process, the laser field suppresses the Coulomb barrier, therefore the bound electron penetrates through the barrier and get free. In the circular polarized

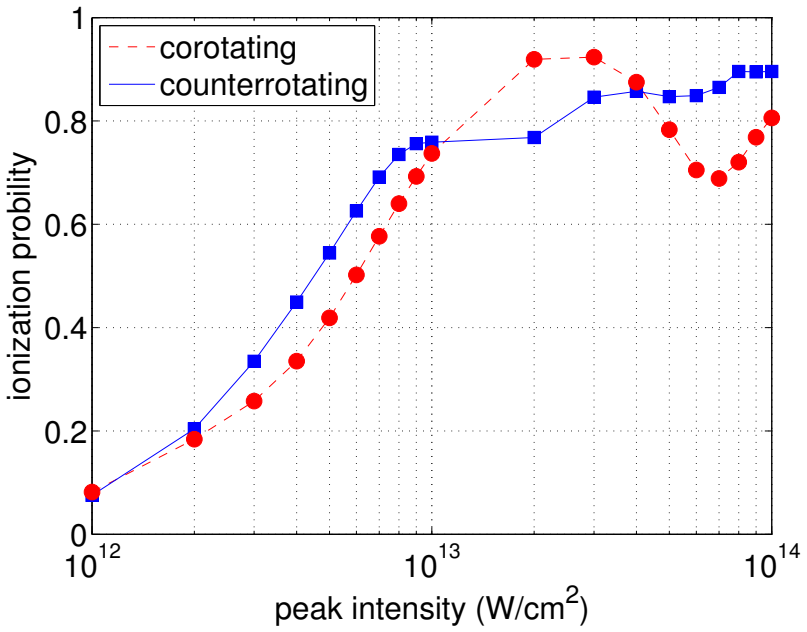


Figure 2–4: The ionization yields of counter-rotating and corotating with different peak intensity I. $\omega = 0.07$ a.u.

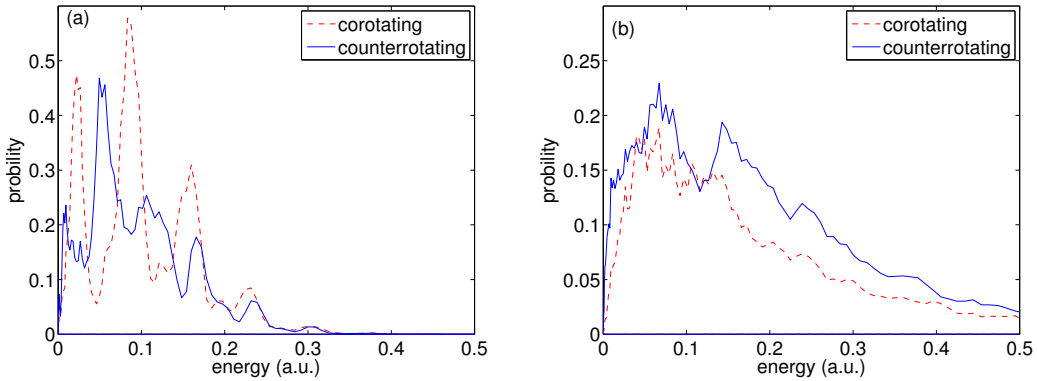


Figure 2–5: The energy spectrum of (a) $\omega = 0.07$ a.u., $I = 2e13W/cm^2$ and (b) $\omega = 0.07$a.u., $I = 7e13W/cm^2$

laser field, the electric field rotates as well as the suppressed barrier. If the electron rotation, induced by the first pump pulse, counterrotating with the second laser electric rotation, the electron will see a suppressed barrier more frequently. On the other word, the electron actually meets a gate more frequently and tunnels out more easily. If laser pulses are corotating, there is a delay in the rotation and the electron will miss the gate.

The scenario is very different in the multiphoton regime, in which the electron absorbs several photons and escape from the nuclei. When the electron rotating direction synchronise the laser electric field, the frequency of the electron is accelerated and absorb the extra photon to get ionized. In contrast, the counterrotating laser pulses will decrease the frequency of the electron and lead to low ionization rates.

The result that counterrotating ionization rate is higher then corotating ionization rate may due to the two dimensional model. There is a three dimensional calculation starting from specific m state published by Jarosław H. Bauer *et al* [11] when we just finished our manuscript and they give the similar results.

To summarise, we have studied the ionization of a hydrogen atom in two sequential circularly polarized laser pulses. In the multiphoton ionization regime, corotating laser pulses preferentially ionize the atom with a rotating electron wave packet. However, in tunneling regime, when the electron rotate against the laser electric field, it sees a suppressed Coulomb potential with a higher frequency, therefore more electron wave packet may tunnel out. The difference of the ionization probabilities induced by counterrotating and corotating laser pulses may be enlarged when more bound electron wave packet take part in the rotation. This phenomena is possible to be observed with the current laser technology.

2.3.2 Fraunhofer-like Diffracted Lateral Photoelectron Momentum Distributions of H_2^+ in Charge-Resonance-Enhanced Ionization in Strong Laser Fields

The TDSE used for describing a H_2^+ in the cylindrical coordinate.

$$i\frac{\partial\Psi(z,\rho;t)}{\partial t} = [-\frac{1}{2}\frac{\partial^2}{\partial z^2} - \frac{1}{2}(\frac{1}{\rho}\frac{\partial}{\partial\rho} + \frac{\partial^2}{\partial\rho^2}) - \sum_{s=\pm1}\frac{1}{\sqrt{\rho^2+(z-sR/2)^2}} + zE(t)]\Psi(z,\rho;t) \tag{2–11}$$

where R is the internuclear distance. Here we use the Born-Oppenheimer approximation (BOA) [12] in our model, which in simple is fixing the internuclear distance R during the calculation since the laser pulses is very short. We only apply linearly polarized laser fields E which is defined as $E = -\frac{1}{c}\frac{\partial A(t)}{\partial t}$, with c being the light velocity. The vector potential is written as

$$A(t) = A_0\cos(\omega t)\cos^2(\pi t/\tau), -\tau/2 < t < \tau/2 \tag{2–12}$$

The laser parameters are $\lambda = 3000nm, I = 1.5e14W/cm^2, L = 2T$. T is the period laser pulses. We obtained the initial state by ITPM and propagated the wave packet by C-N. The simulation box is 3000×10000 grids in the ρ−z plane with the step $\Delta\rho = \Delta z = 0.3$. We kept propagating the wave function until the bound and freed states clearly separated in space so that we can calculate the physical property only related to the ionized part. The wave packet in the area $r = \sqrt{\rho^2+z^2} < 50$ a.u. was removed and Fourier transformed the remaining wave function along the z axis and Hankel transformed it along the ρ axis to get the spectrum.

$$\tilde{\psi}(p_\rho, p_z) = \sqrt{2\pi}\int\int \rho d\rho\, dz\psi(\rho,z;t_f)e^{-ip_z z}J_0(p_\rho\rho) \tag{2–13}$$

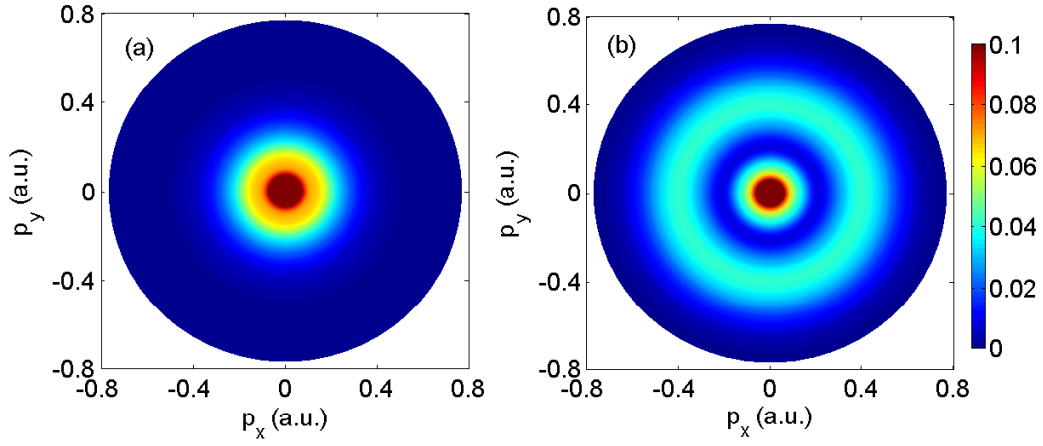


Figure 2–6: The LPMD at the internuclear distance of (a) 6 a.u. and (b) 9 a.u.

where J_0 is the the first kind zeroth order Bessel function. Finally, the Lateral photoelectron momentum distributions (LPMD) [13] is $\int |\tilde{\psi}(p_\rho, p_z)|^2 dp_z$.

Fig. 2–6 (a) and (b) show the LPMD at the internuclear distances of 6 and 9 a.u., respectively. The LPMD shows a cusplike structure with a maximum at $p_\rho = 0$ at internuclear distances shorter than 7 a.u., which is very similar to atomic cases. However, the LPMD is distinct at the internuclear distance of 9 a.u.: the central part is surrounded by a ring. Fig. 2–6 (b) looks very much like the Fraunhofer diffraction which displays an airy disk and a secondary ring. More than one ring out of the central disk are faintly seen (not shown) at the internuclear distance larger than 14 a.u..

We named p_m as the momentum radius with the minimum possibility between the central disk and the secondary ring, and plotted p_m as a function of the internuclear distance R in Fig. 2–7 (a). There is no secondary ring clearly seen when $R \leq 7$ a.u..

We reproduced the two peaks of change-resonance enhanced ionization (CREI) [14] with the above given laser parameters, as shown by the blue solid curve in Fig. 2–7 (b). The red dashed curve in Fig. 2–7 (b) represents the partial ionization probability where only the photoelectron with $p_\rho < p_m$ is counted. It is clearly revealed that the

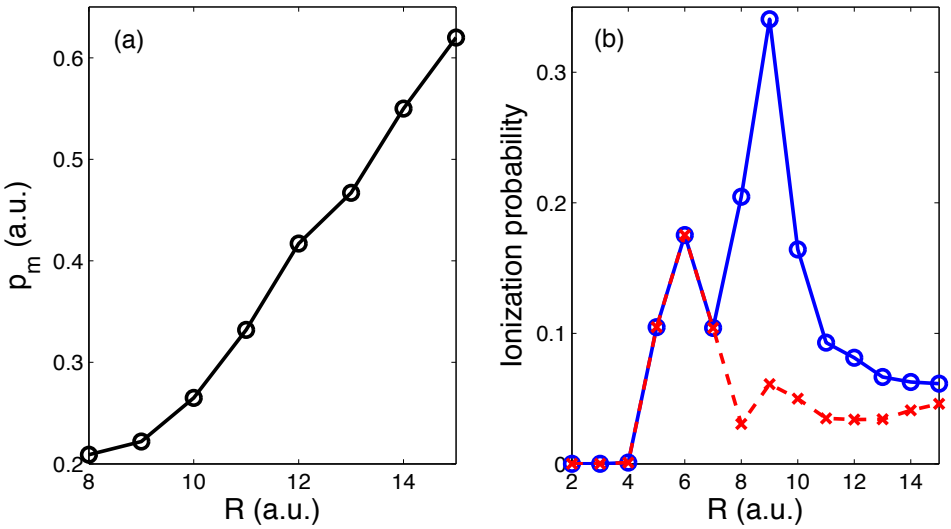


Figure 2–7: (a) The momentum radius with the minimum possibility between the central disk and the secondary ring as a function of the internuclear distance. (b) The total (blue solid line) and partial (red dashed line) ionization probability as a function of the internuclear distance.

second peak is mainly contributed by the secondary ring in Fig. 2–6 (b).

The Fraunhofer-like diffraction pattern and the increasing of p_m with the increasing of R suggest that the ring structure in Fig. 2–6 (b) is due to the diffraction in the diatomic Coulomb potential.

In order to compare our results with the standard Fraunhofer diffraction, we adopted a static electric field instead of an linear polarized laser field for H_2^+ . Our simulations showed more rings can be observed with a larger internuclear distance. Therefore, we set $R = 14$ a.u. for the following calculations. The electric amplitude is 0.053 a.u. ($1 \times 10^{14} W/cm^2$) and points to $+z$ axis. Fig. 2–8 (a) is the addition of several electron distribution snapshots at different time so that the time-dependent fluctuation can be smeared out and a steady electron distribution can be obtained. We can clearly see several stripes in Fig. 2–8 (a). The red line in Fig. 2–8 (a) at $z = -20$ a.u. was

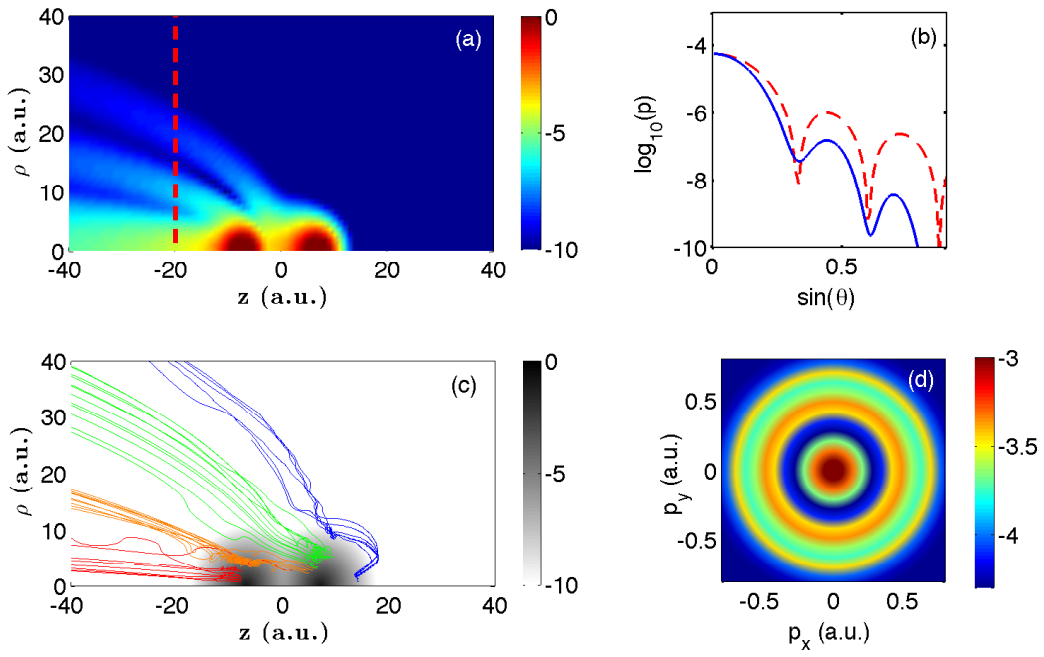


Figure 2–8: (a) The electron distribution after adding several snapshots at different time. (b) The blue solid curve of the electron distribution at $z = -20$ a.u.. The red dashed curve expresses the scaled Fraunhofer diffraction formula. (c) The Bohmian trajectories of possibility on spurs. (d) The finally momentum distribution calculated by Bohmian theories.

plotted in Fig. 2–8 (b) as a function of $\sin\theta$ by the blue solid curve. Here, θ is defined as the angle for the emitted electron direction to the molecular axis. For a standard Fraunhofer diffraction by a circular aperture, the diffracted light intensity distribution is formulated $\propto (\frac{2J_1(q\sin\theta)}{q\sin\theta})^2$ where q is the product of a light wave number and the aperture radius. To make a comparison, we presented an analytical formula inFig. 2–8 (b) by the red dashed curve, which was scaled in order to make two curves overlap at $\theta = 0$ so q was set as 11.7. The agreement of these two curves at the minimum positions convinces us that the photoelectron spatial distribution roughly follows the rule of Fraunhofer diffraction by a circular aperture.

So when and how the electron is diffracted to construct rings either in spatial or momentum distribution? To explore that, we showed semi-classical calculation about Bohmian trajectories [15] in Fig. 2–8 (c). The trajectories (z, ρ) were obtained by solving the following equation

$$\frac{dz}{dt} = v_z(z, \rho; t), \quad \frac{d\rho}{dt} = v_\rho(z, \rho; t), \tag{2–14}$$

The velocity fields $(v_z(z, \rho; t), v_\rho(z, \rho; t))$ are calculated by the phase gradient of the wave function.

$$v_z(z, \rho; t) = \mathrm{Im}[\frac{1}{\psi}\frac{\partial \psi}{\partial z}], \; v_\rho(z, \rho; t) = \mathrm{Im}[\frac{1}{\psi}\frac{\partial \psi}{\partial \rho}] \tag{2–15}$$

The red, orange, green and blue curves precisely display show how electrons on different spurs flow out from the nuclei. Initially, the electron equally distribute on two nuclei, as the grey clouds shown in Fig. 2–8 (c). When the electric field is acted, the electron moves along the direction from right to left. Most of red group curves in Fig. 2–8 (c) depart from the left nuclear side, which is similar to the atomic tunneling ionization. The orange group curves depart from the right nucleus are scattered by the interatomic Coulomb field and the left nucleus, ultimately joining the secondary ring. The blue and green group curves are already diffracted before they approach the left nucleus and contribute to the third and fourth ring. The final momentum distribution given by the Bohmian theory is shown in Fig. 2–8 (d), which also show the same pattern as Fig. 2–6 (b).

The Bohmian trajectories in Fig. 2–8 (c) prove that before the electron approaches the left nucleus, the radial momenta for different trajectories have already been established. Therefore, we conclude the R-dependent LPMD is strongly related to the struc-

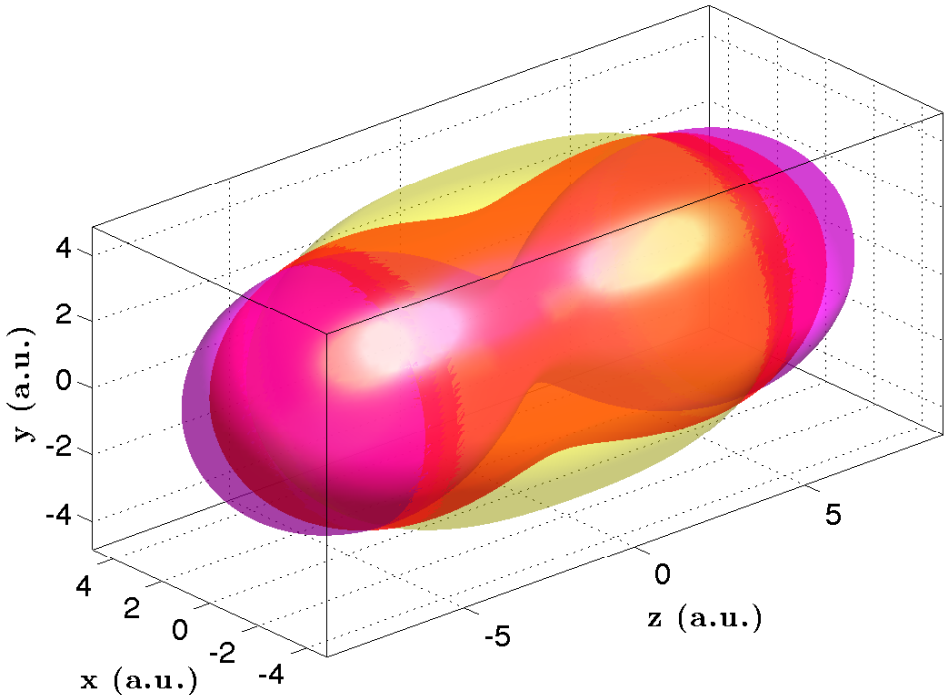


Figure 2–9: Contour planes (the isosurface value is -0.35 a.u.) for the internuclear distance R=6 (yellow), 9 (red), 11 (magenta) a.u..

ture of the interatomic Coulomb potential. The three contour planes for the Coulomb potential at $R = 6, 9, 11$ a.u. with the isosurface value being -0.35 a.u are plotted for the purpose of better explaining. The contour plane between two nuclei works as a tube, inside which the potential is lower therefore the electron may tunnel through. The radius of the tube is larger for a smaller internuclear distance. When the internuclear distance is less than 6 a.u., the tube is wide thus almost no diffraction happens and the LPMD has a similar shape with atomic cases. For a larger internuclear distance, the tube shrinks more. When the electron tunnels through it, it is analogous to pass a small aperture and the diffraction effect becomes much more important. A larger internuclear distance deduces a smaller tube connecting two nuclei, which leads to a

larger diffraction angle. This phenomenological explanation is consistent with the simulation results shown in Fig. 2–7 (a). With this, we explain why the Fraunhofer-like diffraction pattern in the LPMD is most prominent for the internuclear distance where the second peak of CREI locates.

In conclusion, the LPMD presents a central disk surrounded by one or several rings when the internuclear distance of H_2^+ is critical that the CREI is most prominent. The enhanced ionization probability is mainly donated by the electron in the rings. By comparing the LPMD with a standard Fraunhofer diffraction by a circular aperture, we explore the diffraction process: the electron is diffracted when it moves from the up-field core toward the down-field core by penetrating the stretched molecular interatomic Coulomb potential, which works as a circular aperture. Those results was published on Phys. Rev. A 92, 063803 [16].

2.4 Summary

In this chapter, we introduce the popular method used in solving TDSE and we apply this method to the phenomena of ionization dependence on corotating laser and counterrotating laser and the phenomena along with charge-resonance enhanced ionization.

Chapter 3 The Algorithm Using B-splines for TDSE

In the previous chapter we already known the ITPM for solving ground states and FFT, C-N for time propagation. However, those methods has some deflects. Firstly, the ITPM will lead a difference on ground state $e^{-i\Delta t}$ when change from imaginary time to real time. This will lead the auto ionization during the propagation. Furthermore, this method can only get one eigenstate once a time, cannot get specific angular number eigenstate and the eigenstates with higher energy will lead to great error. FFT, C-N is limited by the accuracy of derivative, the calculation time, no direct view of the ionization contribution from different angular number eigenstates. Thus, a better algorithm is in demand. In this chapter we will first introduce the existing algorithms and codes. Secondly, the process about how to construct the algorithm using B-splines is presented.

3.1 Introduction to B-spline Function

The B-spline is also called the basis spline [17]. It is a piecewise linear polynomial which can lower the cost of overlap integral. Moreover, its first-order derivative and second-order derivative can be expressed in the combination of its self, which reduces the error caused by derivative approximation. This is the recursion formula for B-splines.

$$B_{i,1} := \begin{cases} 1 & \text{if } t_i \leq x < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{3–1}$$

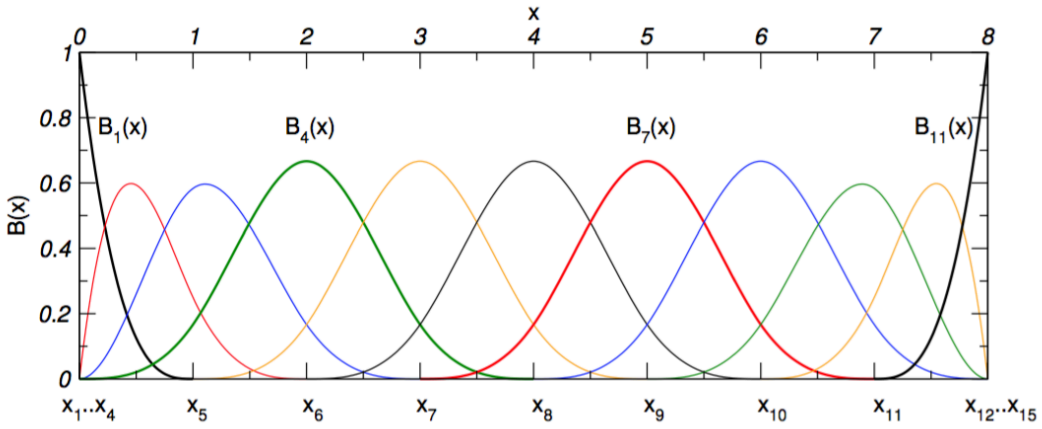


Figure 3–1: Full set of 11 B-splines of order $k = 4$ defined for the knot sequence $\{x_i\} = \{0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 8, 8, 8\}$ [5]

$$B_{i,k} := \frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(x) + \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(x) \tag{3–2}$$

First-order derivative and second-order derivative analytical formula for B-splines.

$$\frac{dB_{i,k}}{dx} = (k-1)\left(\frac{B_{i,k-1}(x)}{t_{i+k-1} - t_i} - \frac{B_{i+1,k-1}(x)}{t_{i+k} - t_{i+1}}\right) \tag{3–3}$$

$$\frac{d^2 B_{i,k}}{dx^2} = (k-1)\left(\frac{\frac{dB_{i,k-1}(x)}{dx}}{t_{i+k-1} - t_i} - \frac{\frac{dB_{i+1,k-1}(x)}{dx}}{t_{i+k} - t_{i+1}}\right) \tag{3–4}$$

The number of point needs for one B-spline function equals to the order of function plus one $m = k + 1$. It is defined over a domain $t_0 \leq x \leq t_m$. The points where $x = t_j$ are known as knots or break-points. An intuitional picture Fig. 3–1 is plotted to illustrate.

In the expansion of the radial wave function, every internal point needs a B-spline function and the number of B-splines in the set is $n = l + k - 1$, starting from the left and to the right. Here l is the number of radial grids we divid the interval.

3.1.1 Gauss Quadrature

The method we used for integral is Gauss Quadrature. This method is completely accurate when applied to linear polynomials. n points are required for the exact result for polynomials of degree $2n-1$. x_i is the point in $[0,1]$ and ω_i is the weight, so the integral between $[a,b]$ is

$$\int_a^b f(x) = \frac{b-a}{2}\sum_{i=1}^{n}\omega_i f(\frac{b-a}{2}x_i + \frac{a+b}{2}) \tag{3–5}$$

3.2 Solving for Eigenstates of Atoms and Molecules

The idea of using B-splines for solving the eigenstate is raised by Fernando Martín *et al* [18, 19].

3.2.1 Eigenstates for Atom Cases

The Schrödinger equation of atom cases is

$$[-\frac{1}{2}\nabla^2 + V(r)]\psi(r) = E\psi(r) \tag{3–6}$$

Here, $V(r)$ is the potential only depending on the radius variable. Due to the separation of variables, we can expand the wave function into the radial part and the angular part. Since the eigenfunction of angular part is spherical harmonics the only equation we need to solve is the radial part equation.

$$\psi_{nlm}(r) = \frac{U_{nl}}{r}Y_l^m(\theta,\phi) \tag{3–7}$$

$$[-\frac{d^2}{2dr^2} + \frac{l(l+1)}{2r^2} + V(r)]U_{nl}(r) = E_{nl}U_{nl}(r) \tag{3–8}$$

The boundary conditions require at $r = 0$, $U_{nl} = 0$ and $r = r_{max} \approx \infty$, $U_{nl} = 0$

$$U_{nl}(r = 0) = 0 \leftarrow \frac{U_{nl}}{r}(r = 0) < \infty \tag{3–9}$$

We now expand the $U_{nl}(r)$ with the B-splines so that

$$U_{nl}(r) = \sum_{i=1}^{N} c_i B_i(r) \tag{3–10}$$

N is the size of basis.

Then we substitute the Eq. 3–10 into Eq. 3–8 and project it on $\frac{B_i(r)}{r}$. In the issue, we achieved the linear equations Eq. 3–11 to solve.

$$\mathbf{H} \cdot \mathbf{c} = E\, \mathbf{S} \cdot \mathbf{c} \tag{3–11}$$

The elements of the **H** and **S** are

$$\begin{aligned} \mathbf{H}_{ij} = -\frac{1}{2}\int_0^{r_{max}} B_i \frac{d^2}{dr^2} B_j(r)dr + \frac{l(l+1)}{2}\int_0^{r_{max}} \frac{B_i(r)B_j(r)}{r^2}dr \\ + \int_0^{r_{max}} B_i(r)V(r)B_j(r)dr \end{aligned} \tag{3–12}$$

$$\mathbf{S}_{ij} = \int_0^{r_{max}} B_i(r)B_j(r)dr \tag{3–13}$$

With the full use of B-splines characteristic Fig. 3–2, we can reduce the overlap integral $\int_0^{r_{max}} B_i \frac{d^2}{dr^2} B_j(r)dr$ and $\int_0^{r_{max}} B_i(r)B_j(r)dr$ to the interval of [index_max, index_min+k]

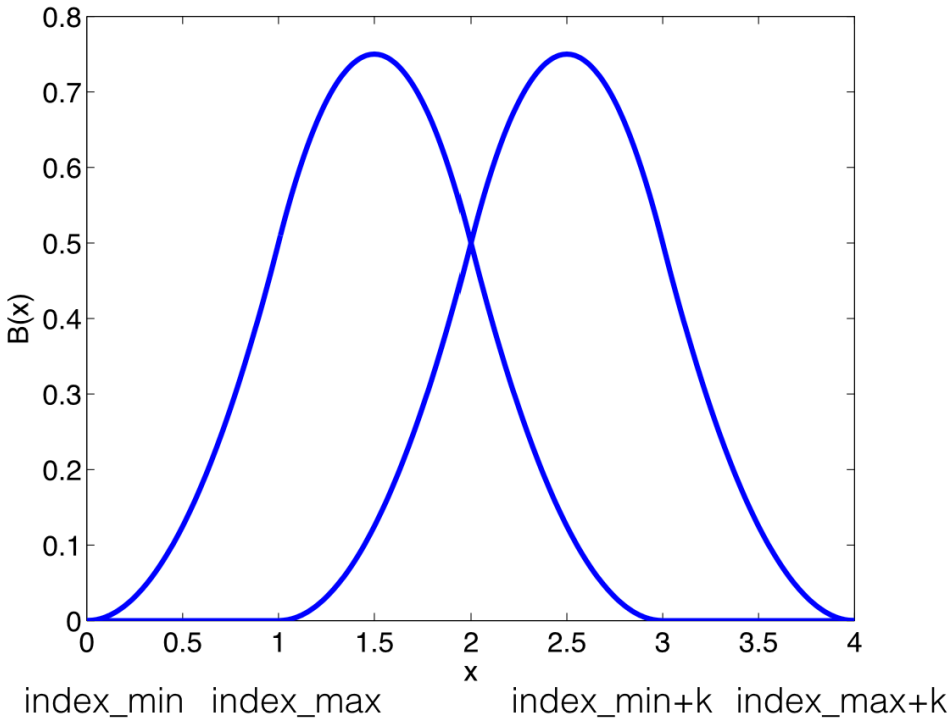


Figure 3–2: Overlap integral reduction

In order to solve Eq. 3–11, multiplied it by $\mathbf{S}^{-1}$

$$\mathbf{S}^{-1}(\mathbf{H} - E\mathbf{S}) \cdot \mathbf{c} = 0 \tag{3–14}$$

$$(\mathbf{S}^{-1}\mathbf{H} - E) \cdot \mathbf{c} = 0 \tag{3–15}$$

Set the $\mathbf{X}$ satisfied

$$\mathbf{SX} = \mathbf{H} \tag{3–16}$$

$$\mathbf{X} = \mathbf{S^{-1}H} \tag{3–17}$$

Table 3–1: Eigen energies of hydrogen (s states) computed with B-splines ($k = 9,\ r_{max} = 100$ a.u., $N = 100$, linear sequence) and compared with the exact values

n	E_n	$E_n^{exact} = -1/2n^2$	ΔE
1	-0.499999999996724	-0.500000000000000	3.27599058991268e-12
2	-0.125000000000001	-0.125000000000000	9.99200722162641e-16
3	-0.055555555555555	-0.0555555555555556	5.55111512312578e-16

Substitute **X** in the previous equation, we get

$$\mathbf{X} \cdot \mathbf{c} = E\mathbf{c} \tag{3–18}$$

So the eigenstate and its energy turned into the problem of eigenvector and eigenvalue of **X**. When we apply this algorithm to code, we first solve the linear equations Eq. 3–17 and then find the eigenvalue of **X**. So we calculated the eigen energies of hydrogen (s states) with B-splines ($k = 9,\ r_{max} = 100$ a.u., $N = 100$, linear sequence) and compared with exact values in Tab. 3–1. The energy difference increases when the main quantum number gets bigger, as shown in Fig. 3–3.

3.2.1.1 Unification Procedure of Atom Cases

Since the eigenvectors which we calculate through Eq. 3–18 do not necessarily satisfy the condition that their norm equal to 1. Hence, unification procedure is an indispensable step for calculation. We suppose now there is a factor that can help unified the

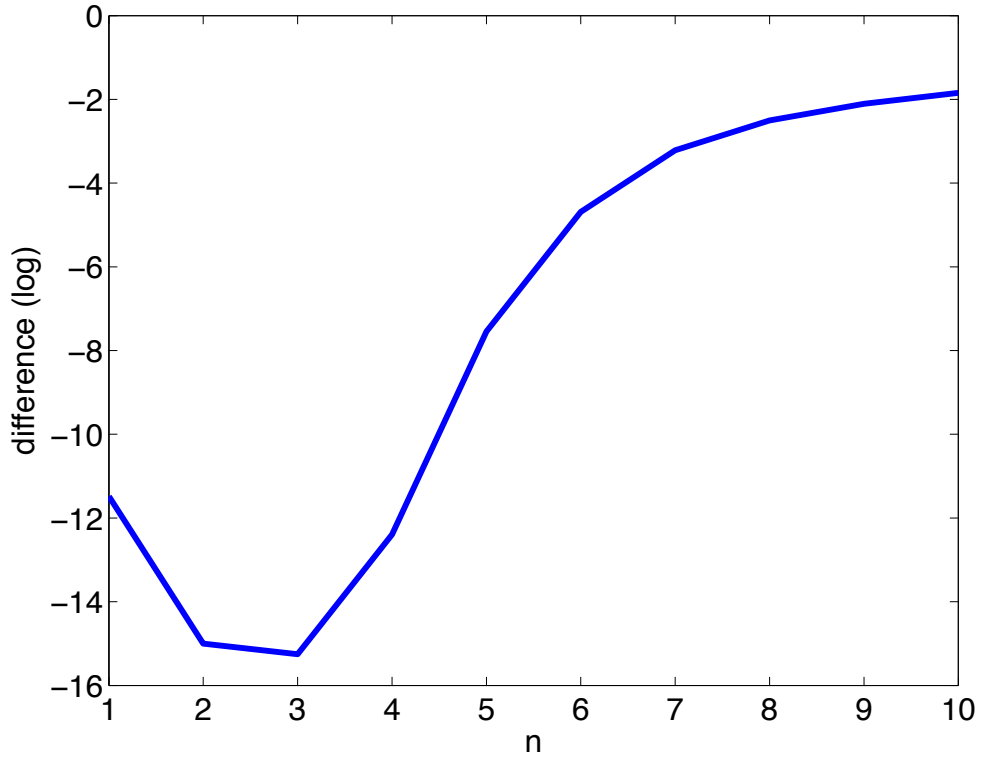


Figure 3–3: The difference between theory and calculation of the hydrogen energy versus main quantum number n

entire wave function and deduce the formula to compute the factor f.

$$\begin{aligned}
&\int_0^\infty \int_0^{2\pi} \int_0^\pi \psi^* \psi r^2 \sin\theta d\theta d\phi dr \\
&= \int_0^\infty \int_0^{2\pi} \int_0^\pi \sum_{il} c_{il} B_i^k(r) Y_l^0(\theta,\phi) \sum_{i'l'} c_{i'l'} B_{i'}^k(r) Y_{l'}^0(\theta,\phi) \sin\theta d\theta d\phi dr \\
&= \int_0^\infty \sum_{il} c_{il} B_i^k(r) \sum_{i'l'} c_{i'l'} B_{i'}^k(r) dr \int_0^{2\pi} \int_0^\pi Y_l^0(\theta,\phi) Y_{l'}^0(\theta,\phi) \sin\theta d\theta d\phi \\
&= f^2 \int_0^\infty \sum_{il} c_{il} B_i^k(r) \sum_{i'l} c_{i'l} B_{i'}^k(r) dr = 1
\end{aligned} \tag{3–19}$$

$$sum = \sum_{il} c_{il} \sum_{i'l} c_{i'l} \int_0^\infty B_i^k(r) B_{i'}^k(r) dr, f = \sqrt{\frac{1}{sum}} \tag{3–20}$$

After calculation of the unification factor, we need to multiply it to the origin eigenvector.

3.2.1.2 Plot

To test the rightness of our results and for direct view of the wave function evolution, we need to reconstruct from the B-splines index c to the wave function ψ. Along the $\theta = 0, \phi = 0$ direction, the wave function is in the formula of

$$\psi = \sum_{il} c_{il} \frac{B_i^k(r)}{r} \sqrt{\frac{(2l+1)}{4\pi}} \tag{3–21}$$

The comparative theoretical formula is

$$R_{40} = \frac{1}{4}(1 - \frac{3}{4}r + \frac{1}{8}r^2 - \frac{1}{192}r^3)e^{-r/4} \tag{3–22}$$

Of course if you like to test other eigenstates you can test them. The result of accuracy is showed in Fig. 3–4 and it's evident that the wave function fit the theoretical formula quite well.

3.2.2 Lapacke Help

During my calculation, I applied the mature linear algebra package Lapacke for solving general linear equations and the eigenvector calculation. Of course you can calculate these through you own code (I already try it). However, due to the efficiency using the existing high-performance library is recommended.

(1) DGESV(N, NRHS, A, LDA, IPIV, B, LDB, INFO)

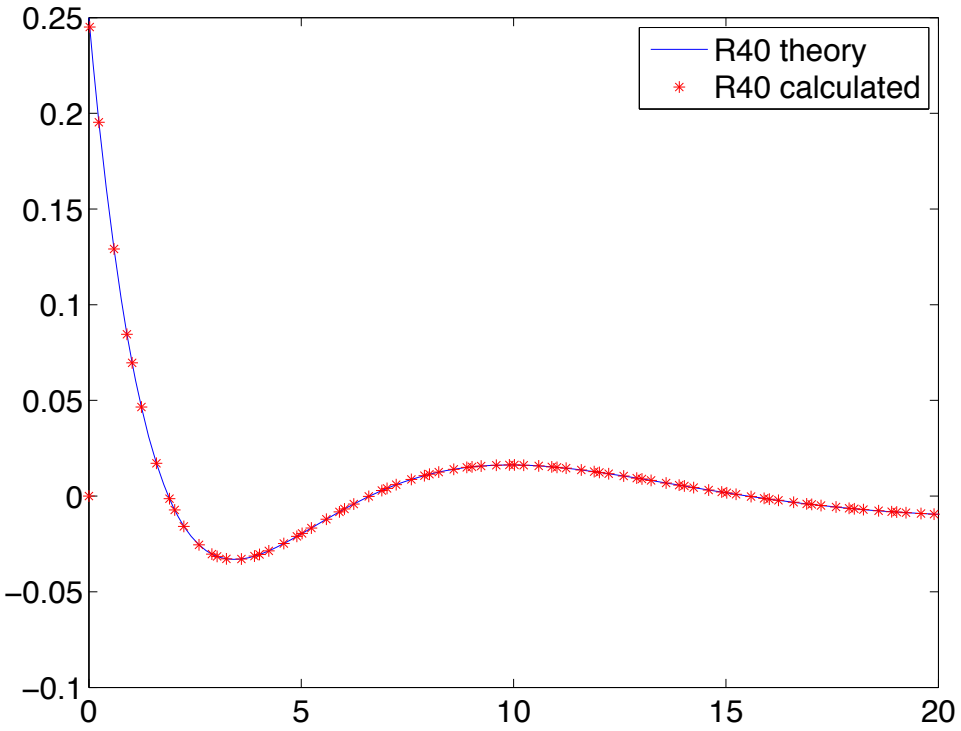


Figure 3–4: theoretical and calculated R_{40} with B-splines ($k = 9$, $r_{max} = 100$ a.u., $N = 100$, linear sequence)

```
1  Solve the general linear equation problem of format A*X=B, A(N*N), X, B(N*NRHS), A=P*L*U;
2
3  N: the order of matrix A
4
5  NRHS: the number of columns of the matrix B
6
7  A: input: matrix, output: L, U
8
9  LDA: (leading dimension of A) the row of matrix A
10
11 IPIV: pivot indices define the permutation matrix P
12
13 B: input: matrix, output: if INFO=0, X
14
15 LDB: same definition
16
17 INFO: =0 successful exit; =-i, the i-th argument had and illegal value; =i, U(i,i) =0, U is singular and
```

```
    can not compute solution.

The C interface is in the structure as follows:
lapack\_int LAPACKE\_dgesv( int matrix\_layout, lapack\_int n, lapack\_int nrhs,double* a, lapack\_int
    lda, lapack\_int* ipiv, double* b, lapack\_int ldb)
```

(2) DGEEV(JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR, LDVR, WORK, LWORK, INFO)

```
Solve the  general eigenvector problem of format A*X=B, A(N*N), X, B(N*NRHS);

JOBL: 'V' $\to$ left eigenvectors of A are computed; 'N' $\to$ left eigenvectors of A are not computed

JOBR: 'V' $\to$ right eigenvectors of A are computed; 'N' $\to$ right eigenvectors of A are not computed

N: same

A: input: matrix, output: overwritten

LDA: same

WR, WI: dimension N, contains real and imaginary part of the computed eigenvalues

VL: LDVL*N left eigenvectors, If the j-the eigenvalue is real u(j)=VL(:,j), if is complex, u(j)=VL(:,j)+i
    *VL(:,j+1), u(j+1)=VL(:,j)-i*VL(:,j+1)

LDVL: JOBVL='V', LDVL>=N

VR: right left eigenvectors

LDVR: same definition

The C interface is in the structure as follows:
lapack\_int LAPACKE\_dgeev( int matrix\_layout, char jobvl, char jobvr,lapack\_int n, double* a, lapack\
    _int lda, double* wr, double* wi, double* vl, lapack\_int ldvl, double* vr,lapack\_int ldvr )
```

3.2.3 Eigenstates for Molecule Cases

The difference between molecule cases and atom cases is that the Coulomb potential no longer only depends on the radius variable but also depends on the angular variable,

which can be easily observed from it's Hamilton operator.

$$H_{el} = -\frac{1}{2}\nabla^2 - \frac{1}{|\mathbf{r}-\mathbf{R_A}|} - \frac{1}{|\mathbf{r}-\mathbf{R_B}|} + \frac{1}{|\mathbf{R_A}-\mathbf{R_B}|} \tag{3–23}$$

Since there exists a angular portion in the Hamilton, so the expansion of a wave function should include the angular part. For simple, we start from linear polarized laser pulses case in which the magnetic quantum number m conserves.

$$\psi(r,\theta,\phi) = \sum_{l=0}^{l_{max}} \sum_{i=0}^{N_l} c_{il}^n \frac{B_i^k}{r} Y_l^m(\theta,\phi) \tag{3–24}$$

3.2.3.1 $m = 0$ Case

Set $m = 0$ to simplify the problem. The elements of $\mathbf{H}, \mathbf{S}$ are

$$H_{il,i'l'} = \int_0^{r_{max}} \int_0^{\pi} \int_0^{2\pi} \frac{B_i^k(r)}{r} Y_l^0(\theta,\phi) H_{el} \frac{B_{i'}^k}{r} Y_{l'}^0(\theta,\phi) r^2 \sin(\theta) dr d\theta d\phi \tag{3–25}$$

$$S_{il,i'l'} = \delta_{ll'} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) dr \tag{3–26}$$

The linear equation we need to solve is still in the same matrix format

$$\mathbf{H} \cdot \mathbf{c} = E\, \mathbf{S} \cdot \mathbf{c} \tag{3–27}$$

Here c has the angular index l since it's angular entangled

$$\mathbf{c} = c_i^l \tag{3–28}$$

What is different in molecule cases is that we need to integrate over the entire space.

$$\begin{aligned} &< i,l| -\frac{1}{2}\nabla^2 |i',l' > \\ &= \int_0^{r_{max}} \int_0^{\pi} \int_0^{2\pi} \frac{B_i^k(r)}{r} Y_l^0(\theta,\phi)(-\frac{1}{2})\nabla^2[\frac{B_{i'}^k}{r}Y_{l'}^0(\theta,\phi)]r^2\sin(\theta)drd\theta d\phi \end{aligned} \tag{3–29}$$

We can derive the analytic formula of the second-order derivative using the characteristic of spherical harmonics.

$$\nabla^2[\frac{B_{i'}^k}{r}Y_{l'}^0(\theta,\phi)] = \frac{1}{r}\frac{\partial^2 B_{i'}^k(r)}{\partial r^2}Y_{l'}^0(\theta) + \frac{B_{i'}^k(r)}{r^3}(-l'(l'+1)Y_{l'}^0(\theta) \tag{3–30}$$

Substitute this in to the equation above, we can simplify the integral

$$\begin{aligned} &< i,l| -\frac{1}{2}\nabla^2 |i',l' >= \int_0^{r_{max}} \int_0^{\pi} \int_0^{2\pi} \frac{B_i^k(r)}{r} Y_l^0(\theta,\phi)(-\frac{1}{2})\cdot \\ &[\frac{1}{r}\frac{\partial^2 B_{i'}^k(r)}{\partial r^2}Y_{l'}^0(\theta) + \frac{B_{i'}^k(r)}{r^3}(-l'(l'+1)Y_{l'}^0(\theta)]r^2\sin(\theta)drd\theta d\phi \\ &= -\frac{1}{2}(\int_0^{r_{max}} B_i^k(r)\frac{\partial^2 B_{i'}^k(r)}{\partial r^2}dr \int_0^{\pi}\int_0^{2\pi} Y_l^0(\theta,\phi)Y_{l'}^0(\theta)\sin(\theta)d\theta d\phi \\ &- l'(l'+1)\int_0^{r_{max}} \frac{B_i^k(r)B_{i'}^k(r)}{r^2}dr \int_0^{\pi}\int_0^{2\pi} Y_l^0(\theta,\phi)Y_{l'}^0(\theta)\sin(\theta)d\theta d\phi) \\ &= -\frac{1}{2}\delta_{ll'}(\int_0^{r_{max}} B_i^k(r)(\frac{\partial^2 B_{i'}^k(r)}{\partial r^2} - l'(l'+1)\frac{B_{i'}^k(r)}{r^2})dr) \end{aligned} \tag{3–31}$$

When we deal with the integral of the Coulomb potential we can not integral the whole space together. In order to save the cost and achieve the analytic formula, we expand the potential into the combination of spheric harmonics.

The method we use here is called Laplace expansion

$$\frac{1}{|\mathbf{r}-\mathbf{R}|}=\sum_{l=0}^{\infty}\frac{4\pi}{2l+1}\sum_{m=-l}^{l}(-1)^m\frac{r_<^l}{r_>^{l+1}}Y_l^{-m}(\theta,\phi)Y_l^m(\theta',\phi') \tag{3–32}$$
$$r_<=min(r,R/2),r_>=max(r,R/2),\mathbf{r}=(r,\theta,\phi),\mathbf{R}=(R/2,\theta',\phi')$$

l on the superscript of r is the power of it. Here we set the molecule axis as z axis so that $(\theta',\phi')=(0,0)$ or $(\pi,0)$

In our case $m=0$

$$Y_l^m=(-1)^{|m|}\sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}e^{im\phi}P_l^m(\cos\theta) \tag{3–33}$$
$$P_l^m(x)=(1-x^2)^{|m|/2}(\frac{d}{dx})^{|m|}P_l(x) \tag{3–34}$$
$$P_l(x)=\frac{1}{2^l l!}(\frac{d}{dx})^l(x^2-1)^l \tag{3–35}$$
$$Y_l^0(0,0)=\sqrt{\frac{(2l+1)}{4\pi}}\frac{1}{2^l l!}\cdot(\frac{d}{dx}^l)(x^2-1)^l|_{x=1} \tag{3–36}$$

Here I substitute $x-1=\xi$

$$\frac{d}{dx}^l(x^2-1)^l|_{x=1}=\frac{d}{d\xi}^l\xi^l(\xi+2)^l|_{\xi=0}=l!2^l \tag{3–37}$$
$$Y_l^0(0,0)=\sqrt{\frac{(2l+1)}{4\pi}} \tag{3–38}$$
$$Y_l^0(\pi,0)=(-1)^l\sqrt{\frac{(2l+1)}{4\pi}} \tag{3–39}$$

Substitute the predigested results in Eq. 3–32 and we get a easy form of the expansion

$$\frac{1}{|\mathbf{r}-\mathbf{R_A}|}=\sum_{l=0}^{\infty}\frac{4\pi}{2l+1}\frac{r_<^l}{r_>^{l+1}}Y_l^0(\theta)\sqrt{\frac{(2l+1)}{4\pi}} \tag{3–40}$$

$$\frac{1}{|\mathbf{r}-\mathbf{R_B}|}=\sum_{l=0}^{\infty}\frac{4\pi}{2l+1}\frac{r_<^l}{r_>^{l+1}}Y_l^0(\theta)\sqrt{\frac{(2l+1)}{4\pi}}(-1)^l \tag{3–41}$$

So the static potential integral is

$$\begin{aligned}<i,l|\frac{1}{|\mathbf{r}-\mathbf{R_A}|}|i',l'>=&\sum_{n=0}^{\infty}\sqrt{\frac{4\pi}{(2n+1)}}\int_0^{r_{max}}B_i^k(r)B_{i'}^k(r)\frac{r_<^n}{r_>^{n+1}}dr\cdot\\&\int_0^{\pi}\int_0^{2\pi}Y_l^0(\theta)Y_n^0Y_{l'}^0\sin(\theta)d\theta d\phi\end{aligned} \tag{3–42}$$

Now the remain question is the integral of three spherical harmonics product. The skill called Wigner 3-j symbols [20] is applied, which is also closely related to the Clebsch–Gordan coefficients.

$$\begin{aligned}&\int_0^{\pi}\int_0^{2\pi}Y_{l_1}^{m_1}(\theta,\phi)Y_{l_2}^{m_2}(\theta,\phi)Y_{l_3}^{m_3}(\theta,\phi)\sin(\theta)d\theta d\phi=\\&\sqrt{\frac{(2l_1+1)(2l_2+1)(2l_3+1)}{4\pi}}\begin{pmatrix}l_1&l_2&l_3\\0&0&0\end{pmatrix}\begin{pmatrix}l_1&l_2&l_3\\m_1&m_2&m_3\end{pmatrix}\end{aligned} \tag{3–43}$$

$$\begin{aligned}&\int_0^{\pi}\int_0^{2\pi}Y_l^0(\theta,\phi)Y_n^0(\theta,\phi)Y_{l'}^0(\theta,\phi)\sin(\theta)d\theta d\phi=\\&\sqrt{\frac{(2l+1)(2n+1)(2l'+1)}{4\pi}}\begin{pmatrix}l&n&l'\\0&0&0\end{pmatrix}^2\end{aligned} \tag{3–44}$$

$\begin{pmatrix}l_1&l_2&l_3\\m_1&m_2&m_3\end{pmatrix}$ is the Wigner 3-j symbols, the relation between it and CG coefficient

$< j_1 m_1 j_2 m_2 | j_3 m_3 >$ is

$$\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix} = \frac{(-1)^{j_1 - j_2 - m_3}}{\sqrt{2j_3 + 1}} < j_1 m_1 j_2 m_2 | j_3 - m_3 > \tag{3–45}$$

$$\begin{pmatrix} l & n & l' \\ 0 & 0 & 0 \end{pmatrix} = \frac{(-1)^{l-n}}{\sqrt{2l' + 1}} < l 0 n 0 | l' 0 > \tag{3–46}$$

Now we put the three spherical harmonics integral in and we get

$$\begin{aligned} &< i, l | \frac{1}{|\mathbf{r} - \mathbf{R_A}|} | i', l' > \\ &= \sqrt{(2l+1)(2l'+1)} \sum_{n=0}^{\infty} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) \frac{r_<^n}{r_>^{n+1}} dr \begin{pmatrix} l & n & l' \\ 0 & 0 & 0 \end{pmatrix}^2 \end{aligned} \tag{3–47}$$

For the nuclear-nuclear interaction

$$\begin{aligned} &< i, l | \frac{1}{R} | i', l' > \\ &= \frac{1}{R} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) dr \int_0^{\pi} \int_0^{2\pi} Y_l^0(\theta, \phi) Y_{l'}^0(\theta, \phi) \sin(\theta) d\theta d\phi \\ &= \frac{\delta_{ll'}}{R} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) dr \end{aligned} \tag{3–48}$$

For the total Hamilton matrix element

$$\begin{aligned}
&< i,l|H|i',l' >=< i,l|-\frac{1}{2}\nabla^2-\frac{1}{|\mathbf{r}-\mathbf{R_A}|}-\frac{1}{|\mathbf{r}-\mathbf{R_B}|}+\frac{1}{R}|i',l'> \\
&=-\frac{1}{2}\delta_{ll'}(\int_0^{r_{max}} B_i^k(r)(\frac{\partial^2 B_{i'}^k(r)}{\partial r^2}-l'(l'+1)\frac{B_{i'}^k(r)}{r^2})dr)- \\
&2\sqrt{\frac{(2l'+1)}{(2l+1)}}\sum_{n=0,n=even}^{\infty}\int_0^{r_{max}} B_i^k(r)B_{i'}^k(r)\frac{r_<^n}{r_>^{n+1}}C(nl'l|000)^2dr+ \\
&\frac{\delta_{ll'}}{R}\int_0^{r_{max}} B_i^k(r)B_{i'}^k dr
\end{aligned} \tag{3–49}$$

The theoretical formula to derive the Clebsch-Gordan coefficients is

$$\begin{aligned}
&< j_1m_1j_2m_2|JM >= \delta_{m,m_1+m_2}\sqrt{\frac{(2J+1)(J+j_1-j_2)!(J-j_1+j_2)!(j_1+j_2-J)!}{(j_1+j_2+J+1)!}}\cdot \\
&\sqrt{(J+m)!(J-m)!(j_1-m_1)!(j_1+m_1)!(j_2-m_2)!(j_2+m_2)!}\cdot \\
&\sum_k \frac{(-1)^k}{k!(j_1+j_2-J-k)!(j_1-m_1-k)!(j_2+m_2-k)!(J-j_2+m_1+k)!(J-j_1-m_2+k)!}
\end{aligned} \tag{3–50}$$

In our case

$$\begin{aligned}
&< n0l'0|l0 >= \sqrt{\frac{(2l+1)(l+n-l')!(l-n+l')!(n+l'-l)!}{(n+l'+l+1)!}}\cdot \\
&l!n!l'!\cdot\sum_k \frac{(-1)^k}{k!(n+l'-l-k)!(n-k)!(l'-k)!(l-l'+k)!(l-n+k)!}
\end{aligned} \tag{3–51}$$

The summation is extended over all integer k for which the argument of every factorial

is nonnegative, which means:

$$\begin{cases} k \leq j_1 + j_2 - j \\ k \leq j_1 - m_1 \\ k \leq j_2 + m_2 \\ k \geq 0 \\ k \geq j_2 - j - m_1 \\ k \geq j_1 + m_2 - j \end{cases} \tag{3–52}$$

In short

$$\begin{cases} k \leq \min[j_1 + j_2 - j, j_1 - m_1, j_2 + m_2] \\ k \geq \max[0, j_2 - j - m_1, j_1 + m_2 - j] \end{cases} \tag{3–53}$$

In our case is that

$$\begin{cases} k \leq \min[j_1 + j_2 - j, j_1, j_2] \\ k \geq \max[0, j_2 - j, j_1 - j] \end{cases} \tag{3–54}$$

To show accuracy of the our result, we compare the calculated energies of eigenstates with the published results from F. Martin [18] and M. Brosolo [21] in Tab. 3–2 and Tab. 3–3. The wave function of $1s\sigma_g$ in Fig. 3–5 is also plotted to show the degree of exactitude.

Table 3–2: Energies of eigenstates of hydrogen molecule ion computed with B-splines and compared with the exact values ($k = 8,\ r_{max} = 20au, N = 240, l_{max} = 8$), linear sequence

	E	E_{exact}	F. Martin
$1s\sigma_g$	-1.100833114624753	-1.10263	-1.10126
$1s\sigma_u$	-0.665325665349692	-0.6675344	
$2s\sigma_g$	-0.360620951685494	-0.36086	-0.36068

Table 3–3: Energies of eigenstates of hydrogen molecule ion computed with B-splines and compared with the exact values ($k = 10,\ r_{max} = 50au, N = 140, l_{max} = 20$), multi points linear sequence

	E	E_{exact}	M. Brosolo	F. Martin
$1s\sigma_g$	-1.102479567208269	-1.10263	-1.10250	-1.10250
$1s\sigma_u$	-0.667369127643038	-0.667534		
$2s\sigma_g$	-0.360843966991405	-0.360865	-0.36050	-0.36085

3.2.3.2 $m \neq 0$ Case

For the case $m \neq 0$ the equations changed to

$$H_{ilm,i'l'm'} = \int_0^{r_{max}} \int_0^{\pi} \int_0^{2\pi} \frac{B_i^k(r)}{r}[Y_l^m(\theta,\phi)]^* H_{el} \frac{B_{i'}^k}{r} Y_{l'}^{m'}(\theta,\phi) r^2 \sin(\theta) dr d\theta d\phi \tag{3–55}$$

$$S_{ilm,i'l'm'} = \delta_{ll'}\delta_{mm'} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) dr \tag{3–56}$$

The kinetic part

$$< i,l,m| -\frac{1}{2}\nabla^2 |i',l',m'> = \int_0^{r_{max}} \int_0^{\pi} \int_0^{2\pi} \frac{B_i^k(r)}{r}[Y_l^m(\theta,\phi)]^* (-\frac{1}{2})\nabla^2 [\frac{B_{i'}^k}{r} Y_{l'}^{m'}(\theta,\phi)] r^2 \sin(\theta) dr d\theta d\phi \tag{3–57}$$

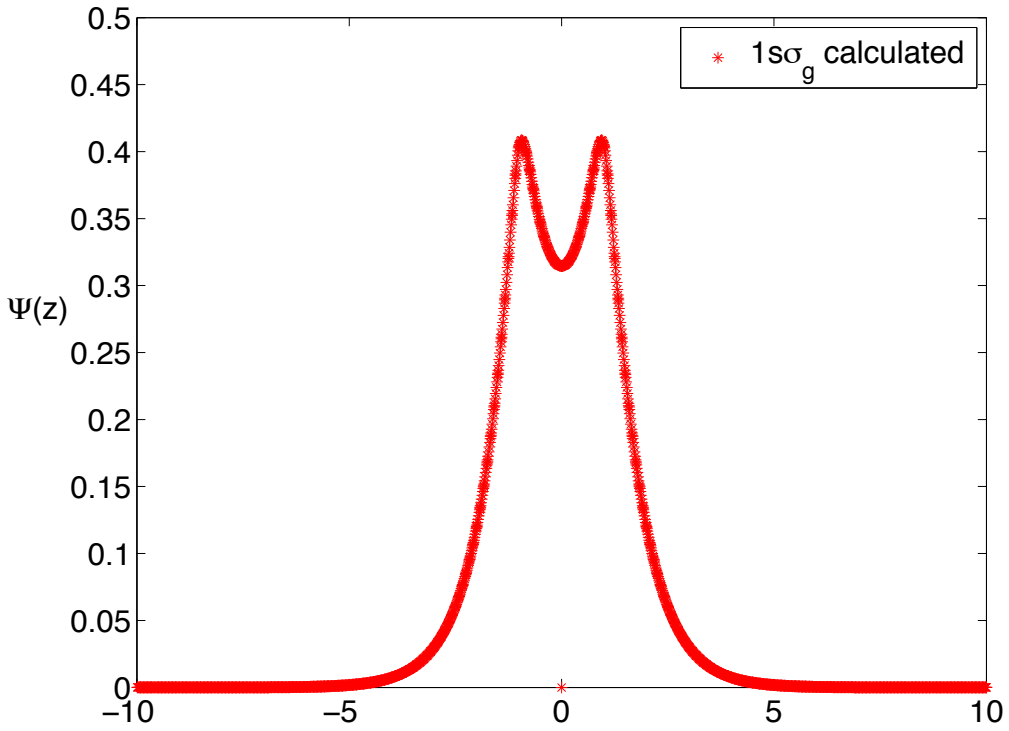


Figure 3–5: The calculated $1s\sigma_g$ through my code

The second derivative has the same format as the $m = 0$ case

$$\nabla^2[\frac{B^k_{i'}}{r}Y^{m'}_{l'}(\theta,\phi)] = \frac{1}{r}\frac{\partial^2 B^k_{i'}(r)}{\partial r^2}Y^{m'}_{l'}(\theta) + \frac{B^k_{i'}(r)}{r^3}(-l'(l'+1)Y^{m'}_{l'}(\theta) \qquad (3\text{–}58)$$

$$
\begin{aligned}
&< i,l,m| -\frac{1}{2}\nabla^2|i',l',m'> \\
&= \int_0^{r_{max}} \int_0^{\pi} \int_0^{2\pi} \frac{B_i^k(r)}{r}[Y_l^m(\theta,\phi)]^*(-\frac{1}{2})[\frac{1}{r}\frac{\partial^2 B_{i'}^k(r)}{\partial r^2}Y_{l'}^{m'}(\theta) + \frac{B_{i'}^k(r)}{r^3} \\
&\quad (-l'(l'+1)Y_{l'}^{m'}(\theta)]r^2\sin(\theta)drd\theta d\phi \\
&= -\frac{1}{2}(\int_0^{r_{max}} B_i^k(r)\frac{\partial^2 B_{i'}^k(r)}{\partial r^2}dr - l'(l'+1)\int_0^{r_{max}} \frac{B_i^k(r)B_{i'}^k(r)}{r^2}dr)\cdot \\
&\quad \int_0^{\pi}\int_0^{2\pi}[Y_l^m(\theta,\phi)]^* Y_{l'}^{m'}(\theta)\sin(\theta)d\theta d\phi \\
&= -\frac{1}{2}\delta_{ll'}\delta_{mm'}(\int_0^{r_{max}} B_i^k(r)(\frac{\partial^2 B_{i'}^k(r)}{\partial r^2} - l'(l'+1)\frac{B_{i'}^k(r)}{r^2})dr)
\end{aligned}
\tag{3–59}
$$

Expand the Coulomb potential through the method mentioned before

$$
\begin{aligned}
&\frac{1}{|\mathbf{r}-\mathbf{R}|} = \sum_{l=0}^{\infty}\frac{4\pi}{2l+1}\sum_{m=-l}^{l}(-1)^m\frac{r_<^l}{r_>^{l+1}}Y_l^{-m}(\theta,\phi)Y_l^m(\theta',\phi') \\
&r_< = min(r,R/2), r_> = max(r,R/2), \mathbf{r} = (r,\theta,\phi), \mathbf{R} = (R/2,\theta',\phi')
\end{aligned}
\tag{3–60}
$$

$$
Y_l^m = (-1)^m\sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}e^{im\phi}P_l^m(\cos\theta) \tag{3–61}
$$

$$
P_l^m(x) = (1-x^2)^{|m|/2}(\frac{d}{dx})^{|m|}P_l(x) \tag{3–62}
$$

$$
P_l(x) = \frac{1}{2^l l!}(\frac{d}{dx})^l(x^2-1)^l \tag{3–63}
$$

$$
\begin{aligned}
&Y_l^m(0,0) = \\
&(-1)^m\sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}e^{im\phi}\frac{1}{2^l l!}(1-x^2)^{\frac{|m|}{2}}\cdot(\frac{d}{dx})^{l+|m|}(x^2-1)^l|_{x=1}
\end{aligned}
\tag{3–64}
$$

For $m \neq 0$

$$
Y_l^m(0,0) = 0 \tag{3–65}
$$

$$
Y_l^m(\pi,0) = 0 \tag{3–66}
$$

$$\frac{1}{|\mathbf{r}-\mathbf{R_A}|}=\sum_{l=0}^{\infty}\frac{4\pi}{2l+1}\frac{r_<^l}{r_>^{l+1}}Y_l^0(\theta)\sqrt{\frac{(2l+1)}{4\pi}} \tag{3–67}$$

$$\frac{1}{|\mathbf{r}-\mathbf{R_B}|}=\sum_{l=0}^{\infty}\frac{4\pi}{2l+1}\frac{r_<^l}{r_>^{l+1}}Y_l^0(\theta)\sqrt{\frac{(2l+1)}{4\pi}}(-1)^l \tag{3–68}$$

The expansion of Coulomb potential is still as same as the $m=0$ case, interestingly.

$$\begin{aligned}<i,l,m|\frac{1}{|\mathbf{r}-\mathbf{R_A}|}|i',l',m'>=&\sum_{n=0}^{\infty}\sqrt{\frac{4\pi}{(2n+1)}}\int_0^{r_{max}}B_i^k(r)B_{i'}^k(r)\frac{r_<^n}{r_>^{n+1}}dr\\&\int_0^{\pi}\int_0^{2\pi}[Y_l^m(\theta)]^*Y_n^0Y_{l'}^{m'}\sin(\theta)d\theta d\phi\end{aligned} \tag{3–69}$$

$$[Y_l^m]^*=(-1)^mY_l^{-m}(\theta,\phi) \tag{3–70}$$

Now we need to calculate the three spherical harmonics integral with different m.

$$\begin{aligned}&\int_0^{\pi}\int_0^{2\pi}[Y_l^m]^*(\theta)Y_n^0Y_{l'}^{m'}\sin(\theta)d\theta d\phi\\&=\int_0^{\pi}\int_0^{2\pi}(-1)^mY_l^{-m}(\theta,\phi)Y_n^0(\theta,\phi)Y_{l'}^{m'}(\theta,\phi)\sin(\theta)d\theta d\phi\\&=(-1)^m\sqrt{\frac{(2l+1)(2n+1)(2l'+1)}{4\pi}}\begin{pmatrix}l&n&l'\\0&0&0\end{pmatrix}\begin{pmatrix}l&n&l'\\-m&0&m'\end{pmatrix}\end{aligned} \tag{3–71}$$

$$\begin{pmatrix}j_1&j_2&j_3\\m_1&m_2&m_3\end{pmatrix}=\frac{(-1)^{j_1-j_2-m_3}}{\sqrt{2j_3+1}}<j_1m_1j_2m_2|j_3-m_3> \tag{3–72}$$

$$\begin{pmatrix}l&n&l'\\0&0&0\end{pmatrix}=\frac{(-1)^{l-n}}{\sqrt{2l'+1}}<l0n0|l'0> \tag{3–73}$$

$$\begin{pmatrix}l&n&l'\\-m&0&m'\end{pmatrix}=\frac{(-1)^{l-n-m'}}{\sqrt{2l'+1}}<l-mn0|l'-m'> \tag{3–74}$$

The formula to derive the Clebsch-Gordan coefficients

$$< j_1 m_1 j_2 m_2 | JM >= \delta_{m,m_1+m_2} \sqrt{\frac{(2J+1)(J+j_1-j_2)!(J-j_1+j_2)!(j_1+j_2-J)!}{(j_1+j_2+J+1)!}} \cdot$$
$$\sqrt{(J+m)!(J-m)!(j_1-m_1)!(j_1+m_1)!(j_2-m_2)!(j_2+m_2)!} \cdot$$
$$\sum_k \frac{(-1)^k}{k!(j_1+j_2-J-k)!(j_1-m_1-k)!(j_2+m_2-k)!(J-j_2+m_1+k)!(J-j_1-m_2+k)!} \tag{3–75}$$

In our case

$$< l0n0|l'0 >= \sqrt{\frac{(2l'+1)(l'+l-n)!(l'-l+n)!(n+l-l')!}{(n+l'+l+1)!}} \cdot$$
$$l!n!l'! \cdot \sum_k \frac{(-1)^k}{k!(l+n-l'-k)!(l-k)!(n-k)!(l'-n+k)!(l'-l+k)!} \tag{3–76}$$

$$< l-mn0|l'-m' >= \delta_{-m,-m'} \sqrt{\frac{(2l'+1)(l'+l-n)!(l'-l+n)!(n+l-l')!}{(n+l'+l+1)!}} \cdot$$
$$\sqrt{(l-m)!(l+m)!(l'-m')!(l'+m')!n!n!} \cdot$$
$$\sum_k \frac{(-1)^k}{k!(l+n-l'-k)!(l+m-k)!(n-k)!(l'-n-m+k)!(l'-l+k)!} \tag{3–77}$$

The integral $\neq 0$ only when $m = m'$

$$\int_0^{\pi} \int_0^{2\pi} [Y_l^m]^*(\theta) Y_n^0 Y_{l'}^{m'} \sin(\theta) d\theta d\phi$$
$$= \delta_{m,m}(-1)^m \sqrt{\frac{(2l+1)(2n+1)(2l'+1)}{4\pi}} \begin{pmatrix} l & n & l' \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} l & n & l' \\ -m & 0 & m \end{pmatrix} \tag{3–78}$$

For the total electronic nuclear interaction, we have

$$\begin{aligned}
&< i,l,m| - \frac{1}{|\mathbf{r}-\mathbf{R_A}|} - \frac{1}{|\mathbf{r}-\mathbf{R_B}|}|i',l',m' >= \\
&= -2(-1)^m \delta_{m,m'} \sum_{n=0,n=even}^{\infty} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) \frac{r_<^n}{r_>^{n+1}} dr \sqrt{(2l+1)(2l'+1)} \cdot \\
&\begin{pmatrix} l & n & l' \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} l & n & l' \\ -m & 0 & m' \end{pmatrix}
\end{aligned} \tag{3–79}$$

For the nuclear-nuclear interaction, we have

$$\begin{aligned}
&< i,l,m|\frac{1}{R}|i',l',m' > \\
&\quad = \frac{1}{R} \int_0^{r_{max}} B_i^k(r) B_{i'}^k dr \int_0^{\pi} \int_0^{2\pi} [Y_l^m]^*(\theta,\phi) Y_{l'}^{m'}(\theta,\phi) \sin(\theta) d\theta d\phi \\
&\quad = \frac{\delta_{ll'}\delta_{mm'}}{R} \int_0^{r_{max}} B_i^k(r) B_{i'}^k dr
\end{aligned} \tag{3–80}$$

For the total Hamilton matrix element, we have

$$\begin{aligned}
&< i,l,m|H|i',l',m' > \\
&= \delta_{mm'} \cdot [-\frac{1}{2}\delta_{ll'}(\int_0^{r_{max}} B_i^k(r)(\frac{\partial^2 B_{i'}^k(r)}{\partial r^2} - l'(l'+1)\frac{B_{i'}^k(r)}{r^2})dr) + \\
&- 2(-1)^m \sum_{n=0,n=even}^{\infty} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) \frac{r_<^n}{r_>^{n+1}} dr \sqrt{(2l+1)(2l'+1)} \cdot \\
&\begin{pmatrix} l & n & l' \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} l & n & l' \\ -m & 0 & m' \end{pmatrix} + \frac{\delta_{ll'}}{R} \int_0^{r_{max}} B_i^k(r) B_{i'}^k dr]
\end{aligned} \tag{3–81}$$

As a result, the equation set we need to calculate here is a set with the same magnetic quantum number. During writing the programme, we can specify the m we input and then get the eigenstate with the specific m.

3.3 Solving for TDSE

3.3.1 Introduction to the Existing Algorithm: QPROP

In order to solve the TDSE using B-splines, we first introduce one of the most elegant algorithm for solving TDSE which is raised by Muller *et al* [1] and have some help for using the open source software QPROP [22]. The basic idea about how to solve TDSE is illustrated below.

The typical TDSE formulated as this

$$i\frac{\partial\psi(\mathbf{r},t)}{\partial t} = H\psi(\mathbf{r},t) \tag{3–82}$$

set $\psi_i(\mathbf{r},t)$ as the ith orbital wave function and set the expansion of it as

$$\psi_i(\mathbf{r},t) = \frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega) \tag{3–83}$$

substitute it in Eq. 3–82

$$i\frac{\partial\frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega)}{\partial t} = H\frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega) \tag{3–84}$$

Now we put the detail of Hamiltonian in it

$$\begin{aligned} &i\frac{\partial\frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega)}{\partial t} \\ &= \frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}(-\frac{1}{2}\frac{\partial^2}{\partial r^2} + \frac{l(l+1)}{2r^2} + V(\mathbf{r}) + V_I(t))\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega) \end{aligned} \tag{3–85}$$

Here $V(\mathbf{r})$ is the static potential which is caused by the Coulomb potential of model itself and $V_I(t)$ is the interaction potential caused by extern laser pulses. Project it to

the Y_{lm}, for the hydrogen case we can get

$$i\frac{\partial\Phi_{ilm}(\mathbf{r},t)}{\partial t} = (-\frac{1}{2}\frac{\partial^2}{\partial r^2}+\frac{l(l+1)}{2r^2}+V(r))\Phi_{ilm}(\mathbf{r},t)+\sum_{l'=0}^{\infty}\sum_{m'=-l}^{l}< Y_{lm}(\Omega)|V_I(t)|\Phi_{il'm'}(\mathbf{r},t)Y_{l'm'}(\Omega) > \tag{3–86}$$

For linear polarization along $\mathbf{e}_z$, we have the external potential in the form of

$$V_I(t)=\begin{cases}-iA(t)\frac{\partial}{\partial z}+\frac{A(t)^2}{2} & \text{velocity gauge}\\ zE(t) & \text{length gauge}\end{cases} \tag{3–87}$$

For the ease of illustration, we use length gauge here for the rest of deduction

$$\begin{aligned}i\frac{\partial\Phi_{ilm}(\mathbf{r},t)}{\partial t} = (-\frac{1}{2}\frac{\partial^2}{\partial r^2}+\frac{l(l+1)}{2r^2}+V(r))\Phi_{ilm}(\mathbf{r},t)\\ +rE(t)\sum_{l'=0}^{\infty}\sum_{m'=-l}^{l}< Y_{lm}(\Omega)|\cos(\theta)|\Phi_{il'm'}(\mathbf{r},t)Y_{l'm'}(\Omega) >\end{aligned} \tag{3–88}$$

The m term vanishes because the external potential only contains θ part. After integrating it, only δ_{mm} remains.

$$\begin{aligned}i\frac{\partial\Phi_{ilm}(\mathbf{r},t)}{\partial t} = (-\frac{1}{2}\frac{\partial^2}{\partial r^2}+\frac{l(l+1)}{2r^2}+V(r))\Phi_{ilm}(\mathbf{r},t)\\ +rE(t)\sum_{l'=0}^{\infty}< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) > \Phi_{il'm}(\mathbf{r},t)\end{aligned} \tag{3–89}$$

As a result, the m is decoupled under the linear polarized laser, let's set the ith orbital

as

$$\psi_i(\mathbf{r},t) = \frac{1}{r}\sum_{l=0}^{\infty}\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega) \approx \frac{1}{r}\sum_{l=0}^{L-1}\Phi_{ilm}(\mathbf{r},t)Y_{lm}(\Omega) \tag{3–90}$$

The ith orbital only has a single magnetic quantum number $m = m_i$, and only L angular orbitals are involved in the propagation with maximum angular momentum $l = L - 1$.

To solve the $< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) >$, we still need the Wigner 3-j symbols we already mentioned. $\cos(\theta)$ can be replaced by Y_1^0

$$Y_1^0 = \frac{1}{2}\sqrt{\frac{3}{\pi}}\cos(\theta) \tag{3–91}$$

The integral is now changed into the form of three harmonics integral

$$< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) >= 2\sqrt{\frac{\pi}{3}} < Y_{lm}|Y_{10}|Y_{l'm} > \tag{3–92}$$

$$\begin{aligned} < Y_{lm}|Y_{10}|Y_{l'm} > &= \int (Y_l^m)^* Y_1^0 Y_{l'}^m d\Omega \\ &= (-1)^m\sqrt{\frac{(2l+1)3(2l'+1)}{4\pi}}\begin{pmatrix} l & 1 & l' \\ 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} l & 1 & l' \\ -m & 0 & m \end{pmatrix} \end{aligned} \tag{3–93}$$

Where

$$\begin{pmatrix} l & 1 & l' \\ 0 & 0 & 0 \end{pmatrix} = \frac{(-1)^{l-1}}{\sqrt{2l'+1}} < l010|l'0 > \begin{pmatrix} l & 1 & l' \\ -m & 0 & m \end{pmatrix} = \frac{(-1)^{l-1-m}}{\sqrt{2l'+1}} < l-m10|l'-m > \tag{3–94}$$

$$< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) >= \sqrt{\frac{2l+1}{2l'+1}} < l010|l'0 >< l-m10|l'-m > \tag{3–95}$$

$$< l-m10|l'-m >= \sqrt{\frac{(2l'+1)(l'+l-1)!(l'-l+1)!(1+l-l')!}{(1+l'+l+1)!}}\cdot$$
$$\sqrt{(l-m)!(l+m)!(l'-m')!(l'+m')!1!1!}\cdot \tag{3–96}$$
$$\sum_k \frac{(-1)^k}{k!(l+1-l'-k)!(l+m-k)!(1-k)!(l'-1-m+k)!(l'-l+k)!}$$

For the three $< l-m10|l'-m >\neq 0$ case

1.$l' = l+1$

$$< j_1 m10|(j_1+1)m >= \sqrt{\frac{(j_1-m+1)(j_1+m+1)}{(2j_1+1)(j_1+1)}} \tag{3–97}$$
$$< l-m10|l+1-m >= \sqrt{\frac{(l+m+1)(l-m+1)}{(2l+1)(l+1)}} \tag{3–98}$$

2.$l' = l$

$$< j_1 m10|(j_1)m >= \frac{m}{\sqrt{j_1(j_1+1)}} \tag{3–99}$$
$$< l-m10|l-m >= \frac{-m}{\sqrt{l(l+1)}} \tag{3–100}$$

3.$l' = l-1$

$$< j_1 m10|(j_1-1)m >= \sqrt{\frac{(j_1-m)(j_1+m)}{(2j_1+1)j_1}} \tag{3–101}$$
$$< l-m10|l-1-m >= \sqrt{\frac{(l-m)(l+m)}{(2l+1)l}} \tag{3–102}$$

As a result

1.$l' = l + 1$

$$< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) >= \sqrt{\frac{(l+m+1)(l-m+1)}{(2l+3)(2l+1)}} \tag{3–103}$$

2.$l' = l$

$$< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) >= 0 \tag{3–104}$$

3.$l' = l - 1$

$$< Y_{lm}(\Omega)|\cos(\theta)|Y_{l'm}(\Omega) >= \sqrt{\frac{(l-m)(l+m)}{(2l+1)(2l-1)}} \tag{3–105}$$

Set $c_{lm} = \sqrt{\frac{(l+1)^2-m^2}{(2l+3)(2l+1)}}$ and write the TDSE into matrix form

$$i\frac{\partial}{\partial t}\Phi_i(rt) = (\mathbf{H_{at}} + \mathbf{H_{ang}})\Phi_i(r,t) \tag{3–106}$$

$$\Phi_i(rt) = [\Phi_{i0m}(rt), \Phi_{i1m}(rt), \cdots, \Phi_{i(L-1)m}(rt)]^{\mathrm{T}} \tag{3–107}$$

$$\mathbf{H_{at}} = (-\frac{1}{2}\frac{\partial^2}{\partial r^2} + \frac{l(l+1)}{2r^2} + V(r)) \tag{3–108}$$

$$\mathbf{H_{ang}} = rE(t)\begin{pmatrix} 0 & c_{0m} & 0 & 0 & 0 \\ c_{0m} & 0 & c_{1m} & 0 & \cdots \\ 0 & c_{1m} & 0 & c_{2m} & \cdots \\ \vdots & 0 & c_{2m} & 0 & \cdots \end{pmatrix} \tag{3–109}$$

In order to calculate the second derivative of the wave function, here we use the implicit

fourth order Simpson and Numerov expressions.

$$\frac{\partial^2 \Phi_{ilm}}{\partial r^2} = \Phi''_{ilm} \approx (1 + \frac{h^2}{12}\Delta_2)^{-1}\Delta_2 \Phi_{ilm} =: -2M_2^{-1}\Delta_2 \Phi_{ilm} \tag{3–110}$$

$$\Delta_2 \Phi_{ilm}(r_n t) =: (\Phi_{ilm}(r_{n+1}t) - 2\Phi_{ilm}(r_n t) + \Phi_{ilm}(r_{n-1}t))/h^2 \tag{3–111}$$

$$M_2 = -\frac{1}{6}\begin{pmatrix} 10 & 1 & & \\ 1 & 10 & 1 & \\ & 1 & 10 & \ddots \\ & & \ddots & \ddots \end{pmatrix} \tag{3–112}$$

$$\mathbf{H} = \mathbf{H_{at}} + \sum_{l=0}^{L-2} \mathbf{H_{ang}^{lm}} \tag{3–113}$$

$$\mathbf{H_{ang}^{0m}} = rE(t)\begin{pmatrix} 0 & c_{0m} & 0 & 0 & 0 \\ c_{0m} & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & \cdots \\ \vdots & 0 & 0 & 0 & \cdots \end{pmatrix} \tag{3–114}$$

In the Coulomb potential case $-\frac{Z}{r}$ the second derivative of a radial s-state orbitals satisfies

$$\Phi''_{il=0m}(0t) = -2Z\Phi'_{il=0m}(0t) \neq 0 \tag{3–115}$$

The deduction is as following

$$-\frac{1}{2}\Phi_{ilm} + [V + \frac{l(l+1)}{r^2}]\Phi_{ilm} = E\Phi_{ilm} \tag{3–116}$$

This is the separation of the radial part. Since $l = 0$, and $\Phi = 0$ at $r = 0$, $\frac{\Phi}{r} = constant$,

as a result

$$-\frac{1}{2}\Phi_{il=0m}+\frac{-Z}{r}\Phi_{il=0m}=E\Phi_{il=0m} \tag{3–117}$$

$$-\frac{1}{2}\Phi_{il=0m}+\frac{-Z}{r}\Phi_{il=0m}=0 \tag{3–118}$$

Since the first derivative is 0 for Coulomb case

$$\frac{d\frac{\Phi}{r}}{r}=0 \tag{3–119}$$

$$\frac{-\Phi}{r^2}+\frac{\Phi'}{r}=0 \tag{3–120}$$

$$\Phi'=\frac{\Phi}{r} \tag{3–121}$$

so $\Phi''_{il=0m}(0t) = -2Z\Phi'_{il=0m}(0t)$. As a result, we should modify the element in Δ_2 and M_2

$$(\Delta_2)_{11}=-\frac{2}{h^2}(1-\frac{Zh}{12-10Zh}) \tag{3–122}$$

$$(M_2)_{11}=-2(1+\frac{h^2}{12}(\Delta)_{11}) \tag{3–123}$$

$$\mathbf{H_{at}}=1_l\otimes(M_2^{-1}\Delta_2+V(r)+\frac{l(l+1)}{2r^2}) \tag{3–124}$$

$$\mathbf{H_{ang}^{lm}}=\mathbf{L^{lm}}\otimes(rE(t))1_r \tag{3–125}$$

$$\mathbf{L^{lm}}=\begin{pmatrix}0 & c_{lm}\\ c_{lm} & 0\end{pmatrix} \tag{3–126}$$

$1_l, 1_r$ are the unity matrices. The split operator is

$$\psi(t_{end}) = e^{\int_{t_0}^{t_{end}} -iHdt}\psi(t_0) \tag{3–127}$$

$$U(t+\Delta t, t) \approx exp[-i\Delta t H(t+\frac{\Delta t}{2})] \tag{3–128}$$

$$t = 2\tau \tag{3–129}$$

$$U_{split}(t+2\tau, t) = \prod_{l=L-2}^{0} exp(-i\tau H_{ang}^{lm}) exp(-2i\tau H_{at}) \prod_{l=0}^{L-2} exp(-i\tau H_{ang}^{lm}) \tag{3–130}$$

$$\psi(\mathbf{t}+\mathbf{2}\tau) = \mathbf{U_{split}}(\mathbf{t}+\mathbf{2}\tau, \mathbf{t})\psi(\mathbf{t}) \tag{3–131}$$

3.3.2 Code Help for QPROP

The following is some tables of the function used in QPROP which can help people understand the well-packaged code and revise it.

Table 3–4: class grid

ngps_x:		Returns number of radial gridpoints Nr
ngps_y:		Returns number of angular momenta L
ngps_z:		Returns number of orbitals N
set_dim:		Defines type of expansion, elliptical polarized dim=44 or linear polarized dim=34
set_ngps	(N_r, L, orbital numbers)	Defines Nr, L, and number of orbitals.
set delt		Defines Δr
set_offs		offset in r, l, and N

Table 3–5: class hamop

init	(g,vecpot_x,vecpot_y,vecpot_z,scalarpotx,scalarpoty,scalarpotz, imagpot,field)

Table 3–6: in the "potential.hh"

scalarpotx	(nuclear-charge, pot-cutoff):	spherically symmetric potential V(r),x→r
imagpot	(long xindex, yindex, zindex,double t, grid g)	imaginary potential $V_{im}(r)$ xindex : radial index

The pot-cutoff is in the modified Coulomb Potential

$$V_{bind}(r) = \begin{cases} -\frac{1}{r} & if\ r < R_{co} \\ \frac{r-R_{co}}{R_{co}^2} - \frac{1}{R_{co}} & if\ R_{co} \leq r < 2R_{co} \\ 0 & r \geq 2R_{co} \end{cases} \tag{3–132}$$

R_{co} is the pot-cutoff

Table 3–7: class fluid: real value one-dimensional arrays

init:	load parameters

The following is the result we get through QPROP. Fig. 3–6 is the vector potential of laser pulses, Fig. 3–7 is the observable variants calculated through the time propagation of wave function. Fig. 3–8 is the single photon ionization of ground state of hydrogen, under the laser pulses of $I = 1 \times 10^{13} W/cm^2 (0.0168 \text{ a.u.}), \omega = 0.8, L = 4fs (165.4 \text{ a.u.})$, the laser pulse is in the format of $E(t) = E_0[\sin^2(\frac{\pi t}{T}) \sin(\omega t) - \frac{\pi}{\omega T} \sin(\frac{2\pi t}{T} \cos(\omega t))]$

However, this algorithm can only be applied to the static potential with spherical symmetry and the derivative method don't have a high accuracy order. That's the reason we need to improve it with B-splines.

Table 3–8: class wavefunction: complex value one-dimensional arrays

init:	(long size)(grid g, int inittype, double w, fluid ls) (grid g, FILE file, int ooi, int iv)	Various ways to initialize wavefunction objects
init_rlm	(grid g, int inittype, double w, fluid ls, fluid ms)	for the qprop_dimens=44
calculate_staticpot	(grid g, hamop hamil)	Calculates static part of total potential (scalarpotx + centrifugal potential + imagpot) using Hamiltonian
dump_to_file_sh	(grid g, FILE file, int st, int iv)	Saves orbitals in file
energy	(double time, grid g, hamop hamil, int me, wavefunction staticpot, double charge)	calculates kinetic energy plus potential energy due to static potential staticpot $E_{orbi} =< \Psi_i \vert -\frac{\nabla^2}{2} - \frac{Z}{r} \vert \Psi_i >$
propagate	(complex timestep, double time, grid g, hamop hamil, int me, wavefunction staticpot, int m, double nuclear charge)	Propagates a wavefunction from time to time+timestep. hamil determines only external fields in this function. Potential of nucleus is supposed to be contained in staticpot (see calculate staticpot() in this table). nuclear charge is used to account for correction $(\Delta_2)_{11} = -\frac{2}{h^2}(1 - \frac{Zh}{12-10Zh})$.
expect_z	(grid g)	Calculates $z =< \Psi(t) \vert z \vert \Psi(t) >$ for a single orbital.

Table 3–9: the meaning of the ““initial.param””

nuclear-charge	double 1.0	
pot-cutoff	double 25.0	
delta-r	double 0.2	
radial-grid-size	double 100.0	the length of radius interval
ell-grid-size	long 1	
qprop-dim	long 34	
initial-m	long 0	
initial-ell	long 0	

Table 3–10: the meaning of the "parameters"

| iinitmode = 2 | 1 – random, 2 – hydrogenic wf. | The initial guess wave function |

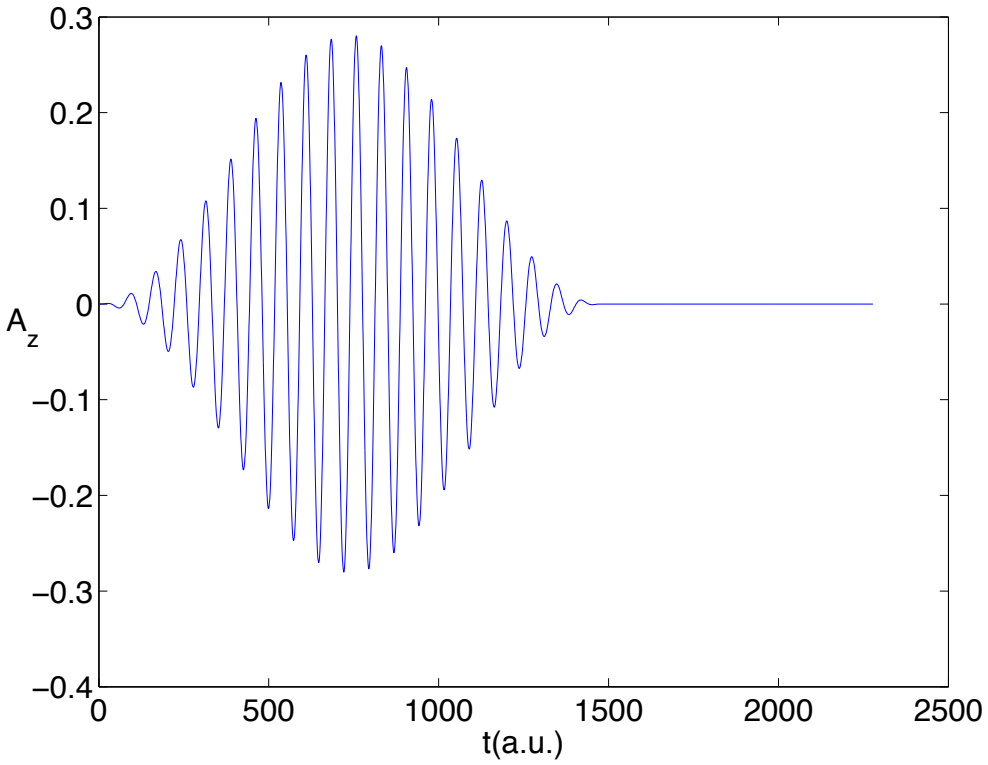


Figure 3–6: The distribution of wave function on different l

3.3.3 Revised Algorithm for Linear Polarized Laser Pulses

In stead expanding a wave function into different orbitals, we expand it with the bases of B-splines.

$$\psi(r,\theta,\phi) = \sum_{l=0}^{l_{max}} \sum_{i=0}^{N_l} c_{il}^n \frac{B_i^k}{r} Y_l^m(\theta,\phi) \tag{3–133}$$

$$i\frac{\partial}{\partial t}\psi(t) = [H_0 + D(t)]\psi(t) \tag{3–134}$$

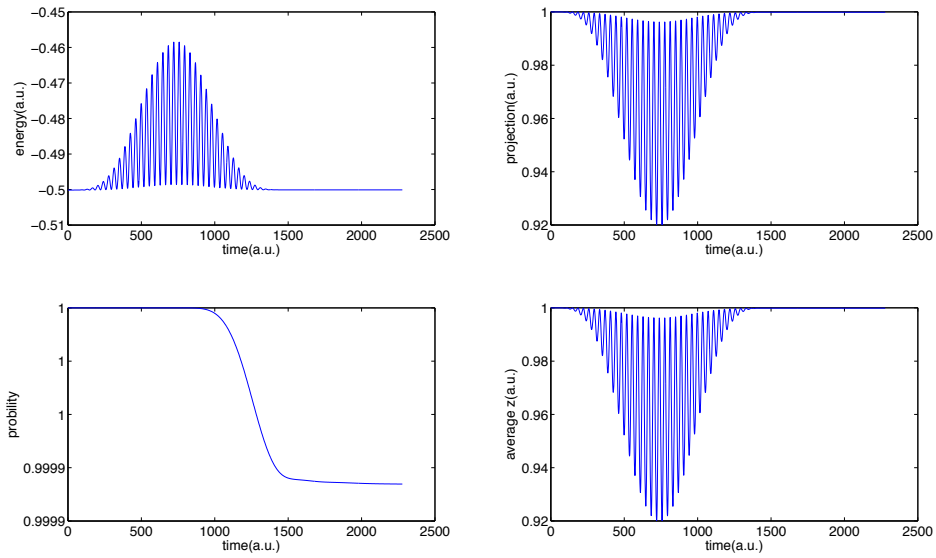


Figure 3–7: The observable variant calculated during the time evolution

Gauge	Formular
Length	$D(t) = -E(t) \cdot r$
Velocity	$D(t) = -A(t) \cdot p$

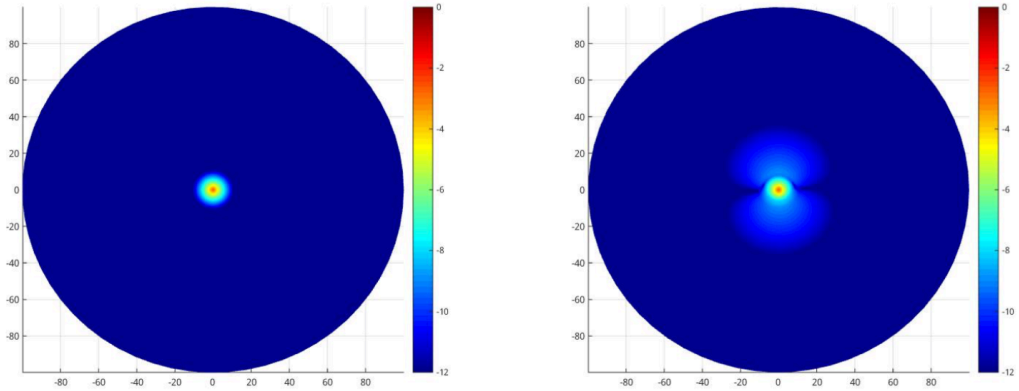
Table 3–11: $D(t)$ in different gauge

Substitute this expansion into TDSE and project it on $\frac{B_i^k}{r} Y_l^m(\theta, \phi)$, we can get the similar matrix form as mentioned previously. $D(t)$ is the laser-atom interaction

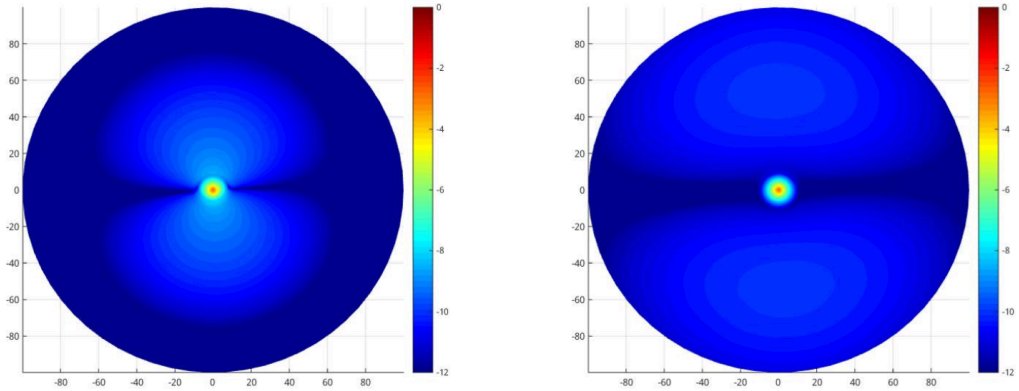
$$i\mathbf{S} \cdot \dot{\mathbf{c}}(t) = (\bar{\mathbf{H}}_0 + \bar{\mathbf{D}}(t)) \cdot \mathbf{c}(t) \tag{3–135}$$

The elements of **S** is as same as before

$$S_{il,i'l'} = \delta_{ll'} \int_0^{r_{max}} B_i^k(r) B_{i'}^k(r) dr \tag{3–136}$$



(a) 0fs (b) 1.33fs



(c) 2.66fs (d) 4.0fs

Figure 3–8: The single photon ionization of ground state of hydrogen, under the laser pulses of $I = 1 \times 10^{13} W/cm^2 (0.0168 \text{ a.u.}), \omega = 0.8, L = 4fs (165.4 \text{ a.u.})$ in ln scale

To solve the problem, we approximate $\dot{c}(t)$ and $c(t)$ to

$$\dot{c}(t) \approx \frac{c(t + \frac{\delta_t}{2}) - c(t - \frac{\delta_t}{2})}{\delta t} \tag{3–137}$$

$$c(t) \approx \frac{c(t + \frac{\delta_t}{2}) + c(t - \frac{\delta_t}{2})}{2} \tag{3–138}$$

With this approximation we can change the TDSE into a simple $Ax = b$ linear equations

$$\mathbf{A}(t) \cdot \mathbf{c}(t + \delta t) = \mathbf{b}(t) \tag{3–139}$$

$$\mathbf{A}(t) = \mathbf{S} + i(\bar{\mathbf{H}}_0 + \bar{\mathbf{D}}(t))\frac{\delta t}{2} \tag{3–140}$$

$$\mathbf{b}(t) = [\mathbf{S} - i((\bar{\mathbf{H}}_0 + \bar{\mathbf{D}}(t))\frac{\delta t}{2}] \cdot \mathbf{c}(t) \tag{3–141}$$

The elements of $\bar{\mathbf{H}}_0$ is the atom hamilton

$$\begin{aligned}(\bar{\mathbf{H}}_0)_{il,i'l'} = \delta_{ll'}[-\frac{1}{2}(\int_0^{r_{max}} B_i^k(r)(\frac{\partial^2 B_{i'}^k(r)}{\partial r^2} - l'(l'+1)\frac{B_{i'}^k(r)}{r^2})dr) \\ + (\int_0^{r_{max}} B_i^k(r) - \frac{Z}{r}B_{i'}^k(r)dr]\end{aligned} \tag{3–142}$$

The laser pulses integral elements are

$$\begin{aligned}\bar{\mathbf{D}}(t)_{il,i'l'} &=< i,l|E_z r\cos(\theta)|i'l' > \\ &= E_z\int_0^{r_{max}} B_i^k(r) r B_{i'}^k(r)dr \begin{cases} c_{lm} & l' = l+1 \\ 0 & else \\ c_{l'm} & l' = l-1 \end{cases}\end{aligned} \tag{3–143}$$

$$Y_l^0(0,0) = \sqrt{\frac{(2l+1)}{4\pi}} \tag{3–144}$$

$$\psi(r,0,0) = \sum_{l=0}^{l_{max}}\sum_{i=0}^{N_l} c_{il}^n \frac{B_i^k}{r} Y_l^0(0,0) \tag{3–145}$$

$$\psi_l(r,0,0) = \sum_{i=0}^{N_l} c_{il}^n \frac{B_i^k}{r} Y_l^0(0,0) \tag{3–146}$$

To solve the complex linear equations, we separate the real and image parts of matrixes

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{3–147}$$

$$(\mathbf{A_{re}}+\mathbf{iA_{im}})\cdot(\mathbf{x_{re}}+\mathbf{ix_{im}})=\mathbf{b_{re}}+\mathbf{ib_{im}} \tag{3–148}$$

$$\mathbf{A_{re}}\cdot\mathbf{x_{re}}-\mathbf{A_{im}}\cdot\mathbf{x_{im}}+\mathbf{i}(\mathbf{A_{im}}\cdot\mathbf{x_{re}}+\mathbf{A_{re}}\cdot\mathbf{x_{im}})=\mathbf{b_{re}}+\mathbf{ib_{im}} \tag{3–149}$$

$$\begin{cases}\mathbf{A_{re}}\cdot\mathbf{x_{re}}-\mathbf{A_{im}}\cdot\mathbf{x_{im}}=\mathbf{b_{re}}\\ \mathbf{A_{im}}\cdot\mathbf{x_{re}}+\mathbf{A_{re}}\cdot\mathbf{x_{im}}=\mathbf{b_{im}}\end{cases}$$

which can convert in to a larger real linear equations

$$\begin{pmatrix}A_{re} & -A_{im}\\ A_{im} & A_{re}\end{pmatrix}\cdot\begin{pmatrix}x_{re}\\ x_{im}\end{pmatrix}=\begin{pmatrix}b_{re}\\ b_{im}\end{pmatrix} \tag{3–150}$$

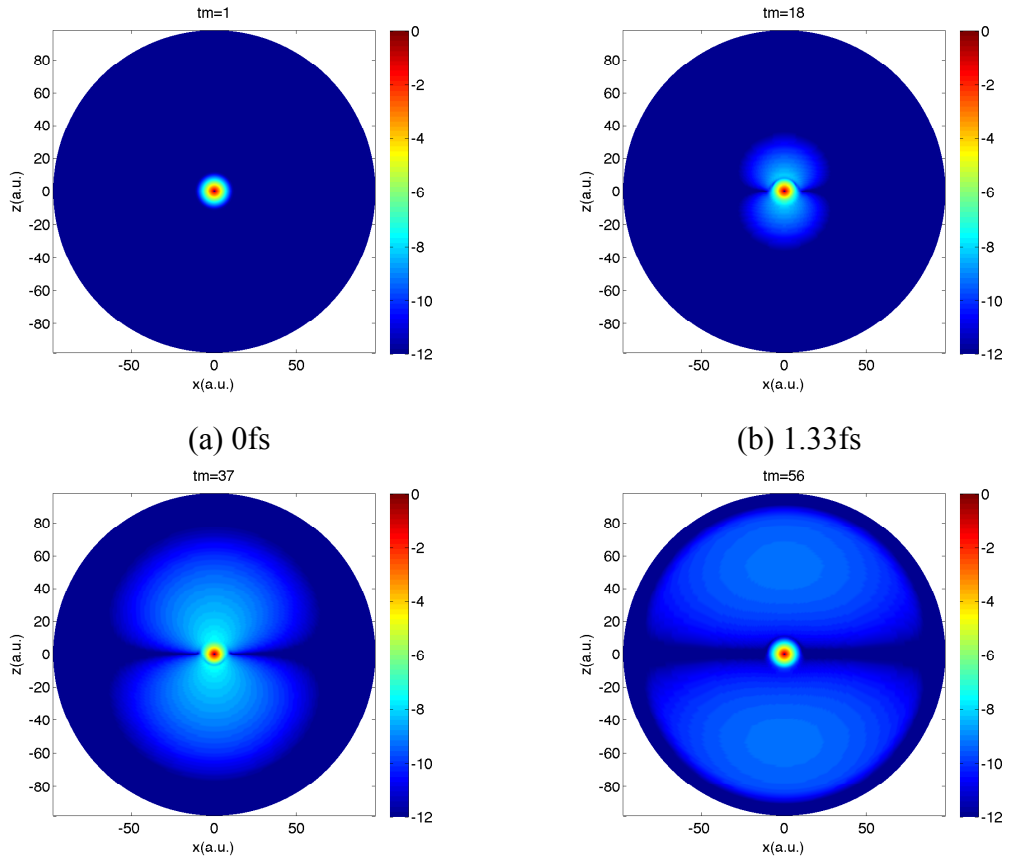
(1) Unification and Plot for Linear Laser Pulse Cases

This is the unification step for the wave function expansion containing angular momentum l part

$$sum=\delta_{ll'}\int_0^{r_{max}}\sum_{l=0}^{l_{max}}\sum_{i=1}^{N_l}\sum_{i'=1}^{N_l}c_{il}c_{i'l}B_i^kB_{i'}^kdr \tag{3–151}$$

$$sum=\delta_{ll'}\sum_{i=1}^{N_l}\sum_{i'=1}^{N_l}\sum_{l=0}^{l_{max}}c_{il}c_{i'l}\int_0^{r_{max}}B_i^kB_{i'}^kdr \tag{3–152}$$

Here I used the same example Fig. 3–9 as Fig. 3–8 to test the rightness of my code. With the advance of this algorithm, we can observe the wave function contribute by different angular momentum.



(a) 0fs (b) 1.33fs

(c) 2.66fs (d) 4.0fs

Figure 3–9: The single photon ionization of ground state of hydrogen, under the laser pulses of $I = 1 \times 10^{13} W/cm^2 (0.0168 \text{ a.u.}), \omega = 0.8, L = 4fs (165.4 \text{ a.u.})$ in ln scale by the revised algorithm.

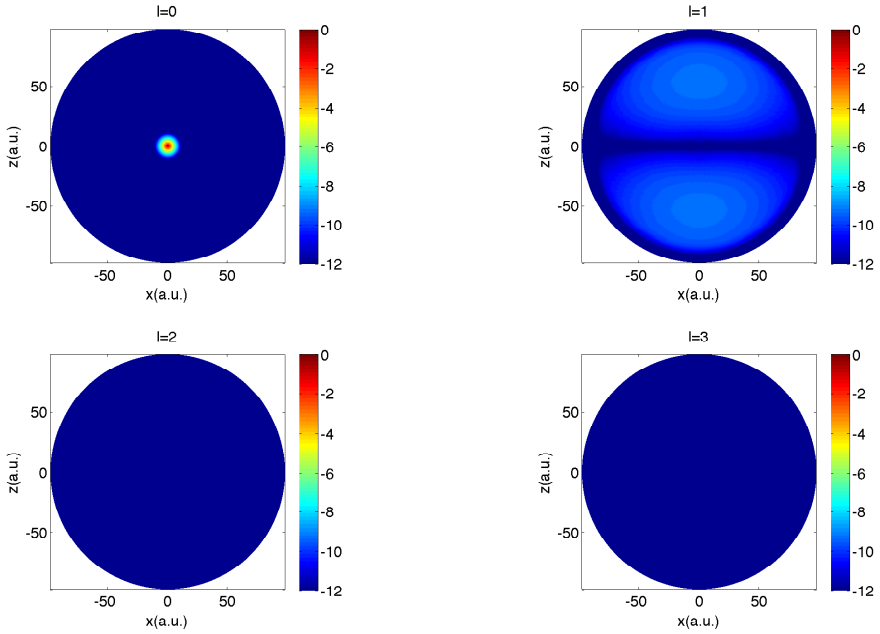


Figure 3–10: The distribution of wave function on different angular momentum of the final state in Fig. 3–9 in ln scale, the ionization rate is mainly attributed to the $l = 1$ state

3.3.4 Revised Algorithm for Circular Polarized Laser Pulses

When we apply circular polarized laser pulses to atoms, the magnetic quantum number m no longer conserves. Therefore, we expand a wave function as

$$\psi(r,\theta,\phi) = \sum_{l=0}^{l_{max}} \sum_{m=-l}^{l} \sum_{i=0}^{N_l} c_{il}^{n} \frac{B_i^k}{r} Y_l^m(\theta,\phi) \tag{3–153}$$

$$i\frac{\partial}{\partial t}\psi(t) = [H_0 + D(t)]\psi(t) \tag{3–154}$$

The TDSE is still in the same form, the only difference is that the elements has the m index.

$$i\mathbf{S}\cdot\dot{\mathbf{c}}(t) = (\bar{\mathbf{H}}_0 + \bar{\mathbf{D}}(t))\cdot\mathbf{c}(t) \tag{3–155}$$

The elements of **S**

$$S_{ilm,i'l'm'} = \delta_{ll'}\delta_{mm'}\int_0^{r_{max}} B_i^k(r)B_{i'}^k(r)dr \tag{3–156}$$

$$\dot{c}(t) = \frac{c(t+\frac{\delta_t}{2}) - c(t-\frac{\delta_t}{2})}{\delta t} \tag{3–157}$$

$$c(t) = \frac{c(t+\frac{\delta_t}{2}) + c(t-\frac{\delta_t}{2})}{2} \tag{3–158}$$

The elements of $\bar{\mathbf{H}}_0$

$$\begin{aligned}(\bar{\mathbf{H}}_0)_{ilm,i'l'm'} = \delta_{ll'}\delta_{mm'}[-\frac{1}{2}(\int_0^{r_{max}} B_i^k(r)(\frac{\partial^2 B_{i'}^k(r)}{\partial r^2} - l'(l'+1)\frac{B_{i'}^k(r)}{r^2})dr) \\ + (\int_0^{r_{max}} B_i^k(r) - \frac{Z}{r}B_{i'}^k(r)dr]\end{aligned} \tag{3–159}$$

The elements of $\bar{\mathbf{D}}(t)$

$$\begin{aligned}\bar{\mathbf{D}}(t)_{ilm,i'l'm'} =& < ilm|E_x r\sin(\theta)\frac{e^{i\phi}+e^{-i\phi}}{2}|i'l'm'> \\ &+ < ilm|E_y r\sin(\theta)\frac{e^{i\phi}-e^{-i\phi}}{2i}|i'l'm'>\end{aligned} \tag{3–160}$$

Now we need to use Y_1^1 and Y_1^{-1} to replace $\sin(\theta)\cos(\phi)$ and $\sin(\theta)\sin(\phi)$ in order to use Wigner 3-j symbols.

$$Y_1^1 = -\frac{1}{2}\sqrt{\frac{3}{2\pi}}\sin(\theta)e^{i\phi} \tag{3–161}$$

$$Y_1^{-1} = \frac{1}{2}\sqrt{\frac{3}{2\pi}}\sin(\theta)e^{-i\phi} \tag{3–162}$$

$$\begin{aligned}&< Y_{lm}|\sin(\theta)\cos(\phi)|Y_{l'm'}(\Omega)> \\ =&< Y_{lm}|\sin(\theta)\frac{e^{i\phi}+e^{-i\phi}}{2}|Y_{l'm'}(\Omega)> \\ =&\sqrt{\frac{2\pi}{3}} < Y_{lm}|-Y_{11}+Y_{1-1}|Y_{l'm'}>\end{aligned} \tag{3–163}$$

$$\begin{aligned} < Y_{lm}|Y_{11}|Y_{l'm} >&= \int (Y_l^m)^* Y_1^1 Y_{l'}^m d\Omega \\ &= (-1)^m \sqrt{\frac{(2l+1)3(2l'+1)}{4\pi}} \begin{pmatrix} l & 1 & l' \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} l & 1 & l' \\ -m & 1 & m \end{pmatrix} \end{aligned} \tag{3–164}$$

$$\begin{pmatrix} l & 1 & l' \\ 0 & 0 & 0 \end{pmatrix} = \frac{(-1)^{l-1}}{\sqrt{2l'+1}} < l010|l'0 > \tag{3–165}$$

$$\begin{pmatrix} l & 1 & l' \\ -m & 1 & m' \end{pmatrix} = \frac{(-1)^{l-1-m}}{\sqrt{2l'+1}} < l-m11|l'-m > \tag{3–166}$$

$$\begin{aligned} &< Y_{lm}(\Omega)|Y_{11}|Y_{l'm'}(\Omega) > \\ &\quad = \sqrt{\frac{3}{4\pi}\frac{2l+1}{2l'+1}} < l010|l'0 >< l-m11|l'-m' > \end{aligned} \tag{3–167}$$

After simplification, we can get

$$\begin{aligned} < Y_{lm}|Y_{11}|Y_{l'm'} >=& \sqrt{\frac{3}{4\pi}}\delta_{m,m'+1}(\delta_{l,l'+1}\sqrt{\frac{(l+m-1)(l+m)}{2(2l-1)(2l+1)}} \\ &- \delta_{l,l'-1}\sqrt{\frac{(l-m+1)(l-m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}}) \end{aligned} \tag{3–168}$$

$$\begin{aligned} < Y_{lm}|Y_{1-1}|Y_{l'm'} >=& \sqrt{\frac{3}{4\pi}}\delta_{m,m'-1}(\delta_{l,l'+1}\sqrt{\frac{(l-m-1)(l-m)}{2(2l-1)(2l+1)}} \\ &- \delta_{l,l'-1}\sqrt{\frac{(l+m+1)(l+m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}}) \end{aligned} \tag{3–169}$$

Now the origin integral equals to

$$\begin{aligned}
\sqrt{\frac{2\pi}{3}} < Y_{lm}| - Y_{11} + Y_{1-1}|Y_{l'm'} >= \\
- \sqrt{\frac{1}{2}}\delta_{m,m'+1}(\delta_{l,l'+1}\sqrt{\frac{(l+m-1)(l+m)}{2(2l-1)(2l+1)}} \\
- \delta_{l,l'-1}\sqrt{\frac{(l-m+1)(l-m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}}) \\
+\sqrt{\frac{1}{2}}\delta_{m,m'-1}(\delta_{l,l'+1}\sqrt{\frac{(l-m-1)(l-m)}{2(2l-1)(2l+1)}} \\
- \delta_{l,l'-1}\sqrt{\frac{(l+m+1)(l+m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}})
\end{aligned} \tag{3–170}$$

For the same reason

$$\begin{aligned}
< l,m|\sin(\theta)\frac{e^{i\phi}-e^{-i\phi}}{2i}|l'm' > \\
=i\sqrt{\frac{2\pi}{3}} < Y_{lm}|Y_{11} + Y_{1-1}|Y_{l'm'} > \\
=i[\sqrt{\frac{1}{2}}\delta_{m,m'+1}(\delta_{l,l'+1}\sqrt{\frac{(l+m-1)(l+m)}{2(2l-1)(2l+1)}} \\
- \delta_{l,l'-1}\sqrt{\frac{(l-m+1)(l-m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}}) \\
+\sqrt{\frac{1}{2}}\delta_{m,m'-1}(\delta_{l,l'+1}\sqrt{\frac{(l-m-1)(l-m)}{2(2l-1)(2l+1)}} \\
- \delta_{l,l'-1}\sqrt{\frac{(l+m+1)(l+m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}})]
\end{aligned} \tag{3–171}$$

The laser integral elements now can be written as

$$\begin{aligned}\bar{\mathbf{D}}(t)_{ilm,i'l'm'} = E_x \cdot [&- \sqrt{\frac{1}{2}}\delta_{m,m'+1}(\delta_{l,l'+1}\sqrt{\frac{(l+m-1)(l+m)}{2(2l-1)(2l+1)}} \\ &- \delta_{l,l'-1}\sqrt{\frac{(l-m+1)(l-m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}}) \\ &+\sqrt{\frac{1}{2}}\delta_{m,m'-1}(\delta_{l,l'+1}\sqrt{\frac{(l-m-1)(l-m)}{2(2l-1)(2l+1)}} \\ &- \delta_{l,l'-1}\sqrt{\frac{(l+m+1)(l+m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}})] \\ &iE_y[\sqrt{\frac{1}{2}}\delta_{m,m'+1}(\delta_{l,l'+1}\sqrt{\frac{(l+m-1)(l+m)}{2(2l-1)(2l+1)}} \\ &- \delta_{l,l'-1}\sqrt{\frac{(l-m+1)(l-m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}}) \\ &+\sqrt{\frac{1}{2}}\delta_{m,m'-1}(\delta_{l,l'+1}\sqrt{\frac{(l-m-1)(l-m)}{2(2l-1)(2l+1)}} \\ &- \delta_{l,l'-1}\sqrt{\frac{(l+m+1)(l+m+2)(l+1)}{(2l+1)(2l+2)(2l+3)}})] \int_0^{r_{max}} B_i^k(r) r B_{i'}^k(r) dr\end{aligned} \tag{3–172}$$

With this result, we can now solve TDSE under circular polarized laser pulses. The Fig. 3–11 is the evolution under $I = 3.5 \times 10^{14} W/cm^2 (0.1 \text{ a.u.}), \omega = 0.8, L = 3T$), the laser pulse is in the format of $E_x(t) = E_0 \sin(\omega t) \sin^2(\pi t/L), E_y(t) = E_0 \cos(\omega t) \sin^2(\pi t/L)$. Also, the components of anticlockwise wave function in Fig. 3–12 is easily revealed to have the negative m.

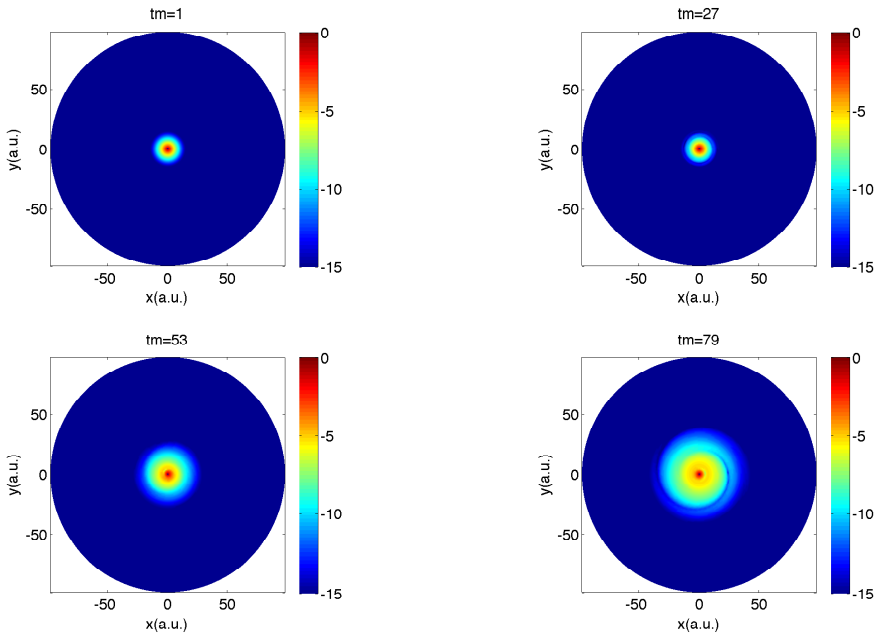


Figure 3–11: The wave function evolution under the circular polarized laser pulse with $I = 3.5 \times 10^{14} W/cm^2 (0.1 \text{ a.u.}), \omega = 0.8, L = 3T$), $E_x(t) = E_0 \sin(\omega t) \sin^2(\pi t/L), E_y(t) = E_0 \cos(\omega t) \sin^2(\pi t/L)$ in ln scale.

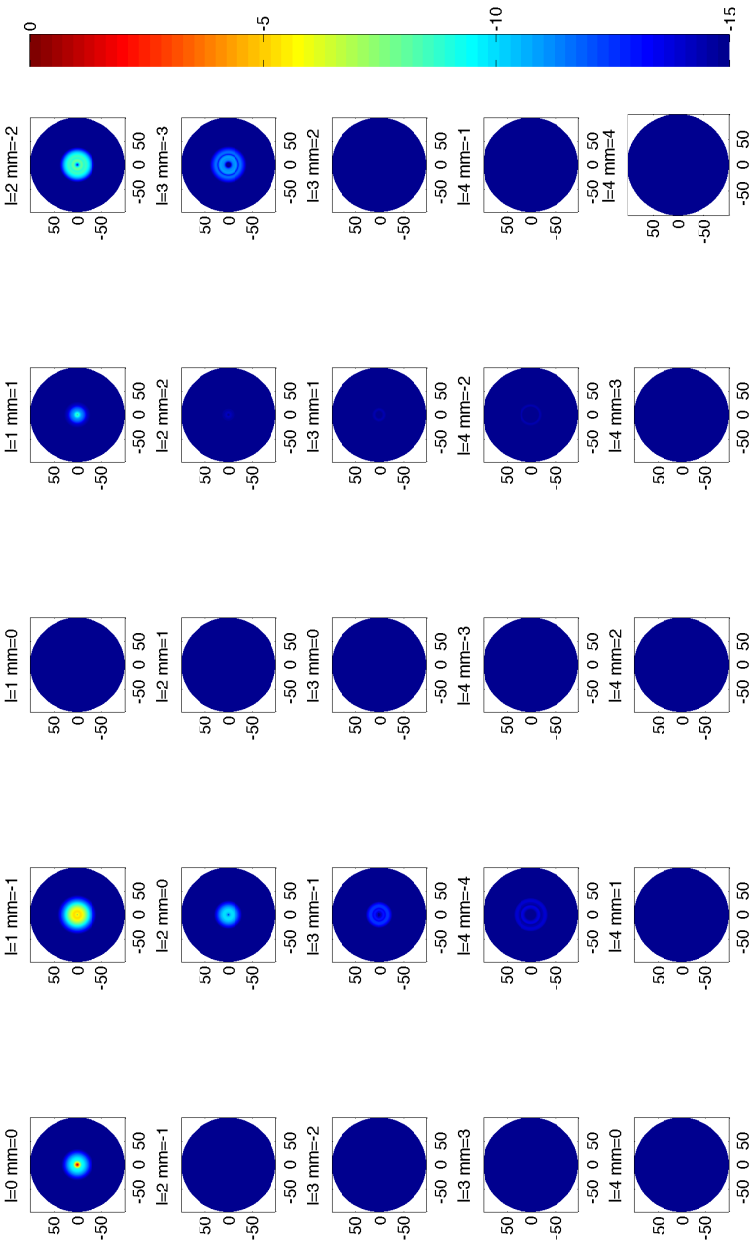


Figure 3–12: The distribution of wave function on different angular momentum and magnetic quantum number of the final state in Fig. 3–11 in ln scale, we can easily see that the anticlockwise wave function are consisted by the $m < 0$ states.

3.4 Summary

In this chapter, we construct a faster and more accurate algorithm based on the exiting methods and knowledge. With this algorithm, we can generate the eigenstate with specific angular momentum number and magnetic quantum number. Moreover, the propagation of wave function can be observed from the aspect of wave functions on different quantum numbers.

Conclusion

To sum up, we introduce the existing algorithms for the simulations in ultrafast and ultrastrong physics in the first part. The image time propagation method is applied to the generation of initial state and the time-dependent Schrödinger equation is solved by the Fourier transformation method and Crank-Nicolson method. Then, we applied these methods to the researches such as ionization dependence of corotating and counterrotating laser pulses and the interesting phenomena along with charge-resonance-enhanced ionization. Secondly, we construct a faster and more accurate algorithm with the existing methods and ideas. The B-spline is applied for simplify the matrix form and increase the accuracy through analytic formulas. With this more powerful tool, we are no longer bothered by the time cost and the accuracy. Furthermore, this tool can help us get more information about the wave function and can be extended to multi nucleuses and multi electrons models.

Of course, scientific researches will never end and our explanations and algorithms are not the best. Hoping there are some further researches on the topic we studied.

Appendix A Code

```
/*******************************
 April-07-2015 Xinyuan You
 b_spline h2+
 1. calculation of bspline and its derivative and second derivative
 2. calculation the overlap intergal S
 3. calcualtion of the kinetic term
 4. calculation of the potential term
 5. generation of the S matrix and H matrix
 6. change to the k=9 and exponential breakpoints
 7. call LAPACK to calculate S\H and eigenvalue and eigenvectors
 8. add the centrifugal term
 *******************************/

/*******************************
 April-20-2015 Xinyuan You
 2nd Edition     for h2+
 1. include the function of Clebsch Gorden coefficient
 2. define the equisdant knot sequence
 3. define l_max, Nl and R(internuclear distance)
 4. calculate the S intergral
 5. calculate the kinetic term in H
 6. calculate the nuclear potential term in H
 7. calculate the electronic potential term in H
 8. generate the H and S matrix
 9. call LAPACK to calculate S\H and eigenvalue and eigenvectors
 10. use MALLOC to define the huge n_lapack*n_lapack array
 11. differ the basis expansion l_max, and the potential expansion lv_max
 *******************************/

/*******************************
 January-29-2016 Lin Xin
 3nd Edition     for h2+
 1. change the interval of bspline integral to [index_max, index_min+k-1], reduce the time cost to one
      fourth
 2. apply the judgement statment in function electronic_potential_integral thus avoid the unnecessary
      calculation of integral, reduce the time cost to one second
 3. define r_temp for transformation of x_gl
 4. change the number of the point of Gaussian Legendre intergral since n GL point is exact for a 2n+1
      polynomial in a closed term, the H{ij} has the maximum degree of 2k-1 for potential term
```

```
37   5. define the singularity, multiple linear, multiple expoential sequence
38   6. fix the define problem
39   7. fix the type problem of (ll+1)/(ll_p+1) and other /
40   8. when meet the segementation fault, first enter superuser then ulimit -s memory size
41   9. calculate the plot points r
42   10. calculate the unification factor of the wavefunction
43   11. calculate the wave function along the z axis
44   12. seperate the procedure of wave function to another code to save time
45   13. dynamically allocate the s_matrix and h_matrix
46   ********************************/
47
48  /********************************
49   April-11-2016 Lin Xin
50   4nd Edition     for h under linear polarized laser
51   1. define the intergral for the static potential, laser potential, A matrix, b vector
52   2. define the laser pulse , generation of the time step
53   3. define charge m_load delta_t E_z omega tm_min T tm_max
54   4. update the version of unification and gen_w
55   5. define the gen_a_b_matrix for efficiency
56   6. define out_tm
57   7. calcualte the time dependent schrodinger euqation
58   ********************************/
59
60  /********************************
61   April-22-2016 Lin Xin
62   5th Edition     for h under circular polarized laser
63   1. update every function to the ll,mm grid
64   2. define dx_matrix, dy_matrix, dz_matrix
65   3. define E_x, E_y, E_z, E_x_vec, E_y_vec, E_z_vec
66   4. define ludcmp lubksb to get rid of lapack
67   5. update to fit the c89
68   ********************************/
69
70  /********************************
71   May-16-2016 Lin Xin
72   6th Edition for h under circular polarized laser
73   1. define gen_b_spline_matrix gen_dd_spline_matrix gen_r_temp to save cost
74   2. define struct para, matrix to package the parameters and matrix and revise the entrie program so that
         avoid globlal parameters and make the input much easier.
75   3. fix the global variable problem and now can use lapack for faster calculation.
76   ********************************/
77
78  #include <math.h>
79  #include <complex.h>
80  #include <stdlib.h>
81  #include <stdio.h>
```

```
#include <time.h>
#include <string.h>
#include "lapacke.h"
#include "lapacke_config.h"
#include "lapacke_mangling.h"
#include "lapacke_utils.h"



struct para
{
    int l;//  the number of segment, break points -1
    int k;//  the order of b_spline
    int n;//  the number of b_spline in a complete set
    int n_knot;// the number of knots
    int n_gl;// the point of Gaussian Legendre integral
    int n_lapack;// lapack call reduce b_spline remove b_1 and b_n [note" here must be number] also
        multipy angular 1
    int n_load;//the size of the initial load eigenvector and other load quantity
    int seq_load; //the initial eigenvector of the sequence in energy in seq_load
    int l_load;////the initial eigenvector with angular momentum number l_load
    int m_load; //the initial eigenvector with magnetic quantum number m_load
    int l_max;// the highest angular momentum considered in the basis expansion
    double charge;//The charge of atom nuclus
    double r_min;// start value of radius grid
    double r_max;// end value of radius grid
    const char *dist;// the grid distribution method
    double epsilon;// remove the singularity in the denominator
    double delta_t;// the time step for propragation
    double tm_min;//the start point of the time
    double tm_max;//the end point of the time
    int tm_size;//the size of t
    double omega;// the angular frequency of the laser
    double T;// the period of the laser pulse
    double E_x;//the amplitude for the laser in length gauge in x direction
    double E_y;//the amplitude for the laser in length gauge in y direction
    double E_z;//the amplitude for the laser in length gauge in z direction
    double mask_prop;// the mask propotion
    int out_tm;// the time step for wavefunction output
};

struct matrix
{
    double *x_gl;// gauss integral point
    double *w_gl;// gauss integral weight
    double *t;  // radius grid with the first point and last point
```

```
    double *tm; // time grid
    double *r_temp;// radius grid plus the gauss integral seperation
    double *r;//radius grid for plotting
    double **bspline_matrix; //bspline matrix pre calcualted
    double **dd_bspline_matrix; //dd_bpsline_matrix pre calculated
    double **bspline_matrix_plot;// bspline matrix for plot pre calculated
    double *mask_vector;
};

/*****
#include "nrutil.c"
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}
#define TINY 1.0e-20

void ludcmp(double **a, int n_lu, int *indx, double *d)
//Given a matrix a[1..n][1..n], this routine replaces it by the LU decomposition of a rowwise permutation
      of itself. a and n are input. a is output, arranged as in equation (2.3.14) above; indx[1..n] is an
      output vector that records the row permutation effected by the partial pivoting; d is output as ±1
     depending on whether the number of row interchanges was even or odd, respectively. This routine is
     used in combination with lubksb to solve linear equations or invert a matrix.
{
    int i_lu,imax,j_lu,k_lu;
    double big,dum,sum,temp;
    double *vv;
    vv=vector(1,n_lu);
    *d=1.0;
    for (i_lu=1;i_lu<=n_lu;i_lu++)
    {
        big=0.0;
        for (j_lu=1;j_lu<=n_lu;j_lu++)
            //vv stores the implicit scaling of each row.No row interchanges yet. Loop over rows to get
                the implicit scaling information.
            if ((temp=fabs(a[i_lu][j_lu])) > big) big=temp;
        if (big == 0.0) nrerror("Singular matrix in routine ludcmp"); //No nonzero largest element.
        vv[i_lu]=1.0/big;//Save the scaling.
    }
    for (j_lu=1;j_lu<=n_lu;j_lu++)
    {
        for (i_lu=1;i_lu<j_lu;i_lu++)
        {
            sum=a[i_lu][j_lu];
            //This is the loop over columns of Crout's method.
            //This is equation (2.3.12) except for i = j.
            for (k_lu=1;k_lu<i_lu;k_lu++) sum -= a[i_lu][k_lu]*a[k_lu][j_lu];
            a[i_lu][j_lu]=sum;
        }
```

```
168         big=0.0;
169         for (i_lu=j_lu;i_lu<=n_lu;i_lu++)
170         {
171             sum=a[i_lu][j_lu];
172             for (k_lu=1;k_lu<j_lu;k_lu++)
173                 //Initialize for the search for largest pivot element. This is i = j of equation (2.3.12)
                        and i = j+1...N of equation (2.3.13).
174                 sum -= a[i_lu][k_lu]*a[k_lu][j_lu]; a[i_lu][j_lu]=sum;
175             if ( (dum=vv[i_lu]*fabs(sum)) >= big) {
176                 //Is the figure of merit for the pivot better than the best so far?
177                 big=dum;
178                 imax=i_lu;
179             }
180         }
181         if (j_lu != imax) {
182             for (k_lu=1;k_lu<=n_lu;k_lu++) {
183                 dum=a[imax][k_lu];
184                 a[imax][k_lu]=a[j_lu][k_lu];
185                 a[j_lu][k_lu]=dum;
186             }
187             *d = -(*d);
188             vv[imax]=vv[j_lu];
189             //Do we need to interchange rows? Yes, do so and change the parity of d. Also interchange the
                    scale factor.
190         }
191         indx[j_lu]=imax;
192         if (a[j_lu][j_lu] == 0.0) a[j_lu][j_lu]=TINY;
193         //If the pivot element is zero the matrix is singular (at least to the precision of the algorithm
             ). For some applications on singular matrices, it is desirable to substitute TINY for zero.
194         if (j_lu != n_lu) { //Now, finally, divide by the pivot element.
195             dum=1.0/(a[j_lu][j_lu]);
196             for (i_lu=j_lu+1;i_lu<=n_lu;i_lu++) a[i_lu][j_lu] *= dum;
197         }
198
199     } //Go back for the next column in the reduction.
200     free_vector(vv,1,n_lu);
201 }
202
203 void lubksb(double **a, int n_lu, int *indx, double b[])
204 //Solves the set of n linear equations A·X = B. Here a[1..n][1..n] is input, not as the matrix A but
      rather as its LU decomposition, determined by the routine ludcmp. indx[1..n] is input as the
      permutation vector returned by ludcmp. b[1..n] is input as the right-hand side vector B, and returns
       with the solution vector X. a, n, and indx are not modified by this routine and can be left in
      place for successive calls with different right-hand sides b. This routine takes into account the
      possibility that b will begin with many zero elements, so it is efficient for use in matrix
      inversion.
```

```
{
    int i,ii=0,ip,j;
    double sum;

    for (i=1;i<=n_lu;i++) {
        ip=indx[i];
        sum=b[ip];
        b[ip]=b[i];
        if (ii)
            //When ii is set to a positive value, it will become the index of the first nonvanishing
                element of b. We now do the forward substitution, equation (2.3.6). The only new wrinkle
                 is to unscramble the permutation as we go.
            for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
        else if (sum) ii=i;
        b[i]=sum;
    }
    for (i=n_lu;i>=1;i--) {
        //A nonzero element was encountered, so from now on we will have to do the sums in the loop above
             .
        //Now we do the backsubstitution, equation (2.3.7).
        sum=b[i];
        for (j=i+1;j<=n_lu;j++)sum -= a[i][j]*b[j];
        b[i]=sum/a[i][i]; //Store a component of the solution vector X.
    }
}
*******/


void gen_x_w_gl(struct para p, struct matrix m)
{
    if(p.n_gl==3)
    {
        /***** GL for 3 points *****/
        //  the GL x
        double x_gl_data[3+1]={0, -0.774596669241483377035853 1,0,
        0.774596669241483377035853 1};

        //  the GL w
        double w_gl_data[3+1]={0, 0.555555555555555555555556,0.888888888888888888888889,
         0.555555555555555555555556};
        int i;
        for (i=1; i<=p.n_gl+1; i++)
        {
            m.x_gl[i]=x_gl_data[i];
        }
        for (i=1; i<=p.n_gl+1; i++)
```

```
248         {
249             m.w_gl[i]=w_gl_data[i];
250         }
251     }
252     if(p.n_gl==4)
253     {
254         double x_gl_data[4+1]={0, -0.86113631159405257522394 65, -0.33998104358485626480266 58,
255         0.33998104358485626480266 58,0.86113631159405257522394 65};
256 
257         //  the GL w
258         double w_gl_data[4+1]={0, 0.65214515486254614262693 61, 0.34785484513745385737306 39,
259         0.34785484513745385737306 39, 0.65214515486254614262693 61};
260         int i;
261         for (i=1; i<=p.n_gl+1; i++)
262         {
263             m.x_gl[i]=x_gl_data[i];
264         }
265         for (i=1; i<=p.n_gl+1; i++)
266         {
267             m.w_gl[i]=w_gl_data[i];
268         }
269     }
270 
271     if(p.n_gl==5)
272     {
273         /****** GL for 5 points ******/
274         //  the GL x
275         double x_gl_data[5+1]={0, -0.90617984593866399279762 69, -0.53846931010568309103631 44, 0,
                0.53846931010568309103631 44, 0.90617984593866399279762 69};
276         //  the GL w
277         double w_gl_data[5+1]={0, 0.23692688505618908751426 40, 0.47862867049936646804129 15,
                0.56888888888888888888888 889, 0.47862867049936646804129 15, 0.23692688505618908751426 40};
278         int i;
279         for (i=1; i<=p.n_gl+1; i++)
280         {
281             m.x_gl[i]=x_gl_data[i];
282         }
283         for (i=1; i<=p.n_gl+1; i++)
284         {
285             m.w_gl[i]=w_gl_data[i];
286         }
287     }
288     if(p.n_gl==7)
289     {
290         /***** GL for 7 points ******/
291         //  the GL x
```

```
292         double x_gl_data[7+1]={0, -0.94910791234275852452618 97, -0.74153118559939443986386 48,
                -0.40584515137739716690660 64, 0, +0.40584515137739716690660 64, +0.74153118559939443986386 48,
                 +0.94910791234275852452618 97};
293
294         //  the GL w
295         double w_gl_data[7+1]={0, 0.12948496616886969327061 14, 0.27970539148927666790146 78,
                0.38183005050511894495036 98, 0.41795918367346938775510 20,  0.38183005050511894495036 98,
                0.27970539148927666790146 78,  0.12948496616886969327061 14};
296         int i;
297         for (i=1; i<=p.n_gl+1; i++)
298         {
299             m.x_gl[i]=x_gl_data[i];
300         }
301         for (i=1; i<=p.n_gl+1; i++)
302         {
303             m.w_gl[i]=w_gl_data[i];
304         }
305     }
306     if(p.n_gl==8)
307     {
308         /******* GL for 8 points *******/
309         //  the GL x
310         double x_gl_data[8+1]={0, -0.96028985649753623, -0.79666647741362674, -0.52553240991632899,
                -0.18343464249565045, 0.18343464249565045,
311         0.52553240991632899, 0.79666647741362674, 0.96028985649753623};
312         int i;
313         for (i=1; i<=p.n_gl+1; i++)
314         {
315             m.x_gl[i]=x_gl_data[i];
316         }
317
318
319         //  the GL w
320         double w_gl_data[8+1]={0, 0.10122853629037626, 0.22238103445337447,
321          0.31370664587788728, 0.36268378337836198, 0.36268378337836198,
322          0.31370664587788728, 0.22238103445337447,  0.10122853629037626};
323
324         for (i=1; i<=p.n_gl+1; i++)
325         {
326             m.w_gl[i]=w_gl_data[i];
327         }
328     }
329     if(p.n_gl==9)
330     {
331         /****  GL for 9 points *****/
332         //  the GL x
```

```
333         double x_gl_data[9+1]={0, -0.968160239507626, -0.836031107326636, -0.613371432700590,
                -0.324253423403809, 0, 0.324253423403809,
334          0.613371432700590, 0.836031107326636,  0.968160239507626};
335
336         //  the GL w
337         double w_gl_data[9+1]={0, 0.330239355001260, 0.312347077040003, 0.312347077040003,
338          0.260610696402935, 0.260610696402935, 0.180648160694857,
339          0.180648160694857, 0.081274388361574, 0.081274388361574};
340         int i;
341         for (i=1; i<=p.n_gl+1; i++)
342         {
343             m.x_gl[i]=x_gl_data[i];
344         }
345         for (i=1; i<=p.n_gl+1; i++)
346         {
347             m.w_gl[i]=w_gl_data[i];
348         }
349     }
350     if(p.n_gl==10)
351     {
352         /***** GL for 10 points ******/
353         //  the GL x
354         double x_gl_data[10+1]={0,-0.973906528517171720077964 0,-0.865063366688984510732096 7,
355 -0.679409568299024406234327 4,-0.433395394129247190799265 9,
356 -0.148874338981631210884826 0,0.148874338981631210884826 0,
357 0.433395394129247190799265 9,0.679409568299024406234327 4,
358 0.865063366688984510732096 7,0.973906528517171720077964 0};
359
360         //  the GL w
361         double w_gl_data[10+1]={0,0.066671344308688137593568 8,0.149451349150580593145776 3,
362         0.219086362515982043995534 9,0.269266719309996355091226 9,
                0.295524224714752870173893 0,0.295524224714752870173893 0,
363         0.269266719309996355091226 9,0.219086362515982043995534 9,
364         0.149451349150580593145776 3,0.066671344308688137593568 8};
365         int i;
366         for (i=1; i<=p.n_gl+1; i++)
367         {
368             m.x_gl[i]=x_gl_data[i];
369         }
370         for (i=1; i<=p.n_gl+1; i++)
371         {
372             m.w_gl[i]=w_gl_data[i];
373         }
374     }
375     if(p.n_gl==16)
376     {
```

```
377         /******* GL for 16 points ******/
378         //  the GL x
379         double x_gl_data[16+1]={0, -0.989400934991650, -0.944575023073233,
380          -0.865631202387832, -0.755404408355003, -0.617876244402644,
381          -0.458016777657227, -0.281603550779259, -0.095012509837637,
382          0.095012509837637, 0.281603550779259, 0.458016777657227,
383          0.617876244402644, 0.755404408355003, 0.865631202387832,
384          0.944575023073233,  0.989400934991650};
385         //  the GL w
386         double w_gl_data[16+1]={0, 0.027152459411754, 0.062253523938648,
387         0.095158511682493, 0.124628971255534, 0.149595988816577,
388         0.169156519395003, 0.18260341504492, 0.189450610455069,
389         0.189450610455069, 0.18260341504492, 0.169156519395003,
390          0.149595988816577, 0.124628971255534, 0.095158511682493,
391           0.062253523938648,   0.027152459411754};
392         int i;
393         for (i=1; i<=p.n_gl+1; i++)
394         {
395             m.x_gl[i]=x_gl_data[i];
396         }
397         for (i=1; i<=p.n_gl+1; i++)
398         {
399             m.w_gl[i]=w_gl_data[i];
400         }
401     }
402 }
403
404 //  define the knot sequence t[]
405 void gen_t(struct para p, struct matrix m)
406 {
407     //  define the two boundarys
408     int i;
409     for (i=1; i<=p.k; i++) {
410         m.t[i]=p.r_min;
411     }
412     for (i=p.n_knot-p.k+1; i<=p.n_knot; i++) {
413         m.t[i]=p.r_max;
414     }
415
416
417     //  define the interior points  (equidistant)
418     if(strcmp(p.dist,"equidstant")==0)
419     {
420         for ( i=p.k; i<=p.n_knot-p.k+1; i++)
421         {
422             m.t[i]=p.r_min+(p.r_max-p.r_min)*(i-p.k)/(p.n_knot-2*p.k+1);
```

```
423         }
424     }
425     else if(strcmp(p.dist,"expoential")==0)
426     {
427         //  define the interior points (expoential)
428         double gama=5;
429         for ( i=p.k; i<=p.n_knot-p.k+1; i++)
430         {
431             m.t[i]=p.r_min+(p.r_max-p.r_min)*(exp(gama*(i-p.k)/(p.n_knot-2*p.k+1))-1)/(exp(gama)-1);
432         }
433     }
434     else
435     {
436         printf("No distribution mode defined");
437     }
438
439 }
440
441 void gen_r_temp(struct para p, struct matrix m)
442 {
443     int ii,jj;
444     for(ii=2;ii<=(p.n-1);ii++)
445     {
446         for (jj=1; jj<=p.n_gl; jj++)
447         {
448             m.r_temp[(ii-2)*p.n_gl+jj-1]=((m.t[ii+1]-m.t[ii])*m.x_gl[jj]+(m.t[ii+1]+m.t[ii]))/2;
449         }
450     }
451 }
452
453 // generate the radius grid for plotting start from the second point
454 void gen_r(struct para p, struct matrix m)
455 {
456     int i;
457     for (i=2; i<=p.n-1; i++)
458     {
459         m.r[i-2]=m.t[i];
460     }
461 }
462
463 // define time step sequence
464 void gen_tm(struct para p, struct matrix m)
465 {
466     int i;
467     for (i=1;i<=p.tm_size;i++)
468     {
```

```
469         m.tm[i]=p.tm_min+p.delta_t*(i-1);
470     }
471 }
472
473 //define the laser vector
474 void gen_laser(double *E_x_vec, double *E_y_vec, double *E_z_vec, struct para p , struct matrix m)
475 {
476     int i;
477     for (i=1;i<=p.tm_size;i++)
478     {
479         // sine shape
480         E_x_vec[i]=p.E_x*sin(p.omega*m.tm[i])*pow(sin(M_PI*m.tm[i]/p.tm_max),2);
481         E_y_vec[i]=p.E_y*cos(p.omega*m.tm[i])*pow(sin(M_PI*m.tm[i]/p.tm_max),2);
482         E_z_vec[i]=p.E_z*sin(p.omega*m.tm[i])*pow(sin(M_PI*m.tm[i]/p.tm_max),2);
483         //E_x_vec[i]=p.E_x*(sin(p.omega*m.tm[i])*pow(sin(M_PI*m.tm[i]/p.tm_max),2)-M_PI/(p.omega*p.tm_max
                )*sin(2*M_PI*m.tm[i]/p.tm_max)*cos(p.omega*m.tm[i]));
484         //E_y_vec[i]=p.E_y*(sin(p.omega*m.tm[i])*pow(sin(M_PI*m.tm[i]/p.tm_max),2)-M_PI/(p.omega*p.tm_max
                )*sin(2*M_PI*m.tm[i]/p.tm_max)*cos(p.omega*m.tm[i]));
485         //E_z_vec[i]=p.E_z*(sin(p.omega*m.tm[i])*pow(sin(M_PI*m.tm[i]/p.tm_max),2)-M_PI/(p.omega*p.tm_max
                )*sin(2*M_PI*m.tm[i]/p.tm_max)*cos(p.omega*m.tm[i]));
486     }
487 }
488
489 // generate the mask vector for prevent the reflection
490 void gen_mask_vector(struct para p, struct matrix m)
491 {
492     double L=m.r[p.n-3]-m.r[0];
493     double r_dist=L*p.mask_prop;
494     double r_0=m.r[p.n-3]-r_dist;
495     int i;
496     for (i=2; i<=p.n-1; i++)
497     {
498         if (m.r[i-2]<r_0)
499         {
500             m.mask_vector[i-2]=1;
501         }
502         else
503         {
504             m.mask_vector[i-2]=pow(cos((m.r[i-2]-r_0-p.epsilon)/(r_dist)*M_PI/2),1.0/6.0);
505         }
506     }
507 }
508
509 // mask the radio wave function with the mask matrix
510 void mask(double complex *c, double complex *c_new, struct para p, struct matrix m )
511 {
```

```
    int ll , mm , i;
    for (ll=0; ll<=p.l_max; ll++)
    {
        for( mm = -ll; mm <= ll; mm ++)
        {
            for (i=2; i<=p.n-1; i++)
            {
                c_new[(ll*(ll+1)+mm)*(p.n-2)+i-2]=c[(ll*(ll+1)+mm)*(p.n-2)+i-2]*m.mask_vector[i];
            }
        }
    }

}


// iteration method generaiton of b_spline
double bspline(int kk, int i, double x, struct para p, struct matrix m)
{
    double  output;
    if (kk==1)
    {
        if (x>=m.t[i] && x<m.t[i+1])
            output=1;
        else
            output=0;
    }
    else
        output=(x-m.t[i])/(m.t[i+kk-1]-m.t[i]+p.epsilon)*bspline(kk-1, i, x, p, m)+(m.t[i+kk]-x)/(m.t[i+
            kk]-m.t[i+1]+p.epsilon)*bspline(kk-1, i+1, x, p, m);
    return output;
}

//  iteration method generation of derivitive of b_spline D[B]
double d_bspline(int kk, int i, double x, struct para p, struct matrix m )
{
    //  declaration
    double bspline();

    double output;

    output=(kk-1+0.0)/(m.t[i+kk-1]-m.t[i]+p.epsilon+0.0)*bspline(kk-1, i, x, p, m)-(kk-1+0.0)/(m.t[i+kk]-
        m.t[i+1]+p.epsilon+0.0)*bspline(kk-1, i+1, x, p , m );

    return output;
}
```

```
556 //  iteration method geneartion of second derivative of bspline D^2[B]
557 double dd_bspline(int kk, int i, double x,  struct para p, struct matrix m )
558 {
559     //  declaration
560     double d_bspline();
561 
562     double output;
563 
564     output=(kk-1+0.0)/(m.t[i+kk-1]-m.t[i]+p.epsilon+0.0)*d_bspline(kk-1, i, x , p, m)-(kk-1+0.0)/(m.t[i+
          kk]-m.t[i+1]+p.epsilon+0.0)*d_bspline(kk-1, i+1, x , p , m);
565 
566     return output;
567 }
568 
569 // previously get the matrix value of the bspline
570 void gen_bspline_matrix(struct para p, struct matrix m)
571 {
572     double bspline();
573     int i;
574     int ii,jj;
575     for (i=2; i<=(p.n-1); i++)
576     {
577         for(ii=2;ii<=(p.n-1);ii++)
578         {
579             for (jj=1; jj<=p.n_gl; jj++)
580             {
581                 m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]=bspline(p.k,i,m.r_temp[(ii-2)*p.n_gl+jj-1], p ,
                        m);
582             }
583         }
584     }
585 }
586 
587 void gen_dd_bspline_matrix(struct para p, struct matrix m)
588 {
589     double dd_bspline();
590     int i;
591     int ii,jj;
592     for (i=2; i<=(p.n-1); i++)
593     {
594         for(ii=2;ii<=(p.n-1);ii++)
595         {
596             for (jj=1; jj<=p.n_gl; jj++)
597             {
598                 m.dd_bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]=dd_bspline(p.k,i,m.r_temp[(ii-2)*p.n_gl+jj
                        -1], p , m);
```

```
599                 }
600             }
601         }
602 }
603
604 // previously get the matrix value of the bspline
605 void gen_bspline_matrix_plot(struct para p, struct matrix m)
606 {
607     double bspline();
608     int i, i_p;
609     for (i=2; i<=(p.n-1); i++)
610     {
611         for(i_p=2;i_p<=(p.n-1);i_p++)
612         {
613             m.bspline_matrix_plot[i-2][i_p-2]=bspline( p.k , i_p , m.r[i-2], p , m);
614         }
615     }
616 }
617
618
619
620 //  Gauss-Legendre intergral evaluation of overlap intergal S
621 double s_integral(int i, int i_p, int ll, int ll_p, int mm, int mm_p, struct para p, struct matrix m)
622 {
623     //  declaration
624
625     double output;
626
627     // use the property that b_i*b_j=0 when |i-j|>=k && <Ylm||Yl'm'>=0 when ll!=ll_p
628
629     if (abs(i-i_p)>=p.k || ll!=ll_p || mm!=mm_p)
630     {
631         output=0;
632     }
633     else
634     {
635         //  index_min=min(i,j)  index_max=max(i,j)
636         int index_min;
637         int index_max;
638         if (i<i_p)
639         {
640             index_min=i;
641             index_max=i_p;
642         }
643         else
644         {
```

```
645             index_min=i_p;
646             index_max=i;
647         }
648
649         // define the summation variablr to sum the integration of different interval
650         double sum_integral=0;
651         int ii,jj;
652         for (ii=index_max; ii<=(index_min+p.k-1); ii++)
653         {
654             //  the integral on a interval t[ii]--t[ii+1]
655             double sum_interval=0;
656             for (jj=1; jj<=p.n_gl; jj++)
657             {
658                 // the transformation between x_gl to r
659                 sum_interval=sum_interval+m.w_gl[jj]*m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]*m.
                        bspline_matrix[i_p-2][(ii-2)*p.n_gl+jj-1];
660             }
661             sum_interval=sum_interval*(m.t[ii+1]-m.t[ii]+0.0)/2.0;
662             sum_integral=sum_integral+sum_interval;
663         }
664         output=sum_integral;
665     }
666     return output;
667 }
668
669
670 //  Gauss-Legendre intergral evaluation of kinetic term for electron B[] D^2 B[]
671 double kinetic_integral(int i, int i_p, int ll, int ll_p, int mm, int mm_p, struct para p, struct matrix
        m)
672 {
673     //  declaration
674
675     double output;
676
677     // use the property that b_i*b_j\d^2(b_j)/dr^2=0 when |i-j|>=k && <Ylm||Yl'm'>=0 when ll!=ll_p
678     if (abs(i-i_p)>=p.k || ll!=ll_p || mm!=mm_p)
679     {
680         output=0.0;
681     }
682     else
683     {
684         //  index_min=min(i,j)  index_max=max(i,j)
685         int index_min;
686         int index_max;
687         if (i<i_p)
688         {
```

```
689             index_min=i;
690             index_max=i_p;
691         }
692         else
693         {
694             index_min=i_p;
695             index_max=i;
696         }
697
698         // define the summation variable to sum the integration of different interval
699         double sum_integral=0.0;
700         int ii,jj;
701         for (ii=index_max; ii<=(index_min+p.k-1); ii++)
702         {
703             //  the integral on a interval t[ii]--t[ii+1]
704             double sum_interval=0.0;
705             for (jj=1; jj<=p.n_gl; jj++)
706             {
707                 sum_interval=sum_interval+m.w_gl[jj]*m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]*(m.
                        dd_bspline_matrix[i_p-2][(ii-2)*p.n_gl+jj-1]-(ll_p*(ll_p+1.0)*m.bspline_matrix[i_p
                        -2][(ii-2)*p.n_gl+jj-1]+0.0)/(pow( (m.r_temp[(ii-2)*p.n_gl+jj-1]+p.epsilon) ,2)+0.0)
                        );
708             }
709             sum_interval=sum_interval*(m.t[ii+1]-m.t[ii]+0.0)/2.0;
710             sum_integral=sum_integral+sum_interval;
711         }
712         sum_integral=(-0.5)*sum_integral;
713         output=sum_integral;
714     }
715     return output;
716 }
717
718
719 //  Gauss-Legendre intergral evaluation of static potential term Z/r term
720 double static_potential_integral(int i, int i_p, int ll, int ll_p, int mm, int mm_p, struct para p,
        struct matrix m)
721 {
722     //  declaration
723     double output=0.0;
724
725     // use the property that b_i*b_j=0 when |i-j|>=k && <Ylm||Yl'm>=0 when ll!=ll_p
726     if (abs(i-i_p)>=p.k || ll!=ll_p || mm!=mm_p)
727     {
728         output=0.0;
729     }
730     else
```

```
731     {
732         //  index_min=min(i,j)  index_max=max(i,j)
733         int index_min;
734         int index_max;
735         if (i<i_p)
736         {
737             index_min=i;
738             index_max=i_p;
739         }
740         else
741         {
742             index_min=i_p;
743             index_max=i;
744         }
745 
746         // define the summation variable to sum the integration of different interval
747         double sum_integral=0.0;
748         int ii,jj;
749         for (ii=index_max; ii<=(index_min+p.k-1); ii++)
750         {
751             //  the integral on a interval t[ii]--t[ii+1]
752             double sum_interval=0.0;
753             for (jj=1; jj<=p.n_gl; jj++)
754             {
755                 sum_interval=sum_interval+m.w_gl[jj]*m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]*m.
                        bspline_matrix[i_p-2][(ii-2)*p.n_gl+jj-1]*(-p.charge)/(m.r_temp[(ii-2)*p.n_gl+jj-1]+
                        p.epsilon+0.0);
756             }
757             sum_interval=sum_interval*(m.t[ii+1]-m.t[ii]+0.0)/2.0;
758             sum_integral=sum_integral+sum_interval;
759         }
760         output=sum_integral;
761     }
762     return output;
763 }
764 
765 
766 ///  Gauss-Legendre intergral evaluation of laser potential term without the amplitude E, a,b,c is the
       intergral on x,y,z direction
767 //void laser_potential_integral(int i, int i_p, int ll, int ll_p, int mm, int mm_p, double *a, double *b,
       double *c)
768 //{
769 //    //  declaration
770 //    double bspline(int kk, int i, double x);
771 //    double Y_x(int ll, int ll_p, int mm, int mm_p);
772 //    double Y_y(int ll, int ll_p, int mm, int mm_p);
```

```
773 //     double Y_z(int ll, int ll_p, int mm, int mm_p);
774 //
775 //     // use the property that b_i*b_j=0 when |i-j|>=k && <Ylm|sin(theta),cos(theta)|Yl'm'>=0 when |ll-
    ll_p|!=1
776 //     if (abs(i-i_p)<k && abs(ll_p-ll)==1)
777 //     {
778 //         // use the property <Ylm|cos(phi),sin(phi)|Yl'm'>=0 when |mm-mm_p|!=1, <Ylm||Yl'm'>=0 when |mm-
    mm_p|!=0
779 //         if (abs(mm_p-mm)==1)
780 //         {
781 //             //  index_min=min(i,j)  index_max=max(i,j)
782 //             int index_min;
783 //             int index_max;
784 //             if (i<i_p)
785 //             {
786 //                 index_min=i;
787 //                 index_max=i_p;
788 //             }
789 //             else
790 //             {
791 //                 index_min=i_p;
792 //                 index_max=i;
793 //             }
794 //
795 //             // define the summation variable to sum the integration of different interval
796 //             double sum_integral=0.0;
797 //             int ii,jj;
798 //             for (ii=index_max; ii<=(index_min+k-1); ii++)
799 //             {
800 //                 //  the integral on a interval t[ii]--t[ii+1]
801 //                 double sum_interval=0.0;
802 //                 for (jj=1; jj<=n_gl; jj++)
803 //                 {
804 //                     double r_temp=((t[ii+1]-t[ii])*x_gl[jj]+(t[ii+1]+t[ii])+0.0)/2.0;
805 //                     sum_interval=sum_interval+w_gl[jj]*bspline(k, i, r_temp )*bspline(k, i_p, r_temp)*
    r_temp;
806 //                 }
807 //                 sum_interval=sum_interval*(t[ii+1]-t[ii]+0.0)/2.0;
808 //                 sum_integral=sum_integral+sum_interval;
809 //             }
810 //             *a=sum_integral*Y_x(ll,ll_p,mm,mm_p);
811 //             *b=sum_integral*Y_y(ll,ll_p,mm,mm_p);
812 //             *c=0.0;
813 //         }
814 //         else if(mm==mm_p)
815 //         {
```

```
816 //              *a=0.0;
817 //              *b=0.0;
818 //              int index_min;
819 //              int index_max;
820 //              if (i<i_p)
821 //              {
822 //                  index_min=i;
823 //                  index_max=i_p;
824 //              }
825 //              else
826 //              {
827 //                  index_min=i_p;
828 //                  index_max=i;
829 //              }
830 //
831 //              // define the summation variable to sum the integration of different interval
832 //              double sum_integral=0.0;
833 //              int ii,jj;
834 //              for (ii=index_max; ii<=(index_min+k-1); ii++)
835 //              {
836 //                  //  the integral on a interval t[ii]--t[ii+1]
837 //                  double sum_interval=0.0;
838 //                  for (jj=1; jj<=n_gl; jj++)
839 //                  {
840 //                      double r_temp=((t[ii+1]-t[ii])*x_gl[jj]+(t[ii+1]+t[ii])+0.0)/2.0;
841 //                      sum_interval=sum_interval+w_gl[jj]*bspline(k, i, r_temp )*bspline(k, i_p, r_temp)*
    r_temp;
842 //                  }
843 //                  sum_interval=sum_interval*(t[ii+1]-t[ii]+0.0)/2.0;
844 //                  sum_integral=sum_integral+sum_interval;
845 //              }
846 //              *c=sum_integral*Y_z(ll,ll_p,mm,mm_p);
847 //          }
848 //          else
849 //          {
850 //              *a=0;
851 //              *b=0;
852 //              *c=0;
853 //          }
854 //      }
855 //      else
856 //      {
857 //          *a=0;
858 //          *b=0;
859 //          *c=0;
860 //      }
```

```
861 //}
862
863 //  Gauss-Legendre intergral evaluation of laser potential term without the amplitude E
864 void laser_potential_integral(int i, int i_p, int ll, int ll_p, int mm , int mm_p , double *d , struct
        para p , struct matrix m )
865 {
866     //  declaration
867     double Y_x();
868     double Y_y();
869     double Y_z();
870
871     // use the property that b_i*b_j=0 when |i-j|>=k && <Ylm|cos(theta)|Yl'm'>=0 when |ll-ll_p|!=1
872     if (abs(i-i_p)<p.k && abs(ll_p-ll)==1)
873     {
874         // use the property <Ylm|cos(phi),sin(phi)|Yl'm'>=0 when |mm-mm_p|!=1, <Ylm||Yl'm'>=0 when |mm-
              mm_p|!=0
875         if (abs(mm_p-mm)==1)
876         {
877             //printf("abs(ll_p-ll)==1,abs(mm_p-mm)==1\n");
878             //  index_min=min(i,j)  index_max=max(i,j)
879             int index_min;
880             int index_max;
881             if (i<i_p)
882             {
883                 index_min=i;
884                 index_max=i_p;
885             }
886             else
887             {
888                 index_min=i_p;
889                 index_max=i;
890             }
891
892             // define the summation variable to sum the integration of different interval
893             double sum_integral=0.0;
894             int ii,jj;
895             for (ii=index_max; ii<=(index_min+p.k-1); ii++)
896             {
897                 //  the integral on a interval t[ii]--t[ii+1]
898                 double sum_interval=0.0;
899                 for (jj=1; jj<=p.n_gl; jj++)
900                 {
901                     sum_interval=sum_interval+m.w_gl[jj]*m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]*m.
                          bspline_matrix[i_p-2][(ii-2)*p.n_gl+jj-1]*m.r_temp[(ii-2)*p.n_gl+jj-1];
902                 }
903                 sum_interval=sum_interval*(m.t[ii+1]-m.t[ii]+0.0)/2.0;
```

```
                sum_integral=sum_integral+sum_interval;
            }
            d[0]=sum_integral*Y_x(ll,ll_p,mm,mm_p);
            d[1]=sum_integral*Y_y(ll,ll_p,mm,mm_p);
            d[2]=0.0;
        }
        else if(mm==mm_p)
        {
            //printf("abs(ll_p-ll)==1,abs(mm_p-mm)==0\n");
            d[0]=0.0;
            d[1]=0.0;
            int index_min;
            int index_max;
            if (i<i_p)
            {
                index_min=i;
                index_max=i_p;
            }
            else
            {
                index_min=i_p;
                index_max=i;
            }

            // define the summation variable to sum the integration of different interval
            double sum_integral=0.0;
            int ii,jj;
            for (ii=index_max; ii<=(index_min+p.k-1); ii++)
            {
                //  the integral on a interval t[ii]--t[ii+1]
                double sum_interval=0.0;
                for (jj=1; jj<=p.n_gl; jj++)
                {
                    sum_interval=sum_interval+m.w_gl[jj]*m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]*m.
                        bspline_matrix[i_p-2][(ii-2)*p.n_gl+jj-1]*m.r_temp[(ii-2)*p.n_gl+jj-1];
                }
                sum_interval=sum_interval*(m.t[ii+1]-m.t[ii]+0.0)/2.0;
                sum_integral=sum_integral+sum_interval;
            }
            d[2]=sum_integral*Y_z(ll,ll_p,mm,mm_p);
        }
        else
        {
            d[0]=0;
            d[1]=0;
            d[2]=0;
```

```
949
950             }
951         }
952         else
953         {
954             d[0]=0;
955             d[1]=0;
956             d[2]=0;
957         }
958 }
959
960
961 // the integral for the angular direction for the z part of the laser
962 double Y_x(int ll, int ll_p, int mm, int mm_p)
963 {
964     double output;
965     if (mm==mm_p+1)
966     {
967         if (ll==ll_p+1)
968         {
969             output=-sqrt(1.0/2.0)*sqrt((ll+mm-1.0)*(ll+mm)/(2.0*(2.0*ll-1.0)*(2.0*ll+1.0)));
970         }
971         else if(ll==ll_p-1)
972         {
973             output=-sqrt(1.0/2.0)*-sqrt((ll-mm+1.0)*(ll-mm+2.0)*(ll+1.0)/((2.0*ll+1)*(2.0*ll+2.0)*(2.0*ll
                    +3.0)));
974         }
975         else
976         {
977             output=0.0;
978         }
979
980     }
981     else if(mm==mm_p-1)
982     {
983         if (ll==ll_p+1)
984         {
985             output=sqrt(1.0/2.0)*sqrt((ll-mm-1.0)*(ll-mm)/(2.0*(2.0*ll-1)*(2.0*ll+1)));
986         }
987         else if(ll==ll_p-1)
988         {
989             output=sqrt(1.0/2.0)*-sqrt((ll+mm+1.0)*(ll+mm+2.0)*(ll+1.0)/((2.0*ll+1)*(2.0*ll+2.0)*(2.0*ll
                    +3.0)));
990         }
991         else
992         {
```

```
            output=0.0;
        }
    }
    else
    {
        output=0.0;
    }
    return output;
}

// the integral for the angular direction for the y part of the laser
double Y_y(int ll, int ll_p, int mm, int mm_p)
{
    double output;
    if (mm==mm_p+1)
    {
        if (ll==ll_p+1)
        {
            output=sqrt(1.0/2.0)*sqrt((ll+mm-1.0)*(ll+mm)/(2.0*(2.0*ll-1.0)*(2.0*ll+1.0)));
        }
        else if(ll==ll_p-1)
        {
            output=sqrt(1.0/2.0)*-sqrt((ll-mm+1.0)*(ll-mm+2.0)*(ll+1.0)/((2.0*ll+1)*(2.0*ll+2.0)*(2.0*ll
                +3.0)));
        }
        else
        {
            output=0.0;
        }

    }
    else if(mm==mm_p-1)
    {
        if (ll==ll_p+1)
        {
            output=sqrt(1.0/2.0)*sqrt((ll-mm-1.0)*(ll-mm)/(2.0*(2.0*ll-1)*(2.0*ll+1)));
        }
        else if(ll==ll_p-1)
        {
            output=sqrt(1.0/2.0)*-sqrt((ll+mm+1.0)*(ll+mm+2.0)*(ll+1.0)/((2.0*ll+1)*(2.0*ll+2.0)*(2.0*ll
                +3.0)));
        }
        else
        {
            output=0.0;
        }
```

```
1037        }
1038        else
1039        {
1040            output=0.0;
1041        }
1042        return output;
1043    }
1044
1045    // the integral for the angular direction for the z part of the laser
1046    double Y_z(int ll, int ll_p, int mm, int mm_p)
1047    {
1048        double output;
1049        if (mm==mm_p)
1050        {
1051            if (ll_p==ll+1)
1052            {
1053                output=sqrt((pow(ll+1.0,2)-pow(mm,2))/((2*ll+3.0)*(2*ll+1.0)));
1054            }
1055            else if(ll_p==ll-1)
1056            {
1057                output=sqrt((pow(ll_p+1.0,2)-pow(mm,2))/((2*ll_p+3.0)*(2*ll_p+1.0)));
1058            }
1059            else
1060            {
1061                output=0.0;
1062            }
1063
1064        }
1065        else
1066        {
1067            output=0.0;
1068        }
1069        return output;
1070    }
1071
1072
1073    //  generate the S matrix   the matrix has already in the form of exlude b1 and bn, also in the lapack
        form, i.e. s[0][0] has meaning
1074    void gen_s_matrix(double **s_matrix , struct para p , struct matrix m)
1075    {
1076        //  declaration
1077        double s_integral();
1078
1079        int i,ll,mm,i_p,ll_p,mm_p;
1080        for (i=2; i<=(p.n-1); i++)
1081        {
```

```
        for (ll=0; ll<=p.l_max; ll++)
        {
            for (mm=-ll; mm<=ll; mm++)
            {
                for (i_p=2; i_p<=(p.n-1); i_p++)
                {
                    for (ll_p=0; ll_p<=p.l_max; ll_p++)
                    {
                        for (mm_p=-ll_p; mm_p<=ll_p; mm_p++)
                        {
                            s_matrix[(ll*(ll+1)+mm)*(p.n-2)+i-2][(ll_p*(ll_p+1)+mm_p)*(p.n-2)+i_p-2]=
                                s_integral(i, i_p, ll, ll_p, mm, mm_p, p, m);
                        }
                    }
                }
            }
        }
    }
}

//  generate the H matrix, H matrix is the hamilton of the atom, the matrix has already in the form of
    exlude b1 and bn, also in the lapack form, i.e. H[0][0] has meaning [attention: the nuclear part is
    not included in order to compare with F. Martin]
void gen_h_matrix(double **h_matrix, struct para p, struct matrix m)
{
    //  declaration
    double kinetic_integral();
    double static_potential_integral();

    int i,ll,mm,i_p,ll_p,mm_p;
    for (i=2; i<=(p.n-1); i++)
    {
        for (ll=0; ll<=p.l_max; ll++)
        {
            for (mm=-ll; mm<=ll; mm++)
            {
                for (i_p=2; i_p<=(p.n-1); i_p++)
                {
                    for (ll_p=0; ll_p<=p.l_max; ll_p++)
                    {
                        for (mm_p=-ll_p; mm_p<=ll_p; mm_p++)
                        {
                            h_matrix[(ll*(ll+1)+mm)*(p.n-2)+i-2][(ll_p*(ll_p+1)+mm_p)*(p.n-2)+i_p-2]=
                                kinetic_integral(i, i_p, ll, ll_p, mm, mm_p, p, m)+
                                static_potential_integral(i, i_p, ll, ll_p,mm,mm_p, p, m);
                        }
```

```
                    }
                }
            }
        }
    }
}


//  generate the D_x D_y D_z matrix here Dy is in real value so we need to add i before it when calcualte A, B  the matrix has already in the form of exlude bl and bn, also in the lapack form, the amplitude is the parameter varied with time so it is not included here
void gen_d_matrix(double **dx_matrix, double **dy_matrix, double **dz_matrix, struct para p , struct matrix m )
{
    //  declaration
    void laser_potential_integral();

    int i,ll,mm,i_p,ll_p,mm_p;
    for (i=2; i<=(p.n-1); i++)
    {
        for (ll=0; ll<=p.l_max; ll++)
        {
            for (mm=-ll; mm<=ll; mm++)
            {
                for (i_p=2; i_p<=(p.n-1); i_p++)
                {
                    for (ll_p=0; ll_p<=p.l_max; ll_p++)
                    {
                        for (mm_p=-ll_p; mm_p<=ll_p; mm_p++)
                        {
                            // the pointer for the intergral value
                            double *d=(double *)malloc(3*sizeof(double));
                            laser_potential_integral(i, i_p, ll, ll_p, mm, mm_p,  d , p , m );
                            dx_matrix[(ll*(ll+1)+mm)*(p.n-2)+i-2][(ll_p*(ll_p+1)+mm_p)*(p.n-2)+i_p-2]=d[0];
                            dy_matrix[(ll*(ll+1)+mm)*(p.n-2)+i-2][(ll_p*(ll_p+1)+mm_p)*(p.n-2)+i_p-2]=d[1];
                            dz_matrix[(ll*(ll+1)+mm)*(p.n-2)+i-2][(ll_p*(ll_p+1)+mm_p)*(p.n-2)+i_p-2]=d[2];
                        }
                    }
                }
            }
        }
    }
}
```

```
1163
1164
1165
1166 //  generate the A matrix A=S+i*(H+E_x(t)*D_x+E_y(t)*I*D_y+E_z(t)*D_z)*delta_t/2
1167 // generate the B matrix B=S-i*(H+E_x(t)*D_x+E_y(t)*I*D_y+E_z(t)*D_z)*delta_t/2
1168 void gen_a_b_matrix(double complex **a_matrix,double complex **b_matrix, double **dx_matrix, double **
         dy_matrix, double **dz_matrix, double **h_matrix, double **s_matrix,double E_x_temp, double E_y_temp
         , double E_z_temp ,struct para p, struct matrix m )
1169 {
1170     int i,ll,mm,i_p,ll_p,mm_p;
1171     for (i=2; i<=(p.n-1); i++)
1172     {
1173         for (ll=0; ll<=p.l_max; ll++)
1174         {
1175             for (mm=-ll; mm<=ll; mm++)
1176             {
1177                 for (i_p=2; i_p<=(p.n-1); i_p++)
1178                 {
1179                     for (ll_p=0; ll_p<=p.l_max; ll_p++)
1180                     {
1181                         for (mm_p=-ll_p; mm_p<=ll_p; mm_p++)
1182                         {
1183                             int index_temp=(ll*(ll+1)+mm)*(p.n-2)+i-2;
1184                             int index_temp_p=(ll_p*(ll_p+1)+mm_p)*(p.n-2)+i_p-2;
1185                             a_matrix[index_temp][index_temp_p]=s_matrix[index_temp][index_temp_p]+I*(
                                     h_matrix[index_temp][index_temp_p]+E_x_temp*dx_matrix[index_temp][
                                     index_temp_p]+I*E_y_temp*dy_matrix[index_temp][index_temp_p]+E_z_temp*
                                     dz_matrix[index_temp][index_temp_p])*p.delta_t/2.0;
1186                             b_matrix[index_temp][index_temp_p]=s_matrix[index_temp][index_temp_p]-I*(
                                     h_matrix[index_temp][index_temp_p]+E_x_temp*dx_matrix[index_temp][
                                     index_temp_p]+I*E_y_temp*dy_matrix[index_temp][index_temp_p]+E_z_temp*
                                     dz_matrix[index_temp][index_temp_p])*p.delta_t/2.0;
1187                         }
1188                     }
1189                 }
1190             }
1191         }
1192     }
1193 }
1194
1195
1196
1197 //  generate the b vector b=B*c(t)
1198 void gen_b_vector(double complex *b_vector, double complex **b_matrix, double complex *c,struct para p,
         struct matrix m)
1199 {
```

```
1200      int i,j;
1201      for (i=0; i<p.n_lapack; i++)
1202      {
1203          // define the sum varible
1204          double complex sum=0.0;
1205          for (j=0; j<p.n_lapack; j++)
1206          {
1207              sum=sum+b_matrix[i][j]*c[j];
1208          }
1209          b_vector[i]=sum;
1210      }
1211
1212  }
1213
1214
1215  // unificate the c for the initial state
1216  void unification( double complex *c, double complex *c_new, struct para p, struct matrix m)
1217  {
1218      //  calculate the factor of the eigenvector sum(ll) sum(ii,ii_p)c_ii,ll*c_ii_p,ll int b_i*b_j
1219      // sum varible for different ll and mm
1220      double sum=0.0;
1221      int i,i_p;
1222      for (i=2; i<=p.n-1; i++)
1223      {
1224          for (i_p=2; i_p<=p.n-1; i_p++)
1225          {
1226              // sum varible for the mutiply of radial and angular direction
1227              double sum_index=0.0;
1228              if (abs(i-i_p)<p.k)// use the property that b_i*b_j=0 when |i-j|>=k
1229              {
1230                  int index_min;
1231                  int index_max;
1232                  if (i<i_p)
1233                  {
1234                      index_min=i;
1235                      index_max=i_p;
1236                  }
1237                  else
1238                  {
1239                      index_min=i_p;
1240                      index_max=i;
1241                  }
1242
1243                  // sum varible of the radial grid
1244                  double sum_integral=0.0;
1245                  int ii,jj;
```

```
1246                for (ii=index_max; ii<=(index_min+p.k-1); ii++)
1247                {
1248                    //  the integral on a interval t[ii]--t[ii+1]
1249                    double sum_interval=0.0;
1250                    for (jj=1; jj<=p.n_gl; jj++)
1251                    {
1252                        sum_interval=sum_interval+m.w_gl[jj]*m.bspline_matrix[i-2][(ii-2)*p.n_gl+jj-1]*m.
                                bspline_matrix[i_p-2][(ii-2)*p.n_gl+jj-1];
1253                    }
1254                    sum_interval=sum_interval*(m.t[ii+1]-m.t[ii]+0.0)/2.0;
1255                    sum_integral=sum_integral+sum_interval;
1256                }
1257                //sum varible of the angular graid for c*c
1258                double sum_angular=0.0;
1259                int ll,mm;
1260                for (ll=0; ll<=p.l_max; ll++)
1261                {
1262                    for(mm=-ll;mm<=ll;mm++)
1263                    {
1264                        sum_angular=sum_angular+c[(ll*(ll+1)+mm)*(p.n-2)+i-2]*c[(ll*(ll+1)+mm)*(p.n-2)+
                                i_p-2];
1265                    }
1266                }
1267                sum_index=sum_angular*sum_integral;
1268            }
1269            else
1270            {
1271                sum_index=0.0;
1272            }
1273            sum=sum+sum_index;
1274        }
1275    }
1276
1277    //f=sqrt(1/sum)
1278    double complex f;
1279    f=sqrt(1.0/(sum+0.0));
1280
1281    //mupltiply the factor to the initial states
1282    int ll,mm;
1283    for (i=2; i<=p.n-1; i++)
1284    {
1285        for (ll=0; ll<=p.l_max; ll++)
1286        {
1287            for(mm=-ll;mm<=ll;mm++)
1288            {
1289                c_new[(ll*(ll+1)+mm)*(p.n-2)+i-2]=f*c[(ll*(ll+1)+mm)*(p.n-2)+i-2];
```

```
1290                }
1291            }
1292        }
1293    }
1294
1295
1296    // generate the wave function for plotting without the angular part
1297    void gen_w(double complex *w, double complex *c, struct para p, struct matrix m)
1298    {
1299        int ll,mm,i,i_p;
1300        for(ll=0; ll<=p.l_max; ll++)
1301        {
1302            for(mm=-ll;mm<=ll;mm++)
1303            {
1304                for (i=2; i<=p.n-1; i++)
1305                {
1306                    // the sum for the contribution of different b spline on one point
1307                    double complex sum_index=0;
1308                    for (i_p=2; i_p<=p.n-1; i_p++)
1309                    {
1310                        if (abs(i_p-i)<p.k)// use the property that b_j=0 when |i-j|>=k
1311                        {
1312                            sum_index=sum_index+c[(ll*(ll+1)+mm)*(p.n-2)+i_p-2]*m.bspline_matrix_plot[i-2][
                                    i_p-2];
1313                        }
1314
1315                    }
1316                    // the radial part is  sum c*b/r
1317                    w[(ll*(ll+1)+mm)*(p.n-2)+i-2]=sum_index/(m.r[(i-2)]+p.epsilon);
1318                }
1319            }
1320        }
1321    }
1322
1323    int main()
1324    {
1325
1326        //for indexes
1327        int i,j,ii,jj;
1328
1329        // initialize parameters
1330        struct para p;
1331        p.l=100;
1332        p.k=3;
1333        p.n=p.l+p.k-1;
1334        p.n_knot=p.l+2*p.k-1;
```

```
1335    p.n_gl=p.k;
1336    p.l_max=4;
1337    p.n_lapack=(p.l_max+1)*(p.l_max+1)*(p.n-2);
1338    p.n_load=p.n-2;
1339    p.seq_load=0;
1340    p.l_load=0;
1341    p.m_load=0;
1342    p.charge=1.0;
1343    p.r_min=0.0;
1344    p.r_max=100.0;
1345    p.dist="equidstant";
1346    //p.dist="expoential";
1347    p.epsilon=1e-15;
1348    p.delta_t=0.3;
1349    p.tm_min=0.0;
1350    p.omega=0.8;
1351    p.T=2.0*M_PI/(p.omega+0.0);
1352    p.tm_max=2*p.T;
1353    p.tm_size=(p.tm_max-p.tm_min)/p.delta_t;
1354    p.E_x=0.0168;
1355    p.E_y=0.0168;
1356    p.E_z=0;//0.0168;
1357    p.mask_prop=0.2;
1358    p.out_tm=1;
1359
1360    //  declaraiton
1361    // initialize matrixes
1362    struct matrix m;
1363    m.x_gl=(double *)malloc((p.n_gl+1)*sizeof(double));
1364    m.w_gl=(double *)malloc((p.n_gl+1)*sizeof(double));
1365    m.t=(double *)malloc((p.n_knot+1)*sizeof(double));
1366    m.tm=(double *)malloc((p.tm_size+1)*sizeof(double));
1367    m.r_temp=(double *)malloc((p.n-2)*p.n_gl*sizeof(double));
1368    m.r=(double *)malloc((p.n-2)*sizeof(double));
1369    m.bspline_matrix=(double **)malloc((p.n-2)*sizeof(double));
1370    for (i=0; i<(p.n-2); i++)
1371    {
1372        m.bspline_matrix[i]=(double *)malloc((p.n-2)*p.n_gl*sizeof(double));
1373    }
1374
1375    m.dd_bspline_matrix=(double **)malloc((p.n-2)*sizeof(double));
1376    for (i=0; i<(p.n-2); i++)
1377    {
1378        m.dd_bspline_matrix[i]=(double *)malloc((p.n-2)*p.n_gl*sizeof(double));
1379    }
1380
```

```
1381    m.bspline_matrix_plot=(double **)malloc((p.n-2)*sizeof(double));
1382    for (i=0; i<(p.n-2); i++)
1383    {
1384        m.bspline_matrix_plot[i]=(double *)malloc((p.n-2)*sizeof(double));
1385    }
1386
1387    m.mask_vector=(double *)malloc((p.n-2)*sizeof(double));
1388
1389    //  declaraiton
1390    void gen_x_w_gl();
1391    void gen_t();
1392    void gen_r_temp();
1393    void gen_r();
1394    void gen_tm();
1395    void gen_laser();
1396    void gen_bspline_matrix();
1397    void gen_dd_bspline_matrix();
1398    void gen_bspline_matrix_plot();
1399    void gen_s_matrix();
1400    void gen_h_matrix();
1401    void gen_d_matrix();
1402    void gen_a_b_matrix();
1403    void gen_b_vector();
1404    void unification();
1405    void gen_w();
1406    void gen_mask_vector();
1407    void mask();
1408
1409
1410    /******* print banner ************/
1411
1412    printf("hydrogen atom : linear polarized\n");
1413    printf("(C) Copyright by Lin Xin (2016)\n");
1414    printf("------------------------------------------------------------\n");
1415
1416    /******* output the parameters into log file *********/
1417    FILE *file_log=fopen("re_log.txt","w");
1418    fprintf(file_log,"b_spline_order=%d\n",p.k);
1419    fprintf(file_log,"grid distribution=%s\n",p.dist);
1420    fprintf(file_log,"r_min=%f\n",p.r_min);
1421    fprintf(file_log,"r_max=%f\n",p.r_max);
1422    fprintf(file_log,"delta_r=%f\n",(p.r_max-p.r_min)/(p.n_knot-2*p.k+1));
1423    fprintf(file_log,"radius_grid=%d\n",p.n_load);
1424    fprintf(file_log,"charge=%f\n",p.charge);
1425    fprintf(file_log,"seq of initial state=%d\n",p.seq_load);
1426    fprintf(file_log,"l of initial state=%d\n",p.l_load);
```

```
1427    fprintf(file_log,"m of initial state=%d\n",p.m_load);
1428    fprintf(file_log,"l max consider in calculation=%d\n",p.l_max);
1429    fprintf(file_log,"tm_min=%f\n",p.tm_min);
1430    fprintf(file_log,"tm_max=%f\n",p.tm_max);
1431    fprintf(file_log,"delta_t=%f\n",p.delta_t);
1432    fprintf(file_log,"tm_size=%d\n",p.tm_size);
1433    fprintf(file_log,"omega=%f\n",p.omega);
1434    fprintf(file_log,"T=%f\n",p.T);
1435    fprintf(file_log,"E_x=%f\n",p.E_x);
1436    fprintf(file_log,"E_y=%f\n",p.E_y);
1437    fprintf(file_log,"E_z=%f\n",p.E_z);
1438    fprintf(file_log,"mask_prop=%f\n",p.mask_prop);
1439    fprintf(file_log,"out_tm=%d\n",p.out_tm);
1440    fclose(file_log);
1441
1442    printf("b_spline_order=%d\n",p.k);
1443    printf("grid distribution=%s\n",p.dist);
1444    printf("r_min=%f\n",p.r_min);
1445    printf("r_max=%f\n",p.r_max);
1446    printf("delta_r=%f\n",(p.r_max-p.r_min)/(p.n_knot-2*p.k+1));
1447    printf("radius_grid=%d\n",p.n_load);
1448    printf("charge=%f\n",p.charge);
1449    printf("seq of initial state=%d\n",p.seq_load);
1450    printf("l of initial state=%d\n",p.l_load);
1451    printf("m of initial state=%d\n",p.m_load);
1452    printf("l max consider in calculation=%d\n",p.l_max);
1453    printf("tm_min=%f\n",p.tm_min);
1454    printf("tm_max=%f\n",p.tm_max);
1455    printf("delta_t=%f\n",p.delta_t);
1456    printf("tm_size=%d\n",p.tm_size);
1457    printf("omega=%f\n",p.omega);
1458    printf("T=%f\n",p.T);
1459    printf("E_x=%f\n",p.E_x);
1460    printf("E_y=%f\n",p.E_y);
1461    printf("E_z=%f\n",p.E_z);
1462    printf("mask_prop=%f\n",p.mask_prop);
1463    printf("out_tm=%d\n",p.out_tm);
1464
1465
1466    //  clock timing
1467    time_t start,end;
1468    start =time(NULL);
1469
1470
1471
1472
```

```
1473    /***** generate the sequences********/
1474    gen_x_w_gl( p , m );
1475    gen_t( p , m );
1476    gen_r_temp( p , m );
1477    gen_r( p , m);
1478    gen_tm( p , m);
1479    gen_bspline_matrix( p , m );
1480    gen_dd_bspline_matrix( p , m );
1481    gen_bspline_matrix_plot( p , m );
1482    gen_mask_vector( p , m );
1483
1484    // output the radial grid
1485    FILE *file_radius=fopen("r.txt","w");
1486    for ( i=0; i<(p.n-2); i++)
1487    {
1488        fprintf(file_radius,"%.15f\n",m.r[i]);
1489    }
1490    fclose(file_radius);
1491
1492
1493    /****** circular polarized laser ****/
1494    double *E_x_vec;
1495    E_x_vec=(double *)malloc((p.tm_size+1)*sizeof(double));
1496
1497    double *E_y_vec;
1498    E_y_vec=(double *)malloc((p.tm_size+1)*sizeof(double));
1499
1500    double *E_z_vec;
1501    E_z_vec=(double *)malloc((p.tm_size+1)*sizeof(double));
1502
1503    gen_laser( E_x_vec , E_y_vec , E_z_vec , p , m );
1504
1505    FILE *file_laser=fopen("laser.txt","w");
1506    for (i=1; i<=p.tm_size; i++)
1507    {
1508        fprintf(file_laser,"%.15f %.15f %.15f\n",E_x_vec[i],E_y_vec[i],E_z_vec[i]);
1509    }
1510
1511
1512    /********* load the eigenvector *********/
1513    FILE *file_eigen_vec=fopen("v.txt","r");
1514
1515    double **vr_load;
1516    vr_load=(double **)malloc(p.n_load*sizeof(double));
1517    for (i=0; i<p.n_load; i++)
1518    {
```

```
1519            vr_load[i]=(double *)malloc(p.n_load*sizeof(double));
1520        }
1521
1522        for (i=0;i<p.n_load;i++)
1523        {
1524            for(j=0;j<p.n_load;j++)
1525            {
1526                fscanf(file_eigen_vec,"%lf", &vr_load[i][j]);
1527            }
1528        }
1529
1530        fclose(file_eigen_vec);
1531
1532
1533
1534        /******** load sequence *******/
1535        FILE *file_seq=fopen("s.txt","r");
1536        int *seq;
1537        seq=(int *)malloc(p.n_load*sizeof(int));
1538
1539        for (i=0;i<p.n_load;i++)
1540        {
1541            fscanf(file_seq,"%d", &seq[i]);
1542        }
1543
1544        fclose(file_seq);
1545
1546
1547
1548        //initial the c
1549        double complex *c_init;
1550        c_init=(double complex *)malloc(p.n_lapack*sizeof(double complex));
1551
1552        for (i=2; i<=p.n-1; i++)
1553        {
1554            c_init[(p.l_load*(p.l_load+1)+p.m_load)*(p.n-2)+i-2]=vr_load[i-2][seq[p.seq_load]];
1555        }
1556
1557        free(vr_load);
1558        free(seq);
1559
1560
1561        //  unify c
1562        unification( c_init , c_init , p , m );
1563
1564
```

```
1565
1566        //allocate s,h,d_matrix
1567        double **s_matrix=(double **)malloc(p.n_lapack*sizeof(double));
1568        for (i=0; i<p.n_lapack; i++)
1569        {
1570            s_matrix[i]=(double *)malloc(p.n_lapack*sizeof(double));
1571        }
1572
1573        double **h_matrix=(double **)malloc(p.n_lapack*sizeof(double));
1574        for (i=0; i<p.n_lapack; i++)
1575        {
1576            h_matrix[i]=(double *)malloc(p.n_lapack*sizeof(double));
1577        }
1578
1579        double **dx_matrix;
1580        dx_matrix=(double **)malloc(p.n_lapack*sizeof(double));
1581        for (i=0; i<p.n_lapack; i++)
1582        {
1583            dx_matrix[i]=(double *)malloc(p.n_lapack*sizeof(double));
1584        }
1585
1586        double **dy_matrix;
1587        dy_matrix=(double **)malloc(p.n_lapack*sizeof(double ));
1588        for (i=0; i<p.n_lapack; i++)
1589        {
1590            dy_matrix[i]=(double *)malloc(p.n_lapack*sizeof(double));
1591        }
1592
1593        double **dz_matrix;
1594        dz_matrix=(double **)malloc(p.n_lapack*sizeof(double ));
1595        for (i=0; i<p.n_lapack; i++)
1596        {
1597            dz_matrix[i]=(double *)malloc(p.n_lapack*sizeof(double));
1598        }
1599
1600
1601        // generate the s,h,d_matrix
1602        gen_s_matrix( s_matrix , p , m );
1603        gen_h_matrix( h_matrix , p , m );
1604        gen_d_matrix( dx_matrix , dy_matrix , dz_matrix , p , m );
1605
1606        /********
1607        FILE *file_h_matrix=fopen("h_matrix.txt","w");
1608        for (i=0; i<p.n_lapack; i++)
1609        {
1610            for (j=0; j<p.n_lapack; j++)
```

```
1611            {
1612                fprintf(file_h_matrix,"%.15f ",h_matrix[i][j]);
1613            }
1614            fprintf(file_h_matrix,"\n");
1615        }
1616        fclose(file_h_matrix);
1617
1618        FILE *file_s_matrix=fopen("s_matrix.txt","w");
1619        for (i=0; i<p.n_lapack; i++)
1620        {
1621            for (j=0; j<p.n_lapack; j++)
1622            {
1623                fprintf(file_s_matrix,"%.15f ",s_matrix[i][j]);
1624            }
1625            fprintf(file_s_matrix,"\n");
1626        }
1627        fclose(file_s_matrix);
1628
1629        FILE *file_dx_matrix=fopen("dx_matrix.txt","w");
1630        for (i=0; i<p.n_lapack; i++)
1631        {
1632            for (j=0; j<p.n_lapack; j++)
1633            {
1634                fprintf(file_dx_matrix,"%.15f ",dx_matrix[i][j]);
1635            }
1636            fprintf(file_dx_matrix,"\n");
1637        }
1638        fclose(file_dx_matrix);
1639
1640        FILE *file_dy_matrix=fopen("dy_matrix.txt","w");
1641        for (i=0; i<p.n_lapack; i++)
1642        {
1643            for (j=0; j<p.n_lapack; j++)
1644            {
1645                fprintf(file_dy_matrix,"%.15f ",dy_matrix[i][j]);
1646//                printf("%.3f ",dy_matrix[i][j]);
1647            }
1648            fprintf(file_dy_matrix,"\n");
1649//            printf("\n");
1650        }
1651        fclose(file_dy_matrix);
1652
1653        FILE *file_dz_matrix=fopen("dz_matrix.txt","w");
1654        for (i=0; i<p.n_lapack; i++)
1655        {
1656            for (j=0; j<p.n_lapack; j++)
```

```
1657        {
1658            fprintf(file_dz_matrix,"%.15f ",dz_matrix[i][j]);
1659        }
1660        fprintf(file_dz_matrix,"\n");
1661    }
1662    fclose(file_dz_matrix);
1663
1664    return 0;
1665
1666     ****/
1667
1668    // allocate the a,b matrix, b vector
1669    double complex **a_matrix;
1670    a_matrix=(double complex **)malloc(p.n_lapack*sizeof(double complex));
1671    for (i=0; i<p.n_lapack; i++)
1672    {
1673        a_matrix[i]=(double complex *)malloc(p.n_lapack*sizeof(double complex));
1674    }
1675
1676    double complex **b_matrix;
1677    b_matrix=(double complex **)malloc(p.n_lapack*sizeof(double complex));
1678    for (i=0; i<p.n_lapack; i++)
1679    {
1680        b_matrix[i]=(double complex *)malloc(p.n_lapack*sizeof(double complex));
1681    }
1682
1683    double complex *b_vector;
1684    b_vector=(double complex *)malloc(p.n_lapack*sizeof(double complex));
1685
1686
1687    /***
1688    // a_matrix_double is the real form change of a_matrix
1689    double **a_matrix_double;
1690    a_matrix_double=(double **)malloc((2*n_lapack+1)*sizeof(double));
1691    for (i=0; i<=2*n_lapack; i++)
1692    {
1693        a_matrix_double[i]=(double *)malloc((2*n_lapack+1)*sizeof(double));
1694    }
1695
1696    double *b_vector_double;
1697    b_vector_double=(double *)malloc((2*n_lapack+1)*sizeof(double));
1698
1699
1700    //the c propragate with time
1701    double complex *c_prop;
1702    c_prop=(double complex *)malloc(n_lapack*sizeof(double complex));
```

```
1703      for (i=0; i<n_lapack; i++)
1704      {
1705          c_prop[i]=c_init[i];
1706      }
1707
1708      double complex *wf_prop;
1709      wf_prop=(double complex *)malloc(n_lapack*sizeof(double complex));
1710
1711      // the change form for solving complex linear equations
1712
1713
1714      double complex *r_a_matrix_complex;
1715      r_a_matrix_complex=(double complex *)malloc(n_lapack*n_lapack*sizeof(double complex));
1716
1717      double complex *r_b_vector_complex;
1718      r_b_vector_complex=(double complex *)malloc(n_lapack*sizeof(double complex));
1719
1720
1721
1722
1723  //
1724  //
1725  //
1726        gen_a_b_matrix(a_matrix, b_matrix, dx_matrix, dy_matrix, dz_matrix, h_matrix,s_matrix,0.0,0.0,0.0);
1727
1728  //    FILE *file_b_matrix_real=fopen("b_matrix_real.txt","w");
1729  //    for (int i=0; i<n_lapack; i++)
1730  //    {
1731  //        for (int j=0; j<n_lapack; j++)
1732  //        {
1733  //            fprintf(file_b_matrix_real,"%.15f ",creal(b_matrix[i][j]));
1734  //        }
1735  //        fprintf(file_b_matrix_real,"\n");
1736  //    }
1737  //    fclose(file_b_matrix_real);
1738  //
1739  //    FILE *file_b_matrix_imag=fopen("b_matrix_imag.txt","w");
1740  //    for (int i=0; i<n_lapack; i++)
1741  //    {
1742  //        for (int j=0; j<n_lapack; j++)
1743  //        {
1744  //            fprintf(file_b_matrix_imag,"%.15f ",cimag(b_matrix[i][j]));
1745  //        }
1746  //        fprintf(file_b_matrix_imag,"\n");
1747  //    }
1748  //    fclose(file_b_matrix_imag);
```

```
1749
1750
1751 //     FILE *file_a_matrix_real=fopen("a_matrix_real.txt","w");
1752 //     for (int i=0; i<n_lapack; i++)
1753 //     {
1754 //         for (int j=0; j<n_lapack; j++)
1755 //         {
1756 //             fprintf(file_a_matrix_real,"%.15f ",creal(a_matrix[i][j]));
1757 //         }
1758 //         fprintf(file_a_matrix_real,"\n");
1759 //     }
1760 //     fclose(file_a_matrix_real);
1761 //
1762 //     FILE *file_a_matrix_imag=fopen("a_matrix_imag.txt","w");
1763 //     for (int i=0; i<n_lapack; i++)
1764 //     {
1765 //         for (int j=0; j<n_lapack; j++)
1766 //         {
1767 //             fprintf(file_a_matrix_imag,"%.15f ",cimag(a_matrix[i][j]));
1768 //         }
1769 //         fprintf(file_a_matrix_imag,"\n");
1770 //     }
1771 //     fclose(file_a_matrix_imag);
1772
1773     double *mask_vector;
1774     mask_vector=(double *)malloc((n-2)*sizeof(double));
1775     gen_mask_vector(mask_vector,r);
1776
1777     FILE *file_mask_vec=fopen("mask_vector.txt","w");
1778     for (i=0; i<(n-2); i++)
1779     {
1780         fprintf(file_mask_vec,"%.15f\n ",mask_vector[i]);
1781     }
1782     fclose(file_mask_vec);
1783
1784     gen_b_vector(b_vector,b_matrix,c_prop);
1785      ****/
1786
1787     //allocate and initial the c propragate with time
1788     double complex *c_prop;
1789     c_prop=(double complex *)malloc(p.n_lapack*sizeof(double complex));
1790     for (i=0; i<p.n_lapack; i++)
1791     {
1792         c_prop[i]=c_init[i];
1793     }
1794
```

```
1795      free(c_init);
1796
1797      //allocate the wave function propragate with time
1798      double complex *wf_prop;
1799      wf_prop=(double complex *)malloc(p.n_lapack*sizeof(double complex));
1800
1801      FILE *file_wf_prop=fopen("wf_prop.txt","w");
1802
1803
1804      gen_w( wf_prop , c_prop , p , m );
1805
1806      for (ii=0; ii<p.n_lapack; ii++)
1807      {
1808          fprintf(file_wf_prop,"%.15f\t %.15f\n ",creal(wf_prop[ii]),cimag(wf_prop[ii]));
1809      }
1810
1811
1812
1813      /********** start the time propragation *********/
1814
1815  //      int ii,jj;
1816  //
1817  //      for (ii=0; ii<n_lapack; ii++)
1818  //      {
1819  //          fprintf(file_c_prop,"%.15f\t %.15f\n ",creal(c_prop[ii]),cimag(c_prop[ii]));
1820  //      }
1821  //
1822  //      gen_w(wf_prop,r,c_prop);
1823  //
1824  //      for (ii=0; ii<n_lapack; ii++)
1825  //      {
1826  //          fprintf(file_wf_prop,"%.15f\t %.15f\n ",creal(wf_prop[ii]),cimag(wf_prop[ii]));
1827  //      }
1828  //
1829  //      for (ii=0; ii<n_lapack; ii++)
1830  //      {
1831  //          fprintf(file_b_vector,"%.15f\t %.15f\n ",creal(b_vector[ii]),cimag(b_vector[ii]));
1832  //      }
1833  ////      return 0;
1834  //
1835  //      int *indx=(int *)malloc((2*n_lapack+1)*sizeof(int));;
1836  //      double d;
1837
1838
1839      for (i=1; i<=p.tm_size; i++)
1840      {
```

```
1841
1842 //          FILE *file_a_matrix_real=fopen("a_matrix_real.txt","w");
1843 //          for (int ii=0; ii<n_lapack; ii++)
1844 //          {
1845 //              for (int jj=0; jj<n_lapack; jj++)
1846 //              {
1847 //                  fprintf(file_a_matrix_real,"%.15f ",creal(a_matrix[ii][jj]));
1848 //              }
1849 //              fprintf(file_a_matrix_real,"\n");
1850 //          }
1851 //          fclose(file_a_matrix_real);
1852 //
1853 //          FILE *file_a_matrix_imag=fopen("a_matrix_imag.txt","w");
1854 //          for (int ii=0; ii<n_lapack; ii++)
1855 //          {
1856 //              for (int jj=0; jj<n_lapack; jj++)
1857 //              {
1858 //                  fprintf(file_a_matrix_imag,"%.15f ",cimag(a_matrix[ii][jj]));
1859 //              }
1860 //              fprintf(file_a_matrix_imag,"\n");
1861 //          }
1862 //          fclose(file_a_matrix_imag);
1863 //
1864 //          FILE *file_b_matrix_real=fopen("b_matrix_real.txt","w");
1865 //          for (int ii=0; ii<n_lapack; ii++)
1866 //          {
1867 //              for (int jj=0; jj<n_lapack; jj++)
1868 //              {
1869 //                  fprintf(file_b_matrix_real,"%.15f ",creal(b_matrix[ii][jj]));
1870 //              }
1871 //              fprintf(file_b_matrix_real,"\n");
1872 //          }
1873 //          fclose(file_b_matrix_real);
1874 //
1875 //          FILE *file_b_matrix_imag=fopen("b_matrix_imag.txt","w");
1876 //          for (int ii=0; ii<n_lapack; ii++)
1877 //          {
1878 //              for (int jj=0; jj<n_lapack; jj++)
1879 //              {
1880 //                  fprintf(file_b_matrix_imag,"%.15f ",cimag(b_matrix[ii][jj]));
1881 //              }
1882 //              fprintf(file_b_matrix_imag,"\n");
1883 //          }
1884 //          fclose(file_b_matrix_imag);
1885
1886
```

```
1887        gen_a_b_matrix(a_matrix, b_matrix, dx_matrix, dy_matrix, dz_matrix, h_matrix,s_matrix,E_x_vec[i],
                E_y_vec[i],E_z_vec[i] , p , m );
1888
1889        gen_b_vector( b_vector, b_matrix , c_prop , p , m);
1890
1891
1892        /****
1893        //change the a_matrix, b_vector into the real form
1894        for (ii=0; ii<n_lapack; ii++)
1895        {
1896            for (jj=0; jj<n_lapack; jj++)
1897            {
1898                a_matrix_double[ii+1][jj+1]=creal(a_matrix[ii][jj]);
1899                a_matrix_double[ii+1][jj+1+n_lapack]=-cimag(a_matrix[ii][jj]);
1900                a_matrix_double[ii+1+n_lapack][jj+1]=cimag(a_matrix[ii][jj]);
1901                a_matrix_double[ii+1+n_lapack][jj+1+n_lapack]=creal(a_matrix[ii][jj]);
1902            }
1903        }
1904        for (ii=0; ii<n_lapack; ii++)
1905        {
1906            b_vector_double[ii+1]=creal(b_vector[ii]);
1907            b_vector_double[ii+1+n_lapack]=cimag(b_vector[ii]);
1908        }
1909        //  solve the complex equations A(t)c(t+dt)=b(t)
1910
1911        //gaussj(a_matrix_double, 2*n_lapack, b_vector_double, 1);
1912
1913        ludcmp(a_matrix_double,2*n_lapack,indx,&d);
1914        lubksb(a_matrix_double,2*n_lapack,indx,b_vector_double);
1915
1916
1917        // the c(t+dt) is in r_b_vector_complex
1918        for (ii=0; ii<n_lapack; ii++)
1919        {
1920            c_prop[ii]=b_vector_double[ii+1]+I*b_vector_double[ii+1+n_lapack];
1921        }
1922
1923        for (ii=0; ii<n_lapack; ii++)
1924        {
1925            fprintf(file_c_prop,"%.15f %.15f\n",creal(c_prop[ii]),cimag(c_prop[ii]));
1926        }
1927         ****/
1928
1929        //  reduce the H and S matrix to the LAPACK one dim form
1930        double complex *r_a_matrix;
1931        r_a_matrix = (double complex *)malloc( p.n_lapack * p.n_lapack * sizeof(double complex));
```

```
1932
1933        for (ii=0; ii < p.n_lapack; ii++)
1934        {
1935            for (jj=0; jj < p.n_lapack; jj++)
1936            {
1937                r_a_matrix[ii*p.n_lapack+jj]=a_matrix[ii][jj];
1938            }
1939        }
1940
1941        double complex *r_b_vector;
1942        r_b_vector = (double complex *)malloc( p.n_lapack* 1 * sizeof(double complex));
1943
1944        for (ii=0; ii<p.n_lapack; ii++)
1945        {
1946            r_b_vector[ii]=b_vector[ii];
1947        }
1948
1949        int nrhs=1;
1950        int lda=p.n_lapack;
1951        int ldb=1;
1952        int ipiv[p.n_lapack];
1953
1954        //  solve SX=H
1955        //        lapack_int LAPACKE_zgesv( int matrix_layout, lapack_int n, lapack_int nrhs,
1956        //                                  lapack_complex_double* a, lapack_int lda,
1957        //                                  lapack_int* ipiv, lapack_complex_double* b,
1958        //                                  lapack_int ldb )
1959        int info = LAPACKE_zgesv(LAPACK_ROW_MAJOR, p.n_lapack, nrhs, r_a_matrix, lda, ipiv, r_b_vector,
                ldb);
1960
1961        for ( ii=0; ii<p.n_lapack; ii++)
1962        {
1963            c_prop[ii]=r_b_vector[ii];
1964        }
1965
1966        //mask after every time step
1967        mask( c_prop , c_prop , p , m );
1968
1969
1970        if (i%p.out_tm==0)
1971        {
1972            printf("conduct the %d time step\n",i);
1973            gen_w(wf_prop, c_prop , p , m );
1974
1975            for (ii=0; ii<p.n_lapack; ii++)
1976            {
```

```
1977                    fprintf(file_wf_prop,"%.15f %.15f\n",creal(wf_prop[ii]),cimag(wf_prop[ii]));
1978                }
1979            }
1980
1981        }
1982
1983
1984        fclose(file_wf_prop);
1985
1986    //      fclose(file_b_vector);
1987    //      fclose(file_c_prop);
1988
1989        /*** end of lapack ***/
1990
1991        printf("b_spline_order=%d\n",p.k);
1992        printf("grid distribution=%s\n",p.dist);
1993        printf("r_min=%f\n",p.r_min);
1994        printf("r_max=%f\n",p.r_max);
1995        printf("delta_r=%f\n",(p.r_max-p.r_min)/(p.n_knot-2*p.k+1));
1996        printf("radius_grid=%d\n",p.n_load);
1997        printf("charge=%f\n",p.charge);
1998        printf("seq of initial state=%d\n",p.seq_load);
1999        printf("l of initial state=%d\n",p.l_load);
2000        printf("m of initial state=%d\n",p.m_load);
2001        printf("l max consider in calculation=%d\n",p.l_max);
2002        printf("tm_min=%f\n",p.tm_min);
2003        printf("tm_max=%f\n",p.tm_max);
2004        printf("delta_t=%f\n",p.delta_t);
2005        printf("tm_size=%d\n",p.tm_size);
2006        printf("omega=%f\n",p.omega);
2007        printf("T=%f\n",p.T);
2008        printf("E_x=%f\n",p.E_x);
2009        printf("E_y=%f\n",p.E_y);
2010        printf("E_z=%f\n",p.E_z);
2011        printf("mask_prop=%f\n",p.mask_prop);
2012        printf("out_tm=%d\n",p.out_tm);
2013
2014        free(m.x_gl);
2015        free(m.w_gl);
2016        free(m.t);
2017        free(m.tm);
2018        free(m.r_temp);
2019        free(m.r);
2020        free(m.bspline_matrix);
2021        free(m.dd_bspline_matrix);
2022        free(m.bspline_matrix_plot);
```

```
2023        free(m.mask_vector);
2024        free(E_x_vec);
2025        free(E_y_vec);
2026        free(E_z_vec);
2027        free(s_matrix);
2028        free(h_matrix);
2029        free(dx_matrix);
2030        free(dy_matrix);
2031        free(dz_matrix);
2032        free(a_matrix);
2033        free(b_matrix);
2034        free(b_vector);
2035        //      free(a_matrix_double);
2036        //      free(b_vector_double);
2037        free(c_prop);
2038        free(wf_prop);
2039
2040        //  print time
2041        end =time(NULL);
2042        printf("time=%f\n",difftime(end,start)/60.0);
2043
2044
2045        return 0;
2046    }
```

References

[1] Muller H. An efficient propagation scheme for the time-dependent Schrödinger equation in the velocity gauge[J]. LASER PHYSICS-LAWRENCE-, 1999, 9:138–148.

[2] Kosloff R, Tal-Ezer H. A direct relaxation method for calculating eigenfunctions and eigenvalues of the schrödinger equation on a grid[J]. Chemical Physics Letters, 1986, 127(3):223 – 230. http://www.sciencedirect.com/science/article/pii/0009261486802627.

[3] Feit M, Fleck J, Steiger A. Solution of the Schrödinger equation by a spectral method[J]. Journal of Computational Physics, 1982, 47(3):412 – 433. http://www.sciencedirect.com/science/article/pii/0021999182900912.

[4] Goldberg A, Schey H M, Schwartz J L. Computer-Generated Motion Pictures of One-Dimensional Quantum-Mechanical Transmission and Reflection Phenomena[J]. American Journal of Physics, 1967, 35(3).

[5] Vanne Y V. Ionization of molecular hydrogen in ultrashort intense laser pulses[D].[S.l.]: Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät I, 2010.

[6] Zewail A H. Femtochemistry: Atomic-scale dynamics of the chemical bond using ultrafast lasers (Nobel Lecture)[J]. Angewandte Chemie International Edition, 2000, 39(15):2586–2631.

[7] Press W H, Teukolsky S A, Vetterling W T, et al. Numerical Recipes (Cambridge[M].[S.l.]: Cambridge Univ. Press, 1992.

[8] Herath T, Yan L, Lee S K, et al. Strong-Field Ionization Rate Depends on the Sign of the Magnetic Quantum Number[J]. Phys. Rev. Lett., 2012, 109:043004. http://link.aps.org/doi/10.1103/PhysRevLett.109.043004.

[9] Barth I, Smirnova O. Nonadiabatic tunneling in circularly polarized laser fields: Physical picture and calculations[J]. Phys. Rev. A, 2011, 84:063415. http://link.aps.org/doi/10.1103/PhysRevA.84.063415.

[10] Keldysh L. Ionization in the field of a strong electromagnetic wave[J]. Zh. Eksperim. i Teor. Fiz., 1964, 47.

[11] Bauer J H, Mota-Furtado F, O'Mahony P F, et al. Ionization and excitation of the excited hydrogen atom in strong circularly polarized laser fields[J]. Phys. Rev. A, 2014, 90:063402. http://link.aps.org/doi/10.1103/PhysRevA.90.063402.

[12] Born M, Oppenheimer R. Zur Quantentheorie der Molekeln[J]. Annalen der Physik, 1927, 389(20):457–484. http://dx.doi.org/10.1002/andp.19273892002.

[13] Rudenko A, Zrost K, Ergler T, et al. Coulomb singularity in the transverse momentum distribution for strong-field single ionization[J]. Journal of Physics B: Atomic, Molecular and Optical Physics, 2005, 38(11):L191. http://stacks.iop.org/0953-4075/38/i=11/a=L01.

[14] Zuo T, Bandrauk A D. Charge-resonance-enhanced ionization of diatomic molecular ions by intense lasers[J]. Phys. Rev. A, 1995, 52:R2511–R2514. http://link.aps.org/doi/10.1103/PhysRevA.52.R2511.

[15] Takemoto N, Becker A. Visualization and interpretation of attosecond electron dynamics in laser-driven hydrogen molecular ion using Bohmian trajectories[J]. The Journal of Chemical Physics, 2011, 134(7). http://scitation.aip.org/content/aip/journal/jcp/134/7/10.1063/1.3553178.

[16] Xin L, Qin H C, Wu W Y, et al. Fraunhofer-like diffracted lateral photoelectron momentum distributions of H_2^+ in charge-resonance-enhanced ionization in strong laser fields[J]. Phys. Rev. A, 2015, 92:063803. http://link.aps.org/doi/10.1103/PhysRevA.92.063803.

[17] Schoenberg I J. Contributions to the problem of approximation of equidistant data by analytic functions, Part B: On the problem of osculatory interpolation, a second class of analytic approximation formulae[J]. Quart. Appl. Math, 1946, 4(2):112–141.

[18] Martín F. Ionization and dissociation using B-splines: photoionization of the hydrogen molecule[J]. Journal of Physics B: Atomic, Molecular and Optical Physics, 1999, 32(16):R197. http://stacks.iop.org/0953-4075/32/i=16/a=201.

[19] Bachau H, Cormier E, Decleva P, et al. Applications of B-splines in atomic and molecular physics[J]. Reports on Progress in Physics, 2001, 64(12):1815.

[20] Wigner E P, Fano U. Group theory and its application to the quantum mechanics of atomic spectra[J]. American Journal of Physics, 1960, 28(4):408–409.

[21] Brosolo M, Decleva P, Lisini A. Accurate variational determination of continuum wavefunctions by a one-centre expansion in a spline basis. An application to H + 2 and HeH 2+ photoionization[J]. Journal of Physics B: Atomic, Molecular and Optical Physics, 1992, 25(15):3345. http://stacks.iop.org/0953-4075/25/i=15/a=015.

[22] Bauer D, Koval P. Qprop: A Schrödinger-solver for intense laser–atom interaction[J]. Computer Physics Communications, 2006, 174(5):396 – 421. http://www.sciencedirect.com/science/article/pii/S0010465505005825.