Implementing Density Functional Theory

Fadjar Fathurrahman
Hermawan Kresno Dipojono

June 15, 2020

Preface

Importance of density functional theory

Implementation of density functional theory in various program packages (free and commercial)

Several books about density functional theories

The problem: not yet giving necessary details

This book is our humble attempt to demystifying several aspects of practical density functional theory to beginners in the field.

Objective of this book: show the reader how to implement a density functional theory for simple system containing only model potential (such as harmonic potential) and to non-local pseudopotentials which are usually used in typical DFT calculations for molecular and crystalline systems.

Outline of the book: 1d, 2d, 3d, Schrodinger equation, Poisson equation, Kohn-Sham equation for local (pseudo)potentials, Kohn-Sham equation for nonlocal pseudopotentials.

This is for *acknowledgments*.

Bandung,
month year

Fadjar Fathurrahman
Hermawan Kresno Dipojono

Contents

Chapter 1

An introduction to density functional theory

Density functional theory: [1, 2],

Using the Kohn-Sham density functional theory, total energy of system of interacting electrons under external potential $V_{\text{ext}}(\mathbf{r})$ can be written as a functional of a set of single-particle wave functions or Kohn-Sham orbitals $\{\psi_i(\mathbf{r})\}$

$$E\left[\{\psi_i(\mathbf{r})\}\right] = -\frac{1}{2}\int \psi_i(\mathbf{r})\nabla^2\psi_i(\mathbf{r})\,\mathrm{d}\mathbf{r} + \int \rho(\mathbf{r})V_{\text{ext}}(\mathbf{r})\,\mathrm{d}\mathbf{r} + \frac{1}{2}\int \frac{\rho(\mathbf{r})\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\,\mathrm{d}\mathbf{r}\,\mathrm{d}\mathbf{r}' + E_{\text{xc}}\left[\rho(\mathbf{r})\right] \tag{1.1}$$

where single-particle electron density $\rho(\mathbf{r})$ is calculated as

$$\rho(\mathbf{r}) = \sum_i f_i \psi^*(\mathbf{r})\psi(\mathbf{r}) \tag{1.2}$$

In Equation 1.2 the summation is done over all occupied single electronic states i and f_i is the occupation number of the i-th orbital. For doubly-occupied orbitals we have $f_i = 2$. In the usual setting in material science and chemistry, the external potential is usually the potential due to the atomic nuclei (ions):

$$V_{\text{ext}}(\mathbf{r}) = \sum_I \frac{Z_I}{|\mathbf{r}-\mathbf{R}_I|} \tag{1.3}$$

and an additional term of energy, the nucleus-nucleus energy E_{nn}, is added to the total energy functional (1.1):

$$E_{\text{nn}} = \frac{1}{2}\sum_I \sum_J \frac{Z_I Z_J}{|R_I - R_J|} \tag{1.4}$$

The energy terms in total energy functionals are the kinetic, external, Hartree, and exchange-correlation (XC) energy, respectively. The functional form of the last term (i.e. the XC energy) in terms of electron density is not known and one must resort to an approximation. In this article we will use the local density approximation for the XC energy. Under this approximation, the XC energy can be written as:

$$E_{\text{xc}}\left[\rho(\mathbf{r})\right] = \int \rho(\mathbf{r})\epsilon_{\text{xc}}\left[\rho(\mathbf{r})\right]\,\mathrm{d}\mathbf{r} \tag{1.5}$$

where ϵ_{xc} is the exchange-correlation energy per particle. There are many functional forms that have been devised for ϵ_{xc} and they are usually named by the persons who proposed them. An explicit form of ϵ_{xc} will be given later.

Many material properties can be derived from the minimum of the functional (1.1). This minimum energy is also called the *ground state energy*. This energy can be obtained by using direct minimization of the Kohn-Sham

energy functional or by solving the Kohn-Sham equations:

$$\hat{H}_{\mathrm{KS}}\,\psi_i(\mathbf{r}) = \epsilon_i\,\psi_i(\mathbf{r}) \tag{1.6}$$

where ϵ_i are the Kohn-Sham orbital energies and the Kohn-Sham Hamiltonian $\hat{H}_{\mathrm{KS}}$ is defined as

$$\hat{H}_{\mathrm{KS}} = -\frac{1}{2}\nabla^2 + V_{\mathrm{ext}}(\mathbf{r}) + V_{\mathrm{Ha}}(\mathbf{r}) + V_{\mathrm{xc}}(\mathbf{r}) \tag{1.7}$$

From the solutions of the Kohn-Sham equations: ϵ_i and $\psi_i(\mathbf{r})$ we can calculate the corresponding minimum total energy from the functional (1.1).

In the definition of Kohn-Sham Hamiltonian, other than the external potential which is usually specified from the problem, we have two additional potential terms, namely the Hartree and exchange-correlation potential. The Hartree potential can be calculated from its integral form

$$V_{\mathrm{Ha}}(\mathbf{r}) = \int \frac{\rho(\mathbf{r})}{\mathbf{r}-\mathbf{r}'}\,\mathrm{d}\mathbf{r}' \tag{1.8}$$

An alternative way to calculate the Hartree potential is to solve the Poisson equation:

$$\nabla^2 V_{\mathrm{Ha}}(\mathbf{r}) = -4\pi\rho(\mathbf{r}) \tag{1.9}$$

The exchange-correlation potential is defined as functional derivative of the exchange-correlation energy:

$$V_{\mathrm{xc}}(\mathbf{r}) = \frac{\delta E[\rho(\mathbf{r})]}{\delta\rho(\mathbf{r})} \tag{1.10}$$

The Kohn-Sham equations are nonlinear eigenvalue equations in the sense that to calculate the solutions $\{\epsilon_i\}$ and $\{\psi_i(\mathbf{r})\}$ we need to know the electron density $\rho(\mathbf{r})$ to build the Hamiltonian. The electron density itself must be calculated from $\{\psi_i(\mathbf{r})\}$ which are not known (the quantities we want to solve for). The usual algorithm to solve the Kohn-Sham equations is the following:

- (**STEP 1**): start from a guess input density $\rho^{\mathrm{in}}(\mathbf{r})$
- (**STEP 2**): calculate the Hamiltonian defined in (1.7)
- (**STEP 3**): solve the eigenvalue equations in (1.6) to obtain the eigenpairs $\{\epsilon_i\}$, $\{\psi_i(\mathbf{r})\}$
- (**STEP 4**): calculate the output electron density $\rho^{\mathrm{out}}(\mathbf{r})$ from $\{\psi_i(\mathbf{r})\}$ that are obtained from the previous step.
- (**STEP 5**) Check whether the difference between $\rho^{\mathrm{in}}(\mathbf{r})$ and $\rho^{\mathrm{out}}(\mathbf{r})$ is small. If the difference is still above a chosen threshold then back to **STEP 1**. If the difference is already small the stop the algorithm.

The algorithm we have just described is known as the self-consistent field (SCF) algorithm.

Nowadays, there are many available software packages that can be used to solve the Kohn-Sham equations such as Quantum ESPRESSO, ABINIT, VASP, Gaussian, and NWChem are among the popular ones. A more complete list is given in a Wikipedia page [3]. These packages differs in several aspects, particularly the basis set used to discretize the Kohn-Sham equations and characteristic of the systems they can handle (periodic or non-periodic systems). These packages provide an easy way for researchers to carry out calculations based on density functional theory for particular systems they are interested in without knowing the details of how these calculations are performed.

In this book we will describe a simple way to solve the Kohn-Sham equations based on finite difference approximation. Our focus will be on the practical numerical implementation of a solver for the Kohn-Sham equations. The solver will be implemented using Julia programming language [4]. Our target is to calculate the ground state energy of several simple model systems. We will begin to build our solver starting from the ground up. The roadmap of the article is as follows.

- We begin from discussing numerical solution of Schroedinger equation in 1d. We will introduce finite difference approximation and its use in approximating second derivative operator that is present in the Schroedinger equation. We show how one can build the Hamiltonian matrix and solve the resulting eigenvalue equations using standard function that is available in Julia.
- In the next section, we discuss numerical solution of Schroedinger in 2d. We will introduce how one can handle 2d grid and applying finite difference approximation to the Laplacian operator present in the 2d Schroedinger equation. We also present several iterative methods to solve the eigenvalue equations.
- The next section discusses the numerical solution of Schroedinger equation in 3d. The methods presented in this section is a straightforward extension from the 2d case. In this section we start considering V_{ext} that is originated from the Coulomb interaction between atomic nuclei and electrons. We also introduce the concept of pseudopotential which is useful in practical calculations.
- The next section discusses about Poisson equation. Conjugate gradient method for solving system of linear equations. Hartree energy calculation.
- Hartree approximation, implementation of SCF algorithm
- Kohn-Sham equations, XC energy and potential, hydrogen molecule

Chapter 2

Schroedinger equation in 1d

In this chapter we will be concentrating on the problem of finding bound states solution to time-independent Schroedinger equation in one dimension:

$$\left[-\frac{1}{2}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right]\psi(x) = E\,\psi(x) \tag{2.1}$$

with the boundary conditions:

$$\lim_{x\to\pm\infty}\psi(x) = 0 \tag{2.2}$$

This boundary condition is relevant for non-periodic systems such as isolated or free atoms and molecules.

2.1 Grid points

We need to define a spatial domain $[x_{\mathrm{min}}, x_{\mathrm{max}}]$ where $x_{\mathrm{min}}, x_{\mathrm{max}}$ chosen such that the boundary condition 2.2 is approximately satisfied. The next step is to divide the spatial domain x using equally-spaced grid points which we will denote as $\{x_1, x_2, \ldots, x_N\}$ where N is total number of grid points. Various spatial quantities such as wave function $\psi(x)$ and potential $\psi(x)$ will be discretized on these grid points.

The grid points x_i, $i = 1, 2, \ldots$ are chosen to be:

$$x_i = x_{\mathrm{min}} + (i-1)h \tag{2.3}$$

where h is the spacing between the grid points:

$$h = \frac{x_{\mathrm{max}} - x_{\mathrm{min}}}{N-1} \tag{2.4}$$

The following Julia function can be used to initialize the grid points.

```
function init_FD1d_grid( x_min::Float64, x_max::Float64, N::Int64 )
  L = x_max - x_min
  h = L/(N-1) # spacing
  x = zeros(Float64,N) # the grid points
  for i = 1:N
    x[i] = x_min + (i-1)*h
  end
  return x, h
end
```

The function `init_FD1d_grid` takes three arguments:

- `x_min::Int64`: the left boundary point
- `x_max::Int64`: the right boundary point
- `N::Float64`: number of grid points

The function will return x which is an array of grid points and h which is the uniform spacing between grid points. The boundary points `x_min` and `x_max` will be included in the grid points.

As an example of the usage of the function `init_FD1d_grid`, let's sample and plot a Gaussian function

$$\psi(x) = \mathrm{e}^{-\alpha x^2} \tag{2.5}$$

where α is a positive number. We will sample the function within the domain $[x_{\mathrm{min}}, x_{\mathrm{max}}]$ where $x_{\mathrm{min}} = -5$ and $x_{\mathrm{max}} = 5$. The Gaussian function defined in (2.5) can be implemented as the following function.

```
function my_gaussian(x::Float64; α=1.0)
  return exp( -α*x^2 )
end
```

Note that we have set the default value of parameter α to 1.

The full Julia program is as follows.

```
using Printf
using LaTeXStrings

import PyPlot
const plt = PyPlot
plt.rc("text", usetex=true)

include("init_FD1d_grid.jl")

function my_gaussian(x::Float64; α=1.0)
  return exp( -α*x^2 )
end

function main()
  A = -5.0
  B =  5.0
  Npoints = 8
  x, h = init_FD1d_grid( A, B, Npoints )
  @printf("Grid spacing = %f\n", h)
  @printf("\nGrid points:\n")
  for i in 1:Npoints
    @printf("%3d %18.10f\n", i, x[i])
  end
  NptsPlot = 200
  x_dense = range(A, stop=5, length=NptsPlot)
  plt.clf()
  plt.plot(x_dense, my_gaussian.(x_dense), label=L"f(x)")
  plt.plot(x, my_gaussian.(x), label=L"Sampled $f(x)$", marker="o")
  plt.legend()
  plt.tight_layout()
  plt.savefig("IMG_gaussian_1d_8pt.pdf")
end
```

```
main()
```

After execution, the program will print grid spacing and grid points to the standard output and also plot the function to a file named `IMG_gaussian_1d_8pt.pdf` You may try to experiment by changing the value of `N` and compare the result. The resulting plots for `N=8` and `N=21` are shown in Figure XXX.

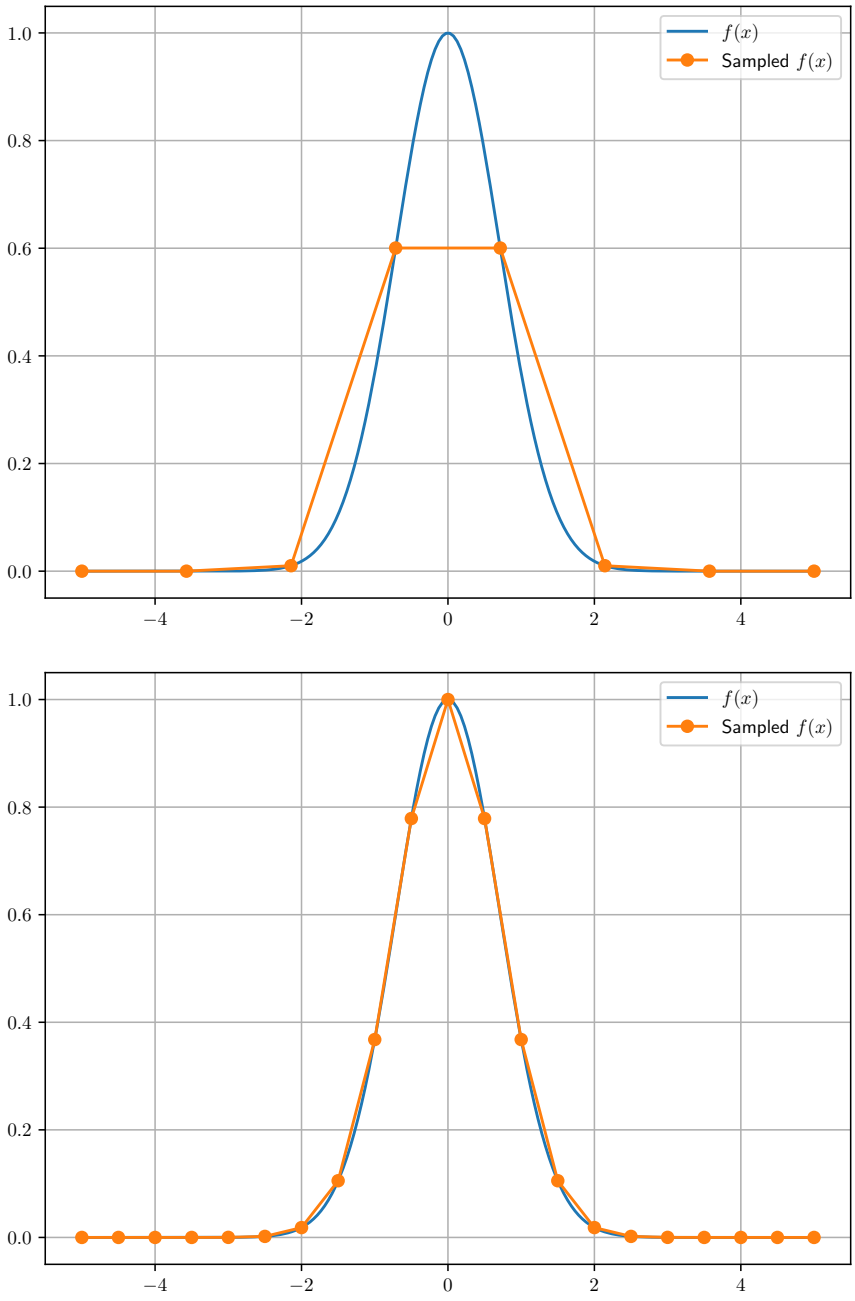

Figure 2.1: Sampling a Gaussian function defined in (2.5), $\alpha = 1$, two different number of grid points: $N = 8$ (upper) and $N = 21$ (lower). The sampled coordinates are marked by dots. The true or "continuous" function is emulated by using a dense sampling points of 200.

Note that we have used a densely-sampled points of `NptsPlot=200` in the program to emulate the true or "continuous" function. The plot of densely-sampled array is not done by not showing the point marker, as opposed to the array with lower sampling points.

Exercise Try to vary the value of α and `N`. Make the program more sophisticated by using loop over for various values of `N` instead of manually changing its value in the program. You also may want to make set the saved filename of the resulting plot programmatically.

2.2 Approximating second derivative operator

Our next task is to find an approximation to the second derivative operator present in the Equation (2.1). Suppose that we have a function sampled at appropriate positions x_i as $\psi(x_i)$. How can we approximate $\psi''(x_i)$? One simple approximation that we can use is the 3-point (central) finite difference:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi_i = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{h^2} \tag{2.6}$$

where we have the following notation have been used: $\psi_i = \psi(x_i)$. Let's see we can apply this by writing out the Equation (2.6).

$$\begin{aligned}
\psi_1'' &\approx (\psi_2 - 2\psi_1 + \psi_0)/h^2 \\
\psi_2'' &\approx (\psi_3 - 2\psi_2 + \psi_1)/h^2 \\
\psi_3'' &\approx (\psi_4 - 2\psi_3 + \psi_2)/h^2 \\
&\vdots \\
\psi_N'' &\approx (\psi_{N+1} - 2\psi_N + \psi_{N-1})/h^2
\end{aligned}$$

In the first and the last equations, there are terms involving ψ_0 and ψ_{N+1} which are not known. Recall that we have numbered our grid from 1 to N, so ψ_0 and ψ_{N+1} are outside of our grid. However, by using the boundary equation (2.2), these quantities can be taken as zeros. So we have:

$$\begin{aligned}
\psi_1'' &\approx (\psi_2 - 2\psi_1)/h^2 \\
\psi_2'' &\approx (\psi_3 - 2\psi_2 + \psi_1)/h^2 \\
\psi_3'' &\approx (\psi_4 - 2\psi_3 + \psi_2)/h^2 \\
&\vdots \\
\psi_N'' &\approx (-2\psi_N + \psi_{N-1})/h^2
\end{aligned}$$

This operation can be compactly expressed by using matrix-vector notation. By taking $\{\psi_i\}$ as a column vector, the second derivative operation can be expressed as matrix multiplication:

$$\{\psi''\} \approx \mathbb{D}^{(2)}\{\psi\} \tag{2.7}$$

where $\mathbb{D}^{(2)}$ is the second derivative matrix operator:

$$\mathbb{D}^{(2)} = \frac{1}{h^2}\begin{bmatrix}
-2 & 1 & 0 & 0 & 0 & \cdots & 0 \\
1 & -2 & 1 & 0 & 0 & \cdots & 0 \\
0 & 1 & -2 & 1 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & 1 & -2 & 1 & 0 \\
0 & \cdots & \cdots & 0 & 1 & -2 & 1 \\
0 & \cdots & \cdots & \cdots & 0 & 1 & -2
\end{bmatrix} \tag{2.8}$$

The following Julia function can be used to initialize the matrix $\mathbb{D}^{(2)}$.

```
function build_D2_matrix_3pt( N::Int64, h::Float64 )
  mat = zeros(Float64,N,N)1
  for i = 1:N-1
    mat[i,i] = -2.0
    mat[i,i+1] = 1.0
    mat[i+1,i] = mat[i,i+1]
  end
  mat[N,N] = -2.0
  return mat/h^2
end
```

The function `build_D2_matrix_3pt` takes two arguments:

- `N::Int64`: number of grid points
- `h::Float64`: the uniform grid spacing

and returns the matrix $\mathbb{D}^{(2)}$ as two dimensional array.

Before use these functions to solve Schroedinger equation, we will test the operation in Equation (2.8) for a simple function for which the second derivative can be calculated analytically. This function also should satisfy the boundary condition 2.2. We will take the the Gaussian function (2.5) that we have used before. The second derivative of this Gaussian function can be calculated as

$$\psi''(x) = \left(-2\alpha + 4\alpha^2 x^2\right) \mathrm{e}^{-\alpha x^2} \tag{2.9}$$

We also need to define the computational domain $[x_{\text{min}}, x_{\text{max}}]$ for our test. Let's choose $x_{\text{min}} = -5$ and $x_{\text{max}} = 5$ again as in the previous example. We can evaluate the value of function $\psi(x)$ at those points to be at the order of 10^{-11}, which is sufficiently small for our purpose.

The full Julia program that we will use is as follows.

```
using Printf
using LaTeXStrings

import PyPlot
const plt = PyPlot
plt.rc("text", usetex=true)

include("init_FD1d_grid.jl")
include("build_D2_matrix_3pt.jl")

function my_gaussian(x, α=1.0)
  return exp(-α*x^2)
end

function d2_my_gaussian(x, α=1.0)
  return (-2*α + 4*α^2 * x^2) * exp(-α*x^2)
end

function main(N::Int64)
  x_min = -5.0
  x_max =  5.0
  x, h = init_FD1d_grid( x_min, x_max, N )
  fx = my_gaussian.(x)
```

```
  Ndense = 200
  x_dense = range(A, stop=B, length=Ndense)
  fx_dense = my_gaussian.(x_dense)
  d2_fx_dense = d2_my_gaussian.(x_dense)

  D2 = build_D2_matrix_3pt(N, h)
  d2_fx_3pt = D2*fx

  plt.clf()
  plt.plot(x, fx, marker="o", label=L"Sampled $f(x)$")
  plt.plot(x_dense, fx_dense, label=L"f(x)")
  plt.plot(x, d2_fx_3pt, marker="o", label=L"Approx $f''(x)$")
  plt.plot(x_dense, d2_fx_dense, label=L"f''(x)")
  plt.legend()
  plt.grid()
  plt.savefig("IMG_gaussian_"*string(N)*".pdf")
end
main(15)
main(51)
```

We have followed similar approach as we have done in the previous section for plotting. The important parts of the program are the lines:

```
D2 = build_D2_matrix_3pt(N, h)
d2_fx_3pt = D2*fx
```

where we build $\mathbb{D}^{(2)}$ matrix, represented by the variable `D2` in the program, and multiply it with the vector `fx` to obtain an approximation to $\psi''(x)$. The results are plotted with two different values of number of grid points, `N=15` and `N=51`, which are shown in Figure XXX.

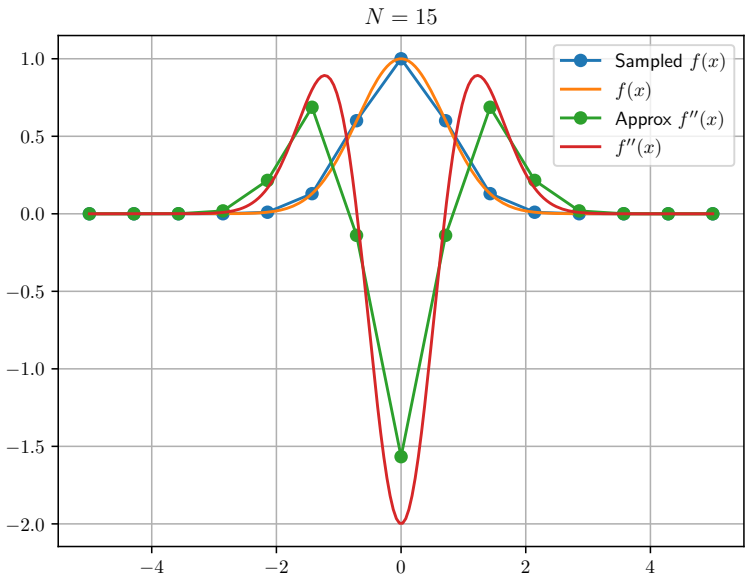

Figure 2.2: Finite difference approximation to a Gaussian function and its second derivative with number of grid points $N = 15$

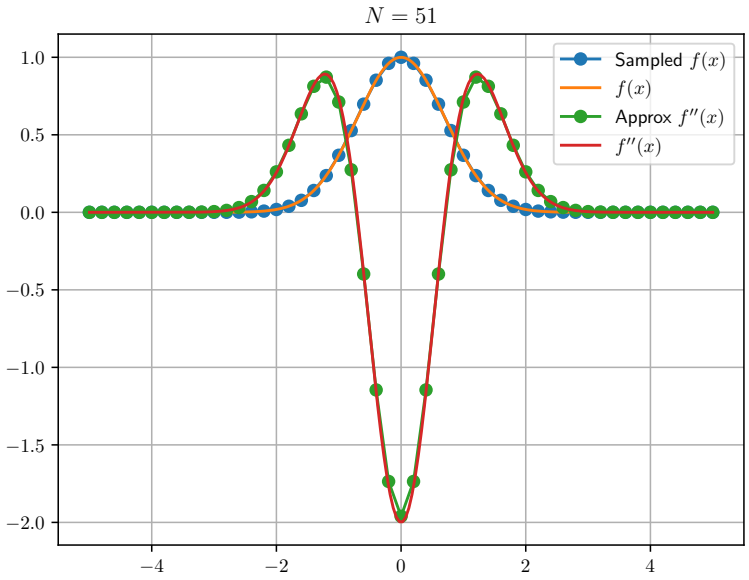

Figure 2.3: Finite difference approximation to a Gaussian function and its second derivative with number of grid points $N = 51$

2.3 Harmonic potential

The time-independent Schroedinger equation (2.1) can be expressed into the following eigenvalue problem in matrix form:

$$\mathbb{H}\{\psi\} = E\{\psi\} \tag{2.10}$$

where $\mathbb{H}$ is the Hamiltonian matrix and $\{\psi\}$ is a vector column representation of wave function. In finite difference representation, the Hamiltonian matrix is

$$\mathbb{H} = -\frac{1}{2}\mathbb{D}^{(2)} + \mathbb{V} \tag{2.11}$$

where $\mathbb{V}$ is a diagonal matrix whose elements are:

$$\mathbb{V}_{ij} = V(x_i)\delta_{ij} \tag{2.12}$$

As an example, we will start with a simple potential with known exact solution, namely the harmonic potential:

$$V(x) = \frac{1}{2}\omega^2 x^2 \tag{2.13}$$

In Julia, we can the process of building the Hamiltonian matrix is very simple

```
x, h = init_FD1d_grid(xmin, xmax, N) # grid points
D2 = build_D2_matrix_3pt(N, h)       # Build 2nd derivative matrix
Vpot = pot_harmonic.(x)              # Potential
Ham = -0.5*D2 + diagm( 0 => Vpot )   # Hamiltonian matrix
```

The function `pot_harmonic` is a function that calculate the harmonic potential

```
function pot_harmonic( x; ω=1.0 )
  return 0.5 * ω^2 * x^2
end
```

Once the Hamiltonian matrix is built, we can solve for E and $\{\psi\}$ by using standard methods. In Julia this can be achieved by simply using the `eigen` function from `LinearAlgebra` standard library as illustrated in the following code.

```
evals, evecs = eigen( Ham )
```

The values of E or eigenvalues are stored in one-dimensional array `evals` and the corresponding wave functions or eigenvectors are stored by column in two-dimensional array or matrix `evecs`.

The complete Julia program to solve the Schroedinger equation for harmonic potential is as follows.

```
using Printf
using LinearAlgebra
using LaTeXStrings

import PyPlot
const plt = PyPlot
plt.rc("text", usetex=true)

include("init_FD1d_grid.jl")
include("build_D2_matrix_3pt.jl")

function pot_harmonic( x; ω=1.0 )
  return 0.5 * ω^2 * x^2
end

function main()
  xmin = -5.0
  xmax =  5.0
  N = 51
  x, h = init_FD1d_grid(xmin, xmax, N)
  D2 = build_D2_matrix_3pt(N, h)
  Vpot = pot_harmonic.(x)
  Ham = -0.5*D2 + diagm( 0 => Vpot )
  evals, evecs = eigen( Ham )
  # We will show the 5 lowest eigenvalues
  Nstates = 5
  @printf("Eigenvalues\n")
  ω = 1.0
  hbar = 1.0
  @printf(" State         Approx              Exact          Difference\n")
  for i in 1:Nstates
    E_ana = (2*i - 1)*ω*hbar/2
    @printf("%5d %18.10f %18.10f %18.10e\n", i, evals[i], E_ana, abs(evals[i]-E_ana))
  end

  # normalize the first three eigenstates
  for i in 1:3
    ss = dot(evecs[:,i], evecs[:,i])*h
    evecs[:,i] = evecs[:,i]/sqrt(ss)
  end
```

```
  # Plot up to 3rd eigenstate
  plot_title = "N="*string(N)
  plt.plot(x, evecs[:,1], label="1st eigenstate", marker="o")
  plt.plot(x, evecs[:,2], label="2nd eigenstate", marker="o")
  plt.plot(x, evecs[:,3], label="3rd eigenstate", marker="o")
  plt.legend()
  plt.grid()
  plt.tight_layout()
  plt.savefig("IMG_main_harmonic_01_"*string(N)*".pdf")
end

main()
```

The program shows five lowest approximate eigenvalues and their comparison with analytical or exact eigenvalues. A sample output from running the program is as follows.

```
Eigenvalues
State         Approx              Exact          Difference
   1       0.4987468513       0.5000000000   1.2531486828e-03
   2       1.4937215179       1.5000000000   6.2784821079e-03
   3       2.4836386480       2.5000000000   1.6361352013e-02
   4       3.4684589732       3.5000000000   3.1541026791e-02
   5       4.4481438504       4.5000000000   5.1856149551e-02
```

The program will also plot three lowest resulting eigenvectors (wave functions) to a file. An example plot is show in Figure 2.4.

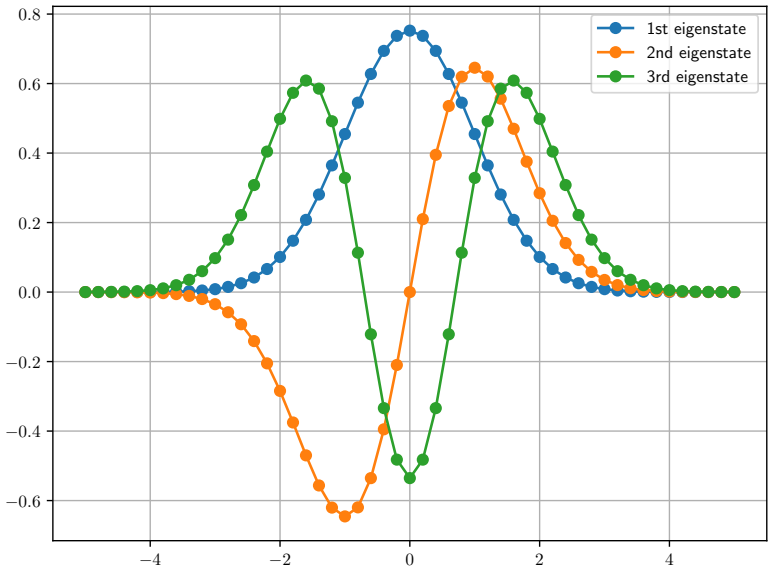

Figure 2.4: Eigenstates of harmonic oscillator

You can get more accurate values of eigenvalues by using higher value for grid points. This is the result if we use $N = 81$.

```
Eigenvalues
State         Approx              Exact          Difference
   1       0.4995112405       0.5000000000   4.8875946328e-04
   2       1.4975542824       1.5000000000   2.4457175928e-03
```

```
   3       2.4936355672       2.5000000000   6.3644328157e-03
   4       3.4877496130       3.5000000000   1.2250387024e-02
   5       4.4798939398       4.5000000000   2.0106060206e-02
```

2.4 Higher order finite difference

To obtain higher accuracy, we can include more points in our approximation to second derivative operator. An example is by using 5-point finite-difference formula:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi_i = \frac{-\psi_{i+2} + 16\psi_{i+1} - 30\psi_i + 16\psi_{i-1} - \psi_{i-2}}{12h^2} \tag{2.14}$$

The corresponding second derivative matrix can be built using the following function.

```
function build_D2_matrix_5pt( N::Int64, h::Float64 )
  mat = zeros(Float64,N,N)
  for i = 1:N-2
    mat[i,i] = -30.0
    mat[i,i+1] = 16.0
    mat[i,i+2] = -1.0
    mat[i+1,i] = mat[i,i+1]
    mat[i+2,i] = mat[i,i+2]
  end
  #
  mat[N-1,N-1] = -30.0
  mat[N-1,N] = 16.0
  mat[N,N-1] = mat[N-2,N-1]
  mat[N,N] = -30.0
  #
  return mat/(12*h^2)
end
```

We will refer to the number of points used to approximate the derivative operator as stencil order, thus the Equation (2.14) has stencil order of 5 and Equation (2.6) has stencil order of 3. Other approximation formulas for second derivative also can be used. In the source code repositoyry, we have implemented 3-, 5-, 7-, 9-, and 11-points formulas. The web page `http://web.media.mit.edu/~crtaylor/calculator.html` can be consulted to obtain the coefficients of finite-difference approximation.

In Figure 2.5 we plot the difference (approximation - exact) of second derivative of the Gaussian function for several approximation formulas using $N = 15$ grid points. The errors have have different magnitude for each points with the maximum error occurs at the center of the Gaussian. We can readily see that the errors for each point are decreasing with increasing stencil order.

2.5 Harmonic potential with higher order finite-difference formulas

We can compare the effect of different stencil order to the accuracy of numerical approximation to the Schroedinger equation. In the following program, we try to compare the accuracy of first eigenvalue of Schroedinger equation with harmonic potential.

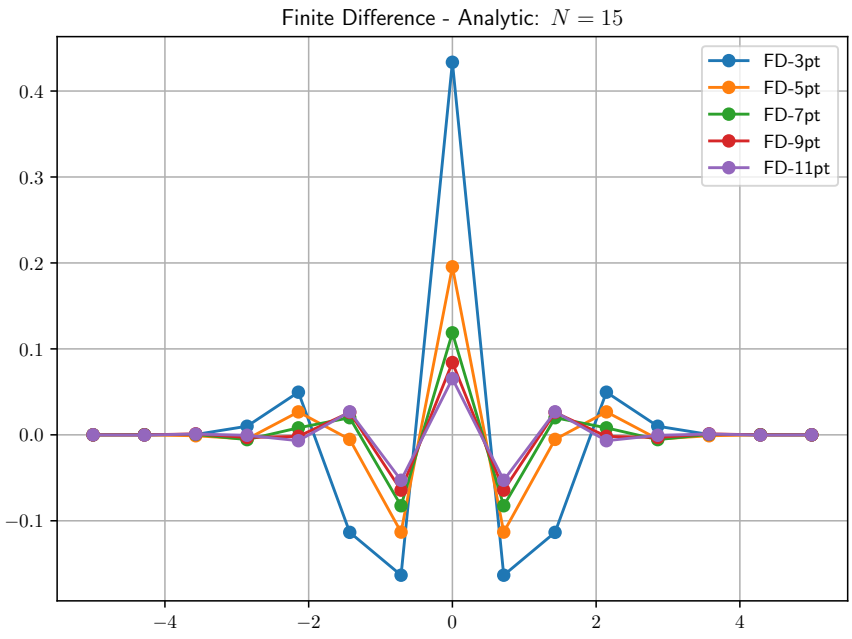

Figure 2.5: Error of 2nd derivative approx

```
function solve_eigvals(N::Int64, D2_func)
  # Initialize the grid points
  x_min = -5.0
  x_max =  5.0
  x, h = init_FD1d_grid(x_min, x_max, N)
  # Build 2nd derivative matrix
  D2 = D2_func(N, h)
  # Potential
  Vpot = pot_harmonic.(x)
  # Hamiltonian
  Ham = -0.5*D2 + diagm( 0 => Vpot )
  # Solve for the eigenvalues only
  evals = eigvals( Ham )
  #
  return evals
end

function main()
  Npoints = 20
  funcs = [build_D2_matrix_3pt, build_D2_matrix_5pt, build_D2_matrix_7pt,
           build_D2_matrix_9pt, build_D2_matrix_11pt]

  ω = 1.0
  hbar = 1.0
  ist = 1
  E_ana = (2*ist - 1)*ω*hbar/2

  for f in funcs
      evals = solve_eigvals(Npoints, f)
      dE = abs(evals[ist]-E_ana) # absolute error
      @printf("%20s: %18.10f %18.10e\n", string(f), evals[ist], dE)
  end
```

```
end

main()
```

The results for fixed point ($N = 20$) and different stencil order are given below. The second colum is the first eigenvalue and the third colum in the difference with exact value (0.5 Hartree). The results clearly shows that the error decreases with with increasing stencil order.

```
  build_D2_matrix_3pt:       0.4911851752   8.8148248058e-03
  build_D2_matrix_5pt:       0.4992583463   7.4165366457e-04
  build_D2_matrix_7pt:       0.4998965249   1.0347512122e-04
  build_D2_matrix_9pt:       0.4999802170   1.9782954165e-05
 build_D2_matrix_11pt:       0.4999952673   4.7326636594e-06
```

2.6 Exercises

Potential well

Gaussian potential

Periodic boundary condition

Chapter 3

Schroedinger equation in 2d

Now we will turn our attention to higher dimensions, i.e 2d. The time-independent Schrodinger equation in 2d can be written as:

$$\left[-\frac{1}{2}\nabla^2 + V(x,y)\right]\psi(x,y) = E\,\psi(x,y) \tag{3.1}$$

where ∇^2 is the Laplacian operator:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3.2}$$

3.1 Describing grid in 2d

Now we have two directions x and y. Our approach to solving the Schroedinger equation is similar to the one we have used before in 1d, however several technical difficulties will arise.

To describe the computational grid, we now need to specify $x_{\max}, x_{\min}$ for the x-domain and $y_{\max}, y_{\min}$ for y-domain. We also need to specify number of grid points in for each x and y-directions, i.e. N_x and N_y. There are quite lot of variables. For easier management, we will collect our grid related variables in one data structure or `struct` in Julia. A `struct` in Julia looks very much like C-struct. It also defines a new custom data type in Julia.

Our struct definition looks like this.

```
struct FD2dGrid
  Npoints::Int64
  Lx::Float64
  Ly::Float64
  Nx::Int64
  Ny::Int64
  hx::Float64
  hy::Float64
  dA::Float64
  x::Array{Float64,1}
  y::Array{Float64,1}
  r::Array{Float64,2}
  idx_ip2xy::Array{Int64,2}
  idx_xy2ip::Array{Int64,2}
  pbc::Tuple{Bool,Bool}
end
```

An instance of `FD2dGrid` can be initialized using the following constructor function:

```
function FD2dGrid( x_domain, Nx, y_domain, Ny )
  x, hx = init_FD1d_grid(x_domain, Nx)
  y, hy = init_FD1d_grid(y_domain, Ny)
  dA = hx*hy
  Npoints = Nx*Ny
  r = zeros(2,Npoints)
  ip = 0
  idx_ip2xy = zeros(Int64,2,Npoints)
  idx_xy2ip = zeros(Int64,Nx,Ny)
  for j in 1:Ny
    for i in 1:Nx
      ip = ip + 1
      r[1,ip] = x[i]
      r[2,ip] = y[j]
      idx_ip2xy[1,ip] = i
      idx_ip2xy[2,ip] = j
      idx_xy2ip[i,j] = ip
    end
  end
  return FD2dGrid(Npoints, Nx, Ny, hx, hy, dA, x, y, r, idx_ip2xy, idx_xy2ip)
end
```

A short explanation about the members of `FD2dGrid` follows.

- `Npoints` is the total number of grid points.
- `Nx` and `Ny` is the total number of grid points in x and y-directions, respectively.
- `hx` and `hy` is grid spacing in x and y-directions, respectively. `dA` is the product of `hx` and `hy`.
- `x` and `y` are the grid points in x and y-directions. The actual two dimensional grid points $r \equiv (x_i, y_i)$ are stored as two dimensional array r.
- Thw two integers arrays `idx_ip2xy` and `idx_xy2ip` defines mapping between two dimensional grids and linear grids.

As an illustration let's build a grid for a rectangular domain $x_{\text{min}} = y_{\text{min}} = -5$ and $x_{\text{max}} = y_{\text{max}} = 5$ and $N_x = 3$, $N_y = 4$. Using the above constructor for `FD2dGrid`:

```
Nx = 3
Ny = 4
grid = FD2dGrid( (-5.0,5.0), Nx, (-5.0,5.0), Ny )
```

Dividing the x and y accordingly we obtain $N_x = 3$ grid points along x-direction

```
julia> println(grid.x)
[-5.0, 0.0, 5.0]
```

and $N_y = 4$ points along the y-direction

```
julia> println(grid.y)
[-5.0, -1.6666666666666665, 1.666666666666667, 5.0]
```

The actual grid points are stored in `grid.r`. Using the following snippet, we can printout all of the grid points:

```
for ip = 1:grid.Npoints
    @printf("%3d %8.3f %8.3f\n", ip, grid.r[1,ip], grid.r[2,ip])
end
```

The results are:

```
  1   -5.000   -5.000
  2    0.000   -5.000
  3    5.000   -5.000
  4   -5.000   -1.667
  5    0.000   -1.667
  6    5.000   -1.667
  7   -5.000    1.667
  8    0.000    1.667
  9    5.000    1.667
 10   -5.000    5.000
 11    0.000    5.000
 12    5.000    5.000
```

We also can use the usual rearrange these points in the usual 2d grid rearrangement:

```
[  -5.000,  -5.000] [  -5.000,  -1.667] [  -5.000,   1.667] [  -5.000,   5.000]
[   0.000,  -5.000] [   0.000,  -1.667] [   0.000,   1.667] [   0.000,   5.000]
[   5.000,  -5.000] [   5.000,  -1.667] [   5.000,   1.667] [   5.000,   5.000]
```

which can be produced from the following snippet:

```
for i = 1:Nx
    for j = 1:Ny
        ip = grid.idx_xy2ip[i,j]
        @printf("[%8.3f,%8.3f] ", grid.r[1,ip], grid.r[2,ip])
    end
    @printf("\n")
end
```

3.2 Laplacian operator

Having built out 2d grid, we now turn our attention to the second derivative operator or the Laplacian in the equation 3.1. There are several ways to build a matrix representation of the Laplacian, but we will use the easiest one.

Before constructing the Laplacian matrix, there is an important observation that we should make about the second derivative matrix $\mathbb{D}^{(2)}$. We should note that the second derivative matrix contains mostly zeros. This type of matrix that most of its elements are zeros is called **sparse matrix**. In a sparse matrix data structure, we only store its non-zero elements with specific formats such as compressed sparse row/column format (CSR/CSC) and coordinate format. We have not made use of the sparsity of the second derivative matrix in the 1d case for simplicity. In the higher dimensions, however, we must make use of this sparsity, otherwise we will waste computational resources by storing many zeros. The Laplacian matrix that we will build from $\mathbb{D}^{(2)}$ is also very sparse.

Given second derivative matrix in x, $\mathbb{D}_x^{(2)}$, y direction, $\mathbb{D}_x^{(2)}$, we can construct finite difference representation of the Laplacian operator $\mathbb{L}$ by using

$$\mathbb{L} = \mathbb{D}_x^{(2)} \otimes \mathbb{I}_y + \mathbb{I}_x \otimes \mathbb{D}_y^{(2)} \tag{3.3}$$

where $\otimes$ is Kronecker product. In Julia, we can use the function `kron` to form the Kronecker product between two matrices `A` and `B` as `kron(A,B)`.

The following function illustrates the above approach to construct matrix representation of the Laplacian operator.

```
function build_nabla2_matrix( grid::FD2dGrid )
  Nx = grid.Nx; hx = grid.hx
  Ny = grid.Ny; hy = grid.hy
  D2x = build_D2_matrix_9pt(Nx, hx)
  D2y = build_D2_matrix_9pt(Ny, hy)
  ∇2 = kron(D2x, speye(Ny)) + kron(speye(Nx), D2y)
  return ∇2
end
```

The standard Julia library does not include definition for speye function but it can be implemented by the following definition.

```
speye(N::Int64) = sparse( Matrix(1.0I, N, N) )
```

In the Figure 3.1, an example to the approximation of 2nd derivative of 2d Gaussian function by using finite difference is shown.

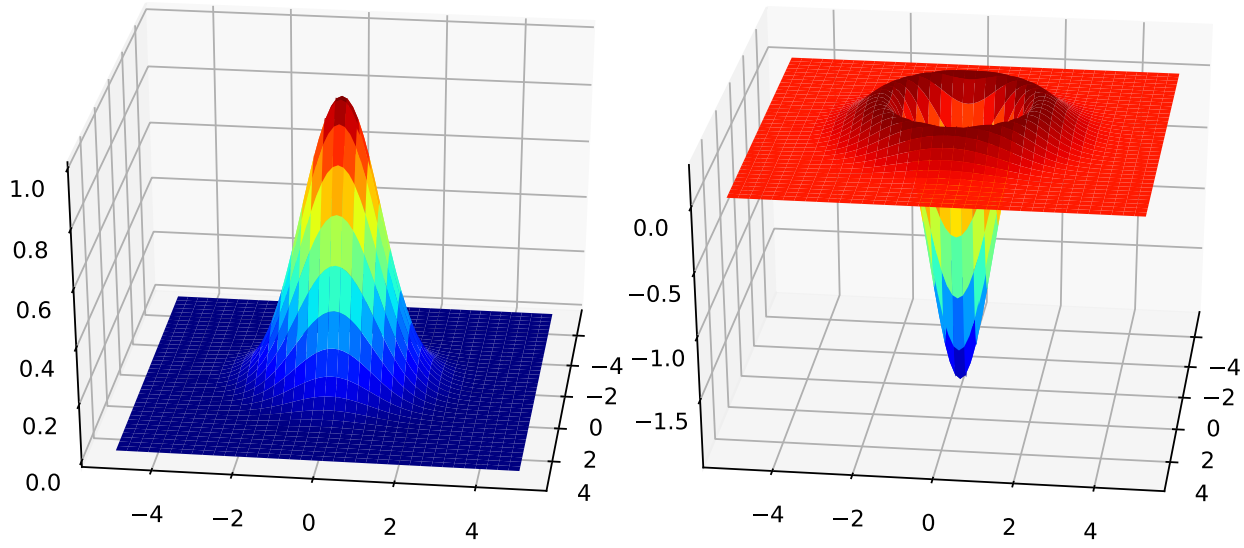

Figure 3.1: Two-dimensional Gaussian function and its finite difference approximation of second derivative

The following program is used to produce the figure.

```
using Printf
using LinearAlgebra
using SparseArrays

import PyPlot
const plt = PyPlot

include("FD2dGrid.jl")
include("build_nabla2_matrix.jl")
include("supporting_functions.jl")

function my_gaussian( grid; α=1.0 )
```

```
  Npoints = grid.Npoints
  f = zeros(Npoints)
  for i in 1:Npoints
    x = grid.r[1,i]
    y = grid.r[2,i]
    r2 = x^2 + y^2
    f[i] = exp(-α*r2)
  end
  return f
end

function main()
  Nx = 75
  Ny = 75
  grid = FD2dGrid( (-5.0,5.0), Nx, (-5.0,5.0), Ny )

  ∇2 = build_nabla2_matrix( grid )

  fg = my_gaussian(grid, α=0.5)
  plt.clf()
  plt.surf(grid.x, grid.y, reshape(fg, grid.Nx, grid.Ny), cmap=:jet)
  plt.gca(projection="3d").view_init(30,7)
  fileplot = "IMG_gaussian2d.pdf"
  plt.savefig(fileplot)

  d2fg = ∇2*fg
  plt.clf()
  plt.surf(grid.x, grid.y, reshape(d2fg, grid.Nx, grid.Ny), cmap=:jet)
  plt.gca(projection="3d").view_init(30,7)
  fileplot = "IMG_d2_gaussian2d.pdf"
  plt.savefig(fileplot)
end

main()
```

3.3 More about sparse matrices in Julia

Using the function `typeof` we can know the type of the variable `∇2` which was returned by the function `build_nabla2_matrix`:

```
julia> typeof(∇2)
SparseMatrixCSC{Float64,Int64}
```

The `CSC` part here is actually describe a certain format to store a sparse matrix. It stands for **C**ompressed **S**parse **C**olumn format. There are other formats as well: such as Compressed Sparse Row format (CSR), coordinate list format (COO), and several others.

The CSC format stores a sparse matrix **A** with size $m \times n$ using three (one-dimensional) arrays (nzval, colptr, rowval). Let Nnz denote the number of nonzero entries in **A**.

The arrays nzval and rowval are of length Nnz, and contain the non-zero values and the row indices of those values respectively.

An integer array colptr contains pointers to the beginning of each column in the arrays nzval and rowval. Thus the content of colptr[i] is the position in arrays nzval and rowval where the i-th row starts. The length of colptr

is n + 1 with colptr[n + 1] containing the number colptr[1] + Nnz, i.e., the address in nzval and rowval of the beginning of a fictitious column m + 1.

The array colptr has one element per column in the matrix and encodes the index in nzval where the given column starts. It may contain an extra end element which is set to Nnz.

Field names:

```
julia> fieldnames(typeof(∇2))
(:m, :n, :colptr, :rowval, :nzval)
```

From the documentation

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
  m::Int                # Number of rows
  n::Int                # Number of columns
  colptr::Vector{Ti}    # Column j is in colptr[j]:(colptr[j+1]-1)
  rowval::Vector{Ti}    # Row indices of stored values
  nzval::Vector{Tv}     # Stored values, typically nonzeros
end
```

Example 1:

$$A = \begin{bmatrix} 0 & 0 & 1.0 & 0.0 \\ 5 & 8 & 0.0 & 0.0 \\ 0 & 0 & 3.0 & 0.0 \\ 0 & 6 & 0.0 & 0.0 \\ 0 & 0 & 0.0 & 0.0 \end{bmatrix} \tag{3.4}$$

```
nzval = [5.0, 8.0, 6.0, 1.0, 3.0]
rowval = [2, 2, 4, 1, 3]
colptr = [1, 2, 4, 6, 6]
```

Example 2:

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 1 & 5 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 6 \end{bmatrix} \tag{3.5}$$

```
nzval  = [5.0, 5.0, 1.0, 3.0, 1.0, 5.0, 6.0]
rowval = [2, 6, 1, 3, 5, 1, 6]
colptr = [1, 3, 3, 6, 8]
```

Example 3:

$$\mathbf{C} = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{3.6}$$

```
nzval  = [-2.0, 1.0, 1.0, -2.0, 1.0, 1.0, -2.0, 1.0, 1.0, -2.0, 1.0, 1.0, -2.0]
rowval = [1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5]
colptr = [1, 3, 6, 9, 12, 14]
```

Another way: sparse(I,J,V) then constructs a sparse matrix such that S[I[k], J[k]] = V[k].

3.4 Iterative methods for eigenvalue problem

Now that we know how to build the Laplacian matrix, we now can build the Hamiltonian matrix given some potential:

```
∇2 = build_nabla2_matrix( grid )
Ham = -0.5*∇2 + spdiagm( 0 => Vpot )
```

Note that we have used sparse diagonal matrix for building the potential matrix by using the function `spdiagm`. Our next task after building the Hamiltonian matrix is to find the eigenvalues and eigenfunctions. However, note that the Hamiltonian matrix size is large. For example, if we use $N_x = 50$ and $N_y = 50$ we will end up with a Hamiltonian matrix with the size of 2500. The use of `eigen` method, as we have done in the 1d case, to solve this eigenvalue problem is thus not practical. Actually, given enough computer memory and time, we can use the function `eigen` anyway to find all eigenvalue and eigenfunction pairs of the Hamiltonian, however it is not recommended nor practical for larger problem size.

Typically, we do not need to solve for all eigenvalue and eigenfunction pairs. We only need to solve for several eigenpairs with lowest eigenvalues. In a typical density functional theory calculations, we only need to solve for $N_{\text{electrons}}$ or $N_{\text{electrons}}/2$ lowest states, where $N_{\text{electrons}}$ is the number of electrons in the system.

In numerical methods, there are several methods to search for several eigenpairs of a matrix. These methods falls into the category of *partial or iterative diagonalization methods*. Several known methods are Lanczos method, Davidson method, preconditioned conjugate gradients, etc. More detailed discussion about these methods are deferred to Appendix XXX. We have prepared several implementation of iterative diagonalization methods which can be used for black boxes for your convenience:

- `diag_Emin_PCG`
- `diag_davidson`
- `diag_LOBPCG`

These functions have similar function signatures. They typically need Hamiltonian, initial guess of wave function and preconditioner. An example of `diag_LOBPCG` is given below.

```
function diag_LOBPCG!( Ham, X::Array{Float64,2}, prec;
                       tol=1e-5, NiterMax=100, verbose=false,
                       verbose_last=false, Nstates_conv=0 )
```

The three mandatory arguments are as follow:

- `Ham`: the Hamiltonian matrix
- `X::Array{Float64,2}`: initial guess of eigenfunctions
- `prec`: the preconditioner

Almost all iterative methods need a good preconditioner to function properly. We will use several ready-to-use preconditioners that have been implemented in several packages in Julia such as incomplete LU and multigrid preconditioners. In the next section we will describe a diagonalization method based on nonlinear minimization as we apply it to a simple case of 2d harmonic potential.

3.5 Minimization approach: 2d harmonic potential

We will test our implementation for solving Schroedinger equation for two dimensional harmonic potentials:

$$V(x,y) = \frac{1}{2}\omega^2(x^2+y^2) \tag{3.7}$$

This potential can be implemented in the following Julia code.

```
function pot_harmonic( grid::FD2dGrid; ω=1.0 )
  Npoints = grid.Npoints
  Vpot = zeros(Npoints)
  for i in 1:Npoints
    x = grid.r[1,i]
    y = grid.r[2,i]
    Vpot[i] = 0.5 * ω^2 *( x^2 + y^2 )
  end
  return Vpot
end
```

This potential has the following analytic solutions for eigenvalues:

$$E_{n_x+n_y} = \hbar\omega\,(n_x+n_y+1) \tag{3.8}$$

For each energy levels, we may have more than one possible combinations of n_x and n_y, for examples:

$E = n_x + n_y + 1$	Values of n_x and n_y
1	(0,0)
2	(1,0) (0,1)
3	(2,0) (1,1) (1,1)
4	(3,0) (0,3) (2,1) (1,2)

Our first task is to build the Hamiltonian. The step is very similar to the one that we have done for 1d case:

```
Nx = 50
Ny = 50
grid = FD2dGrid( (-5.0,5.0), Nx, (-5.0,5.0), Ny )
∇2 = build_nabla2_matrix( grid )
Vpot = pot_harmonic( grid )
Ham = -0.5*∇2 + spdiagm( 0 => Vpot )
```

Note that we have used `spdiagm` instead of `diagm` when constructing the potential matrix.

3.5.1 Orthonormalization

We usually need to provide a guess solution or vectors to our eigensolver. The guess vectors should be orthonormalized properly before they can be supplied to the eigensolver. There are several algorithms that can be used

to orthonormalize the vectors. One of them that will be used here is the Lowdin orthonormalization which can be implemented in Julia as:

```
function ortho_sqrt( psi )
  Udagger = inv(sqrt(psi'*psi))
  return psi*Udagger
end
```

As an example usage, here we start from random vectors and orthonormalize them with `ortho_sqrt`:

```
dVol = grid.dVol
Nstates = 3
Npoints = Nx*Ny
X = rand(Float64, Npoints, Nstates)
ortho_sqrt!(X, dVol)
```

TODO: Gram-Schmidt ortho

TODO: check that the vectors are properly orthonormalized.

3.5.2 Band energy functional and its gradient

One method that can be used is diagonalize the Hamiltonian is based on the minimization of band energy:

$$\min E\left[\psi_i\right] = \langle\psi|H|\psi\rangle - \sum_{ij} \lambda_{ij} \left(\langle\psi_i|\psi_j\rangle - \delta_{ij}\right) \tag{3.9}$$

The gradient of this expression is:

$$\frac{\partial H}{\partial \psi_i^*} = H\psi_i - \sum_j \psi_j \langle\psi_j|H|\psi_i\rangle \tag{3.10}$$

In Julia this can be implemented using the following function:

```
function calc_grad_evals!( Ham, ψ, g, Hsub )
  Nstates = size(ψ,2)
  Hψ = Ham*ψ
  Hsub[:] = ψ' * Hψ
  g[:,:] = Hψ - ψ*Hsub
  return
end
```

3.5.3 Steepest descent method

The simplest method that we can apply to minimize the band energy functional is the steepest descent method. This method can be described in the following steps:

1. Initialize random wave function: $\mathbf{X}$
2. Calculate E for the given $\mathbf{X}$.
3. Calculate the gradient $\mathbf{g} = \nabla E$.
4. Set search direction $\mathbf{d} = -\mathbf{g}$.

5. Update wave function according to: $\mathbf{X} \leftarrow X + \alpha d$, where α is a fixed step length. Orthonormalize $\mathbf{X}$.

6. Calculate E for this updated $\mathbf{X}$ and compare it with the previous value.

The following Julia code implements the steepest descent algorithm for minimizing band energy.

```
for iter = 1:NiterMax
  calc_grad_evals!( Ham, X, g, Hsub )
  d[:] = -g[:]
  # Update wavefunction
  X[:] = X + α_t*d
  ortho_sqrt!(X)
  Hr = Hermitian( X' * ( Ham*X ) )
  evals = eigvals(Hr)
  Ebands = sum(evals)
  devals = abs.( evals - evals_old )
  evals_old = copy(evals)
  nconv = length( findall( devals .< tol ) )
  diffE = abs(Ebands-Ebands_old)
  @printf("SD step %8d = %18.10f   %10.7e  nconv = %5d\n", iter, Ebands, diffE, nconv)
  if nconv >= Nstates_conv
      IS_CONVERGED = true
      @printf("Emin_SD convergence: nconv = %5d in %5d iterations\n", nconv, iter)
      break
  end
  Ebands_old = Ebands
end
```

Note we have calculate the current estimate the eigenvalues and band energy in the following lines:

```
Hr = Hermitian( X' * ( Ham*X ) )
evals = eigvals(Hr)
Ebands = sum(evals)
```

In the file `main_harmonic_Emin_SD.kl` we implement steepest descent algorithm to find three lowest eigenstates of 2d harmonic potential. Using the following parameters:

```
NiterMax = 5000
α = 3e-3
tol = 1e-6
```

we obtain the results:

```
evals[  1] =       0.9999999732 devals =   5.2402526762e-14
evals[  2] =       2.0000187702 devals =   1.1408453116e-07
evals[  3] =       2.0001656321 devals =   9.9907378681e-07
```

which are close to the analytical solution. You might want to vary the number of eigenstates that you want to search by setting higher values for the variable `Nstates` to higher than 3.

Despite its simplicity, the steepest descent method has several drawbacks. The most prominent one is that it is very slow to converge. For this particular case we have:

```
SD step     2258 =       5.0001854888   1.1198655e-06  nconv =     2
SD step     2259 =       5.0001843756   1.1131584e-06  nconv =     3
Emin_SD convergence: nconv =     3 in  2259 iterations
```

The convergence is also dependent to the step length parameter α.

3.5.4 Line minimization

Line minimization:

```
Xc = ortho_sqrt( X + α_t*d )
calc_grad_evals!( Ham, Xc, gt, Hsub )
denum = real(sum(conj(g-gt).*d))
if denum != 0.0
    α = abs( α_t*real(sum(conj(g).*d))/denum )
else
    α = 0.0
end
```

Implemented in the program `main_harmonic_Emin_linmin.jl`. Result using line minimization:

```
linmin step      589 =       5.0000566204   1.4525395e-06  nconv =     2
linmin step      590 =       5.0000552040   1.4164238e-06  nconv =     3
Emin_linmin convergence: nconv =     3 in   590 iterations

Eigenvalues:

evals[  1] =       1.0000001750 devals =   3.4402208904e-07
evals[  2] =       2.0000051631 devals =   1.0311948540e-07
evals[  3] =       2.0000498659 devals =   9.6928218651e-07
```

3.5.5 Preconditioning

Action of preconditioning:

```
Kg[:] = g[:] # copy
for i in 1:Nstates
    @views ldiv!(prec, Kg[:,i])
end
```

ILU preconditioner based on kinetic operator:

```
prec = ilu(-0.5*∇2)
```

Result using ILU preconditioner based on kinetic operator:

```
linmin step       55 =       5.0000009960   1.2258080e-06  nconv =     2
linmin step       56 =       5.0000003689   6.2711750e-07  nconv =     3
Emin_linmin convergence: nconv =     3 in    56 iterations

Eigenvalues:

evals[  1] =       0.9999999739 devals =   6.0261565960e-07
evals[  2] =       1.9999999937 devals =   5.8729066055e-09
```

```
evals[  3] =       2.0000004014 devals =   1.8628935727e-08
```

ILU preconditioner based on Hamiltonian operator:

```
prec = ilu(Ham)
```

```
linmin step       15 =       5.0000004295   3.2233493e-06  nconv =     2
linmin step       16 =       4.9999998058   6.2374058e-07  nconv =     3
Emin_linmin convergence: nconv =     3 in    16 iterations

Eigenvalues:

evals[  1] =       0.9999999787 devals =   1.2492943446e-07
evals[  2] =       1.9999998478 devals =   1.2682084205e-07
evals[  3] =       1.9999999792 devals =   3.7199030434e-07
```

Multigrid preconditioner

```
prec = aspreconditioner(ruge_stuben(Ham))
```

```
linmin step       15 =       5.0000010097   4.8543236e-06  nconv =     2
linmin step       16 =       4.9999999515   1.0581623e-06  nconv =     3
Emin_linmin convergence: nconv =     3 in    16 iterations

Eigenvalues:

evals[  1] =       0.9999999796 devals =   1.0151917307e-07
evals[  2] =       1.9999998532 devals =   1.9803365658e-07
evals[  3] =       2.0000001187 devals =   7.5860946458e-07
```

ILU0 preconditioner, in SPARSKIT.

```
prec = ILU0Preconditioner(Ham)
```

```
linmin step       64 =       5.0000041335   1.4349680e-06  nconv =     2
linmin step       65 =       5.0000030469   1.0865516e-06  nconv =     3
Emin_linmin convergence: nconv =     3 in    65 iterations

Eigenvalues:

evals[  1] =       1.0000001632 devals =   1.3899380646e-07
evals[  2] =       2.0000000485 devals =   6.1344835878e-08
evals[  3] =       2.0000028351 devals =   8.8621293415e-07
```

For testing purpose

```
prec = NoPreconditioner()
```

Comparison of preconditioner size

```
sizeof Ham  =       0.6372146606 MiB
sizeof prec =       6.9084167480 MiB  ILU kinetic
sizeof prec =       3.0494079590 MiB  ILU Hamiltonian
sizeof prec =       2.3046264648 MiB  AMG Ruge-Stuben
sizeof prec =       0.6372070313 MiB  ILU0
```

3.5.6 Conjugate gradient

Conjugate-gradient

```
d = -Kg + β * d_prev
```

Polak-Ribiere formula

```
if iter != 1
  β = real(sum(conj(g-g_prev).*Kg))/real(sum(conj(g_prev).*Kg_prev))
end
if β < 0.0 β = 0.0 end
```

CG using ILU0 (Ham)

```
CG step       29 =       5.0000026056   2.8129344e-06  nconv =     2
CG step       30 =       5.0000011500   1.4555620e-06  nconv =     3
Emin_CG convergence: nconv =     3 in    30 iterations

Eigenvalues:

evals[  1] =       1.0000001156 devals =   1.4606434040e-07
evals[  2] =       2.0000002046 devals =   6.6796002196e-07
evals[  3] =       2.0000008298 devals =   6.4153765811e-07
```

3.5.7 Eigenfunctions

```
ortho_sqrt!(X)
Hr = Hermitian( X' * (Ham*X) )
evals, evecs = eigen(Hr)
X[:,:] = X*evecs
```

The eigenfunctions are shown in Figure 3.2.

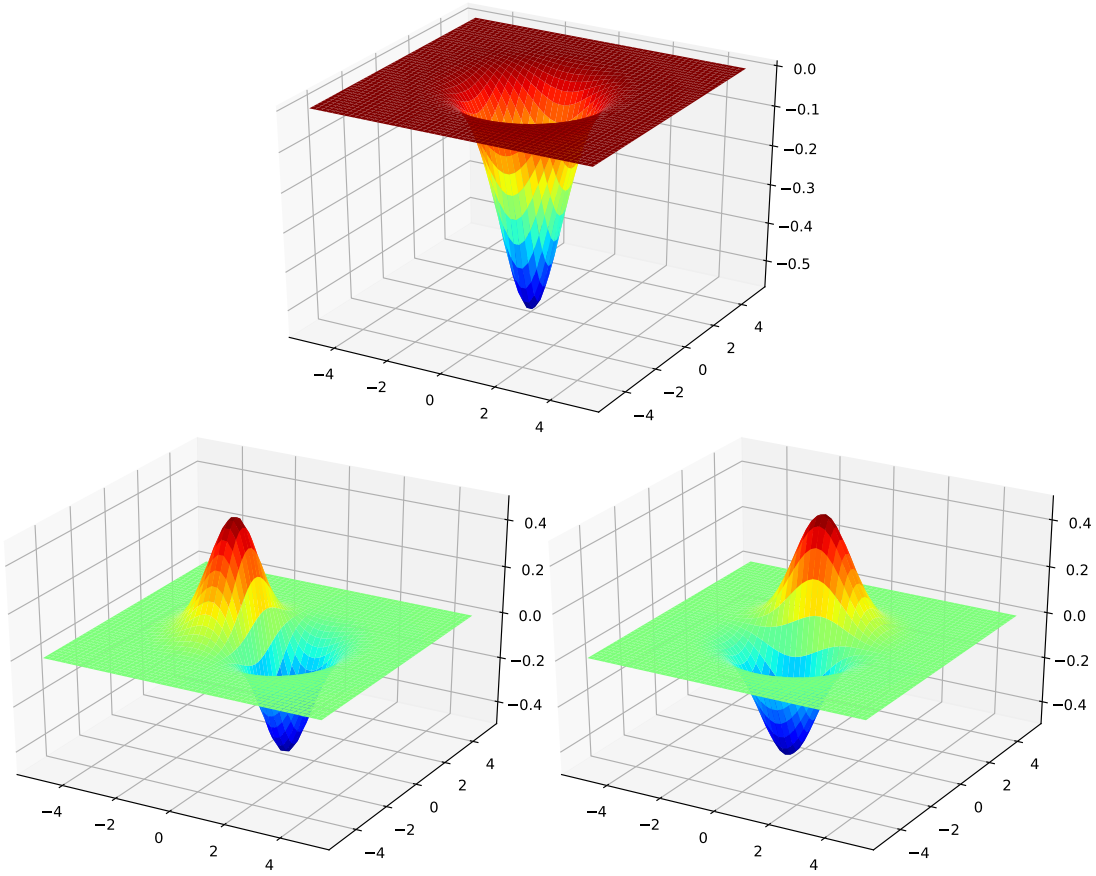

Figure 3.2: Visualization of eigenstates of 2d harmonic potential

3.6 Exercises

2d Gaussian potential

Chapter 4

Schroedinger equation in 3d

After we have considered two-dimensional Schroedinger equations, we are now ready for the extension to three-dimensional systems. In 3d, Schroedinger equation can be written as:

$$\left[-\frac{1}{2}\nabla^2 + V(\mathbf{r})\right]\psi(\mathbf{r}) = E\,\psi(\mathbf{r}) \tag{4.1}$$

where $\mathbf{r}$ is the abbreviation to (x, y, z) and ∇^2 is the Laplacian operator in 3d:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{4.2}$$

4.0.1 Three-dimensional grid

As in the preceeding chapter, our first task is to create a representation of 3d grid points and various quantities defined on it. This task is realized using straightforward extension of `FD2dGrid` to `FD3dGrid`.

Visualization of 3d functions as isosurface map or slice of 3d array.

Introducing 3d xsf

4.0.2 Laplacian operator

The Laplacian in 3d can be written as:

$$\mathbb{L} = \mathbb{D}_x^{(2)} \otimes \mathbb{I}_y \otimes \mathbb{I}_z + \mathbb{I}_x \otimes \mathbb{D}_y^{(2)} \otimes \mathbb{I}_z + \mathbb{I}_x \otimes \mathbb{I}_y \otimes \mathbb{D}_z^{(2)} \tag{4.3}$$

The Julia code is also similar to the one we have used for the case of 2d:

```
const ⊗ = kron
function build_nabla2_matrix( grid )
  # ... SNIPPED
  # build D2x, D2y, and D2z matrices
  IIx = speye(grid.Nx)
  IIy = speye(grid.Ny)
  IIz = speye(grid.Nz)
  ∇2 = D2x⊗IIy⊗IIz + IIx⊗D2y⊗IIz + IIx⊗IIy⊗D2z
  return ∇2
end
```

The main difference is that we have used the symbol ⊗ in place of `kron` function to make our code simpler. In Julia this symbol can be entered by typing `\otimes` in Julia console or text editors that have been set up to use Julia extension and Unicode input.

4.1 3d harmonic oscillator

We hope that at this point you will have no difficulties to create your own Schroedinger equation solver for 3d harmonic potential:

$$V(x, y, z) = \frac{1}{2}\omega^2 \left(x^2 + y^2 + z^2\right) \tag{4.4}$$

The analytic solution for the energy levels of this potential can be written as:

$$E_{n_x+n_y+n_z} = \hbar\omega \left(n_x + n_y + n_z + \frac{3}{2}\right) \tag{4.5}$$

where n_x, n_y, n_z are integers positive integers including zero. For each energy level we have the following degeneracies:

$$g_n = \frac{(n+1)(n+2)}{2} \tag{4.6}$$

where $n = n_x + n_y + n_z$. For examples:

```
n = n_x + n_y + n_z

n = 0: (1)(2)/2 = 1
n = 1: (2)(3)/2 = 3
n = 2: (3)(4)/2 = 6
```

The following is the result of `diag_Emin_PCG`, for the 4 lowest eigenstates:

```
CG step       12 =       8.9999996162 4.4341014e-06
evals[  1] =       1.4999999732, devals =   4.1020002284e-07
evals[  2] =       2.4999998628, devals =   1.0054977597e-06
evals[  3] =       2.4999998760, devals =   1.3278320172e-06
evals[  4] =       2.4999999042, devals =   1.6905715712e-06
iter 12 nconv = 4
Convergence is achieved based on nconv
```

The obtained eigenvalues are close the the analytic solution. Note that the we have 3 degenerate eigenstates. You can try to vary the number of requested eigenstates to very that the degeneracies are correct.

4.2 Hydrogen atom

4.2.1 Coulomb potential and its singularity

Until now, we only have considered simple potentials such as harmonic potential. Now we will move on and consider more realistic potentials which is used in practical electronic calculations.

For most applications in materials physics and chemistry the external potential that is felt by electrons is the Coulombic potential due to atomic nucleus. This potential has the following form:

$$V(r) = -\sum_{I}^{N_{\text{atom}}} \frac{Z_I}{|\mathbf{r} - \mathbf{R}_I|} \tag{4.7}$$

where R_I are the positions and Z_I are the charges of the atomic nucleus present in the system. We will consider the most simplest system, namely the hydrogen atom $Z_I = 1$, for which we have

$$V(r) = -\frac{1}{|\mathbf{r} - \mathbf{R}_0|} \tag{4.8}$$

The following Julia code implement the H atom potential:

```
function pot_H_atom( grid; r0=(0.0, 0.0, 0.0) )
  Npoints = grid.Npoints
  Vpot = zeros(Npoints)
  for i in 1:Npoints
    dx = grid.r[1,i] - r0[1]
    dy = grid.r[2,i] - r0[2]
    dz = grid.r[3,i] - r0[3]
    Vpot[i] = -1.0/sqrt(dx^2 + dy^2 + dz^2)
  end
  return Vpot
end
```

With only minor modification to our program for harmonic potential, we can solve the Schroedinger equation for the hydrogen atom:

```
grid = FD3dGrid( (-5.0,5.0), Nx, (-5.0,5.0), Ny, (-5.0,5.0), Nz )
∇2 = build_nabla2_matrix( grid )
Vpot = pot_H_atom( grid )
Ham = -0.5*∇2 + spdiagm( 0 => Vpot )
prec = aspreconditioner(ruge_stuben(Ham))
Nstates = 1  # only choose the lowest lying state
Npoints = Nx*Ny*Nz
X = ortho_sqrt( rand(Float64, Npoints, Nstates) ) # random initial guess of wave function
evals = diag_LOBPCG!( Ham, X, prec, verbose=true )
```

For the grid size of $N_x = N_y = N_z = 50$ and using 9-point finite-difference approximation to the second derivative operator in 1d we obtain the eigenvalue of -0.4900670759 Ha which is not too bad if compared with the exact value of -0.5 Ha. We can try to increase the grid size until we can get satisfactory result.

Note that there is a caveat when we are trying to use the Coulombic potential. This potential is divergent at $r = 0$, so care must be taken such that this divergence is not encountered in the numerical calculation. In the previous code, we have tried to achieve this by using choosing the numbers N_x, N_y, and N_z to be even numbers. This way, we avoid the evaluation of the potential at the singular point of the Coulomb potential (the point $\mathbf{r} = (0, 0, 0)$ in this particular case).

4.2.2 Pseudopotential

In the many electronic structure calculations, it is sometime convenient to replace the Coulomb potential with another potential which is smoother which we will refer to as a **pseudopotential**. Not all smooth potentials can do the job. The smooth potential should satisfy several requirements. One of the most important requirement is that the smooth potential should have similar scattering properties as the original Coulomb potential that it replaces. This means that the potential should have posses similar eigenvalues as the original Coulomb potential. Usually we don't try to reproduce all eigenvalue spectrum but only the eigenvalues which belongs to the valence electrons. The valence electrons are responsible for most chemically and physically important properties so this is an acceptable approximation for most cases.

The theory and algorithms for constructing pseudopotentials are beyond the scope of this book. Most pseudopotentials are non-local by construction and this can make our program rather complicated. In this chapter we will focus on the so-called local pseudopotential. Practically, local pseudopotentials pose no additional difficulties as the potentials that we have considered so far. As an example of a pseudopotential, we will consider the following local pseudopotential for hydrogen atom:

$$V_{\mathrm{H,ps}}(r) = -\frac{Z_{\mathrm{val}}}{r}\mathrm{erf}\left(\frac{\bar{r}}{\sqrt{2}}\right) + \exp\left(-\frac{1}{2}\bar{r}^2\right)\left(C_1 + C_2\bar{r}^2\right) \tag{4.9}$$

where $\bar{r} = r/r_{\mathrm{loc}}$ and with the parameters $r_{loc} = 0.2$, $Z_{\mathrm{val}} = 1$, $C_1 = -4.0663326$, and $C_2 = 0.6678322$.

The following Julia code implements this pseudopotential.

```
function pot_Hps_HGH( grid; r0=(0.0, 0.0, 0.0) )
  Npoints = grid.Npoints
  Vpot = zeros( Float64, Npoints )
  Zval = 1
  rloc = 0.2
  C1 = -4.0663326
  C2 = 0.6678322
  for ip = 1:Npoints
    dx2 = ( grid.r[1,ip] - r0[1] )^2
    dy2 = ( grid.r[2,ip] - r0[2] )^2
    dz2 = ( grid.r[3,ip] - r0[3] )^2
    r = sqrt(dx2 + dy2 + dz2)
    if r < eps()
      Vpot[ip] = -2*Zval/(sqrt(2*pi)*rloc) + C1
    else
      rrloc = r/rloc
      Vpot[ip] = -Zval/r * erf( r/(sqrt(2.0)*rloc) ) + (C1 +
      ↪  C2*rrloc^2)*exp(-0.5*(rrloc)^2)
    end
  end
  return Vpot
end
```

Note that, you need to import `erf` from the package `SpecialFunctions`:

```
import SpecialFunctions: erf
```

We also have been careful to not evaluate the term Z_{val}/r by adding a check for the value of r. If it is very small (close to zero, or smaller than $eps()$) then we are using the limiting value of 4.9 when $r \to 0$:

$$V_{\mathrm{H,ps}}(r) = -\frac{Z_{\mathrm{val}}}{r}\mathrm{erf}\left(\frac{r}{\sqrt{2}r_{\mathrm{loc}}}\right) + C_1 \tag{4.10}$$

$$V_{\mathrm{H,ps}}(\sqrt{2}r_{\mathrm{loc}}) = -\frac{Z_{\mathrm{val}}}{r\sqrt{2}r_{\mathrm{loc}}}\frac{\mathrm{erf}(r)}{r} + C_1 \tag{4.11}$$

$$V_{\mathrm{H,ps}}(0) = -\frac{Z_{\mathrm{val}}}{\sqrt{2}r_{\mathrm{loc}}}\frac{2}{\sqrt{\pi}} + C_1 \tag{4.12}$$

The comparison of this potential and Coulomb potential can be seen in Figure 4.1.

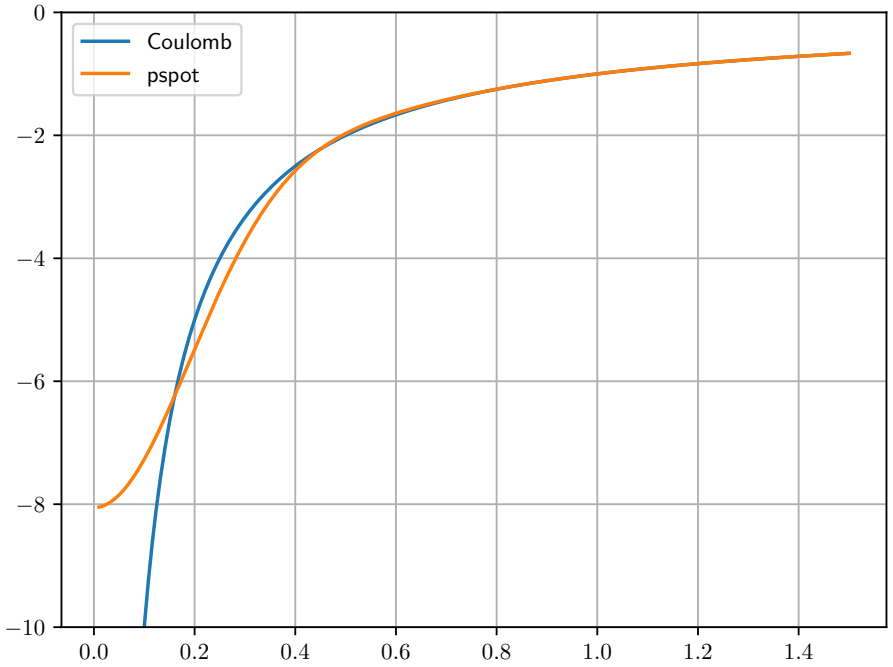

Figure 4.1: Comparison Coulomb potential vs pseudopotential

The pseudopotential in 4.9 is actually a special case of the more general form of the local part of pseudopotential developed by Goedecker-Teter-Hutter (GTH):

$$V_{\text{loc}}^{\text{ps}}(r) = -\frac{Z_{\text{val}}}{r}\text{erf}\left(\frac{\bar{r}}{\sqrt{2}}\right) + \exp\left(-\frac{1}{2}\bar{r}^2\right)\left(C_1 + C_2\bar{r}^2 + C_3\bar{r}^4 + C_4\bar{x}^6\right) \tag{4.13}$$

GTH pseudopotentials also come with the nonlocal potential which is considerably more complicated that the local one. We will consider the case of nonlocal pseudopotentials in Chapter 7. GTH pseudopotentials have are parameterized from density functional calculations for many elements in the periodic table.

4.3 Exercises

Compare the convergence of eigenvalue of hydrogen potential when using Coulomb potential vs pseudopotential.

Higher eigenstates of H atom + visualization using Xcrysden

H2 molecule

He atom

LiH molecule

Chapter 5

Poisson equation

5.1 Conjugate gradient method

In this section we will discuss a second type of equation that is important in solving Kohn-Sham equation, namely the Poisson equation. In the context of solving Kohn-Sham equation, Poisson equation is used to calculate classical electrostatic potential due to some electronic charge density. The Poisson equation that we will solve have the following form:

$$\nabla^2 V_{\text{Ha}}(\mathbf{r}) = -4\pi\rho(\mathbf{r}) \tag{5.1}$$

where $\rho(\mathbf{r})$ is the electronic density. Using finite difference discretization for the operator ∇^2 we end up with the following linear equation:

$$\mathbb{L}\mathbf{V} = \mathbf{f} \tag{5.2}$$

where $\mathbb{L}$ is the matrix representation of the Laplacian operator $\mathbf{f}$ is the discrete representation of the right hand side of the equation 5.1, and the unknown $\mathbf{V}$ is the discrete representation of the Hartree potential.

There exist several methods for solving the linear equation 5.2. We will use the so-called conjugate gradient method for solving this equation. This method is an iterative method, so it generally needs a good preconditioner to achieve good convergence. A detailed derivation about the algorithm is beyond this article and the readers are referred to several existing literatures [5, 6] and a webpage [7] for more information. The algorithm is described in `Poisson_solve_PCG.jl`

```
function Poisson_solve_PCG( Lmat, prec, f; NiterMax=1000 TOL=5.e-10 )
  Npoints = size(f,1)
  phi = zeros( Float64, Npoints )
  r = zeros( Float64, Npoints )
  p = zeros( Float64, Npoints )
  z = zeros( Float64, Npoints )
  nabla2_phi = Lmat*phi
  r = f - nabla2_phi
  z = copy(r)
  ldiv!(prec, z)
  p = copy(z)
  rsold = dot( r, z )
  for iter = 1 : NiterMax
    nabla2_phi = Lmat*p
    alpha = rsold/dot( p, nabla2_phi )
    phi = phi + alpha * p
    r = r - alpha * nabla2_phi
```

```
    z = copy(r)
    ldiv!(prec, z)
    rsnew = dot(z, r)
    deltars = rsold - rsnew
    if sqrt(abs(rsnew)) < TOL
      break
    end
    p = z + (rsnew/rsold) * p
    rsold = rsnew
  end
  return phi
end
```

To test our implementation we will adopt a problem given in Prof. Arias Practical DFT mini-course [8]. In this problem we will solve Poisson equation for a given charge density built from superposition of two Gaussian charge density:

$$\rho(\mathbf{r}) = \frac{1}{(2\pi\sigma_1^2)^{\frac{3}{2}}} \exp\left(-\frac{\mathbf{r}^2}{2\sigma_1^2}\right) - \frac{1}{(2\pi\sigma_2^2)^{\frac{3}{2}}} \exp\left(-\frac{\mathbf{r}^2}{2\sigma_2^2}\right) \tag{5.3}$$

After we obtain $V_{\text{Ha}}(\mathbf{r})$, we calculate the Hartree energy:

$$E_{\text{Ha}} = \frac{1}{2} \int \rho(\mathbf{r}) V_{\text{Ha}}(\mathbf{r}) \, \mathrm{d}\mathbf{r} \tag{5.4}$$

and compare the result with the analytical formula.

```
function test_main( NN::Array{Int64} )
  AA = [0.0, 0.0, 0.0]
  BB = [16.0, 16.0, 16.0]
  # Initialize grid
  FD = FD3dGrid( NN, AA, BB )
  # Box dimensions
  Lx = BB[1] - AA[1]
  Ly = BB[2] - AA[2]
  Lz = BB[3] - AA[3]
  # Center of the box
  x0 = Lx/2.0
  y0 = Ly/2.0
  z0 = Lz/2.0
  # Parameters for two gaussian functions
  sigma1 = 0.75
  sigma2 = 0.50

  Npoints = FD.Nx * FD.Ny * FD.Nz
  rho = zeros(Float64, Npoints)
  phi = zeros(Float64, Npoints)
  # Initialization of charge density
  dr = zeros(Float64,3)
  for ip in 1:Npoints
    dr[1] = FD.r[1,ip] - x0
    dr[2] = FD.r[2,ip] - y0
    dr[3] = FD.r[3,ip] - z0
    r = norm(dr)
    rho[ip] = exp( -r^2 / (2.0*sigma2^2) ) / (2.0*pi*sigma2^2)^1.5 -
              exp( -r^2 / (2.0*sigma1^2) ) / (2.0*pi*sigma1^2)^1.5
```

```
  end
  deltaV = FD.hx * FD.hy * FD.hz
  Laplacian3d = build_nabla2_matrix( FD, func_1d=build_D2_matrix_9pt )
  prec = aspreconditioner(ruge_stuben(Laplacian3d))
  @printf("Test norm charge: %18.10f\n", sum(rho)*deltaV)
  print("Solving Poisson equation:\n")
  phi = Poisson_solve_PCG( Laplacian3d, prec, -4*pi*rho, 1000, verbose=true, TOL=1e-10 )
  # Calculation of Hartree energy
  Unum = 0.5*sum( rho .* phi ) * deltaV
  Uana = ((1.0/sigma1 + 1.0/sigma2 )/2.0 - sqrt(2.0)/sqrt(sigma1^2 + sigma2^2))/sqrt(pi)
  @printf("Numeric  = %18.10f\n", Unum)
  @printf("Uana     = %18.10f\n", Uana)
  @printf("abs diff = %18.10e\n", abs(Unum-Uana))
end
test_main([64,64,64])
```

Result:

```
Numeric  =       0.0551434259
Uana     =       0.0551425277
abs diff =   8.9818466372e-07
```

FIXME: Needs correction, boundary condition is not treated properly.

5.2 Reciprocal space method

Using FFT, natural choice for periodic boundary condition

5.3 Direct integration

Ref: D. Sundholm. Universal method for computation of electrostatic potentials J. Chem. Phys. 122 194107 (2005)

S.A. Losilla, D. Sundholm, and J. Juselius. Direct approach to gravitation and electrostatics method for periodic systems J. Chem. Phys. 132 024102 (2010)

$$V(\mathbf{r}_1) = \int_{-\infty}^{+\infty} \rho(\mathbf{r}_2) \frac{1}{r_{12}} \, \mathrm{d}\mathbf{r}_2 \tag{5.5}$$

Rewriting Coulomb operator as integral expression using:

$$\frac{1}{r} = \frac{2}{\sqrt{2}} \int_0^{\infty} e^{-r^2 t^2} \, \mathrm{d}t \tag{5.6}$$

$$V(\mathbf{r}_1) = \frac{2}{\sqrt{\pi}} \int_0^{\infty} \int_{-\infty}^{+\infty} e^{-t^2 (\mathrm{r}_1 - \mathrm{r}_2)^2} \rho(\mathbf{r}_2) \, \mathrm{d}\mathbf{r}_2 \mathrm{d}t \tag{5.7}$$

Numerical tensorial basis:

$$\rho(\mathbf{r}_2) = \sum_{\alpha\beta\gamma} d_{\alpha\beta\gamma} \chi_\alpha(x_2) \chi_\beta(y_2) \chi_\gamma(z_2) \tag{5.8}$$

$$V_0(x_1, y_1, z_1) = \frac{2}{\sqrt{pi}} \sum_{\alpha_t} \omega_{\alpha_t} \sum_{\alpha\beta\gamma} d_{\alpha\beta\gamma} \int_{-\infty}^{\infty} e^{-t_{\alpha_t}^2 (z_1 - z_2)} \chi_\gamma(z_2) \mathrm{d}z_2 \times \int_{-\infty}^{\infty} e^{-t_{\alpha_t}^2 (y_1 - y_2)} \chi_\beta(y_2) \mathrm{d}y_2 \times \int_{-\infty}^{\infty} e^{-t_{\alpha_t}^2 (x_1 - x_2)} \chi_\alpha(x_2) \mathrm{d}x_2 \quad (5.9)$$

t_{α_t}: integration points, w_{α_t}: integration weights,

$$F_{\gamma_x \alpha_t}^{x, \alpha_x} = \int_{-\infty}^{\infty} e^{-t_{\alpha_t}^2 (x_{\alpha_x} - x_2)^2} \chi_{\gamma_x}(x_2) \, \mathrm{d}x_2 \quad (5.10)$$

$$V_{\alpha_x \alpha_y \alpha_z} = \frac{2}{\sqrt{\pi}} \sum_{\alpha_t} w_{\alpha_t} \sum_{\gamma_z} F_{\gamma_z \alpha_z}^{z, \alpha_t} \sum_{\gamma_y} F_{\gamma_y \alpha_y}^{y, \alpha_t} \sum_{\gamma_x} F_{\gamma_x \alpha_x}^{x, \alpha_t} d_{\gamma_x \gamma_y \gamma_z} \quad (5.11)$$

Chapter 6

Kohn-Sham equation part I

Using local potential only

6.1 Hartree calculation

Ignoring the XC potential.

We will introduce self-consistent field (SCF) method.

New data structure: Hamiltonian

```
mutable struct Hamiltonian
  grid::FD3dGrid
  Laplacian::SparseMatrixCSC{Float64,Int64}
  V_Ps_loc::Vector{Float64}
  V_Hartree::Vector{Float64}
  rhoe::Vector{Float64}
  precKin
  precLaplacian
end
```

```
function Hamiltonian( grid, ps_loc_func::Function )

  Laplacian = build_nabla2_matrix( grid )

  V_Ps_loc = ps_loc_func( grid )
  Npoints = grid.Npoints

  V_Hartree = zeros(Float64, Npoints)

  Rhoe = zeros(Float64, Npoints)

  precKin = aspreconditioner( ruge_stuben(-0.5*Laplacian) )
  precLaplacian = aspreconditioner( ruge_stuben(Laplacian) )

  return Hamiltonian( grid, Laplacian, V_Ps_loc, V_Hartree, Rhoe, precKin, precLaplacian )
end
```

```
import Base: *
function *( Ham::Hamiltonian, psi::Matrix{Float64} )
  Nbasis = size(psi,1)
  Nstates = size(psi,2)
  Hpsi = zeros(Float64,Nbasis,Nstates)
  # include occupation number factor
  Hpsi = -0.5*Ham.Laplacian * psi #* 2.0
  for ist in 1:Nstates, ip in 1:Nbasis
    Hpsi[ip,ist] = Hpsi[ip,ist] + ( Ham.V_Ps_loc[ip] + Ham.V_Hartree[ip] ) * psi[ip,ist]
  end
  return Hpsi
end
```

```
function update!( Ham::Hamiltonian, Rhoe::Vector{Float64} )
  Ham.rhoe[:] = Rhoe[:]
  Ham.V_Hartree = Poisson_solve_PCG(
      Ham.Laplacian, Ham.precLaplacian, -4*pi*Rhoe, 1000,
      verbose=false, TOL=1e-10
  )
  return
end
```

Calculate electron density:

```
function calc_rhoe( psi::Array{Float64,2} )
  Nbasis = size(psi,1)
  Nstates = size(psi,2)
  Rhoe = zeros(Float64,Nbasis)
  for ist in 1:Nstates
    for ip in 1:Nbasis
      Rhoe[ip] = Rhoe[ip] + 2.0*psi[ip,ist]*psi[ip,ist]
    end
  end
  return Rhoe
end
```

Calculate energy terms:

```
mutable struct Energies
  Kinetic::Float64
  Ps_loc::Float64
  Hartree::Float64
end

import Base: sum
function sum( ene::Energies )
  return ene.Kinetic + ene.Ps_loc + ene.Hartree
end

function calc_E_kin( Ham, psi::Array{Float64,2} )
  Nbasis = size(psi,1)
  Nstates = size(psi,2)
  E_kin = 0.0
  nabla2psi = zeros(Float64,Nbasis)
```

```
  dVol = Ham.grid.dVol
  # Assumption: Focc = 2 for all states
  for ist in 1:Nstates
    @views nabla2psi = -0.5*Ham.Laplacian*psi[:,ist]
    E_kin = E_kin + 2.0*dot( psi[:,ist], nabla2psi[:] )*dVol
  end
  return E_kin
end

function calc_energies( Ham::Hamiltonian, psi::Array{Float64,2} )
  dVol = Ham.grid.dVol
  E_kin = calc_E_kin( Ham, psi )
  E_Ps_loc = sum( Ham.V_Ps_loc .* Ham.rhoe )*dVol
  E_Hartree = 0.5*sum( Ham.V_Hartree .* Ham.rhoe )*dVol
  return Energies(E_kin, E_Ps_loc, E_Hartree)
end
```

Self-consistent field:

```
function pot_harmonic( grid; ω=1.0, center=[0.0, 0.0, 0.0] )
  Npoints = grid.Npoints
  Vpot = zeros(Npoints)
  for i in 1:Npoints
    x = grid.r[1,i] - center[1]
    y = grid.r[2,i] - center[2]
    z = grid.r[3,i] - center[3]
    Vpot[i] = 0.5 * ω^2 *( x^2 + y^2 + z^2 )
  end
  return Vpot
end

function main()
  AA = [-3.0, -3.0, -3.0]
  BB = [3.0, 3.0, 3.0]
  NN = [25, 25, 25]

  grid = FD3dGrid( NN, AA, BB )
  my_pot_harmonic( grid ) = pot_harmonic( grid, ω=2 )
  Ham = Hamiltonian( grid, my_pot_harmonic )

  Nbasis = prod(NN)
  dVol = grid.dVol
  Nstates = 4
  psi = rand(Float64,Nbasis,Nstates)
  ortho_sqrt!(psi)
  psi = psi/sqrt(dVol)

  Rhoe = calc_rhoe( psi )
  @printf("Integrated Rhoe = %18.10f\n", sum(Rhoe)*dVol)
  update!( Ham, Rhoe )

  evals = zeros(Float64,Nstates)
  Etot_old = 0.0
  dEtot = 0.0
  betamix = 0.5
```

```
  dRhoe = 0.0
  NiterMax = 100

  for iterSCF in 1:NiterMax
    evals = diag_LOBPCG!( Ham, psi, Ham.precKin, verbose_last=true )
    psi = psi/sqrt(dVol)
    Rhoe_new = calc_rhoe( psi )
    @printf("Integrated Rhoe_new = %18.10f\n", sum(Rhoe_new)*dVol)
    Rhoe = betamix*Rhoe_new + (1-betamix)*Rhoe
    @printf("Integrated Rhoe     = %18.10f\n", sum(Rhoe)*dVol)
    update!( Ham, Rhoe )
    Etot = sum( calc_energies( Ham, psi ) )
    dRhoe = norm(Rhoe - Rhoe_new)
    dEtot = abs(Etot - Etot_old)
    @printf("%5d %18.10f %18.10e %18.10e\n", iterSCF, Etot, dEtot, dRhoe)
    if dEtot < 1e-6
      @printf("Convergence is achieved in %d iterations\n", iterSCF)
      for i in 1:Nstates
        @printf("%3d %18.10f\n", i, evals[i])
      end
      break
    end
    Etot_old = Etot
  end
end
```

6.2 Kohn-Sham calculations

Using XC

Introduction of module:

Electrons type:

```
mutable struct Electrons
  Nelectrons::Int64
  Nstates::Int64
  Nstates_occ::Int64
  Focc::Array{Float64,1}
  energies::Array{Float64,1}
end
```

```
function Electrons( Nelectrons::Int64; Nstates_extra=0 )
  is_odd = (Nelectrons%2 == 1)
  Nstates_occ = round(Int64, Nelectrons/2)
  if is_odd
      Nstates_occ = Nstates_occ + 1
  end
  Nstates = Nstates_occ + Nstates_extra
  Focc = zeros(Float64,Nstates)
  energies = zeros(Float64,Nstates)
  if !is_odd
    for i in 1:Nstates_occ
      Focc[i] = 2.0
```

```
    end
  else
    for i in 1:Nstates_occ-1
      Focc[1] = 2.0
    end
    Focc[Nstates_occ] = 1.0
  end
  return Electrons(Nelectrons, Nstates, Nstates_occ, Focc, energies)
end
```

Example use of Electrons:

New Hamiltonian:

```
mutable struct Hamiltonian
  grid::FD3dGrid
  Laplacian::SparseMatrixCSC{Float64,Int64}
  V_Ps_loc::Vector{Float64}
  V_Hartree::Vector{Float64}
  V_XC::Vector{Float64}
  electrons::Electrons
  rhoe::Vector{Float64}
  precKin
  precLaplacian
  energies::Energies
end
```

Update the potential:

```
function update!( Ham::Hamiltonian, Rhoe::Vector{Float64} )
  Ham.rhoe = Rhoe
  Ham.V_Hartree = Poisson_solve_PCG( Ham.Laplacian, Ham.precLaplacian,
    -4*pi*Rhoe, 1000, verbose=false, TOL=1e-10 )
  Ham.V_XC = excVWN( Rhoe ) + Rhoe .* excpVWN( Rhoe )
  return
end
```

Application of Hamiltonian

```
import Base: *
function *( Ham::Hamiltonian, psi::Matrix{Float64} )
  Nbasis = size(psi,1)
  Nstates = size(psi,2)
  Hpsi = zeros(Float64,Nbasis,Nstates)
  Hpsi = -0.5*Ham.Laplacian * psi
  for ist in 1:Nstates, ip in 1:Nbasis
    Hpsi[ip,ist] = Hpsi[ip,ist] + ( Ham.V_Ps_loc[ip] + Ham.V_Hartree[ip] +
                   Ham.V_XC[ip] ) * psi[ip,ist]
  end
  return Hpsi
end
```

Chapter 7

Numerical solution of Kohn-Sham equation (part II)

Using nonlocal potential (pseudopotential)

Appendix A

Introduction to Julia programming language

This chapter is intended to as an introduction to the Julia programming language.

This chapter assumes familiarity with command line interface.

A.1 Installation

Go to `https://julialang.org/downloads/` and download the suitable file for your platform. For example, on 64 bit Linux OS, we can download the file `julia-1.x.x-linux-x86_64.tar.gz` where `1.x.x` referring to the version of Julia. After you have downloaded the tarball you can unpack it.

```
tar xvf julia-1.x.x-linux-x86_64.tar.gz
```

After unpacking the tarball, there should be a new folder called `julia-1.x.x`. You might want to put this directory under your home directory (or another directory of your preference).

A.2 Using Julia

A.2.1 Using Julia REPL

Let's assume that you have put the Julia distribution under your home directory. You can start the Julia interpreter by typing:

```
/home/username/julia-1.x.x/bin/julia
```

You should see something like this in your terminal:

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.1 (2019-05-16)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |
```

```
julia>
```

This is called the Julia REPL (read-eval-print loop) or the Julia command prompt. You can type the Julia program and see the output. This is useful for interactive exploration or debugging the program.

The Julia code can be typed after the `julia>` prompt. In this way, we can write Julia code interactively.

Example Julia session

```
julia> 1.2 + 3.4
4.6

julia> sin(2*pi)
-2.4492935982947064e-16

julia> sin(2*pi)^2 + cos(2*pi)^2
1.0
```

Using Unicode:

```
julia> α = 1234;

julia> β = 3456;

julia> α * β
4264704
```

The symbol α can be entered in the Julia console by typing `\alpha` and then followed by typing Tab key. Another example is the ∇ symbol can be entered by typing `\nabla` followed by typing Tab key. Julia supports many Unicode symbols that can be used as names or operators. We can even use superscript symbol as in ∇^2. A list of supported Unicode symbols in Julia can be found at `https://docs.julialang.org/en/v1/manual/unicode-input/`.

To exit the console we can type

```
julia> exit()
```

A.2.2 Julia script file

We also can put the code in a text file with `.jl` extension and execute it with the command:

```
julia filename.jl
```

For example, type the following code in a file named `say_hello.jl`

```
function say_hello(name)
    println("Hello: ", name)
end
say_hello("efefer")
```

The run it by using the following command in the usual terminal session (not in a Julia console)

```
julia say_hello.jl
```

and the following output will be displayed in the terminal:

```
Hello: efefer
```

A.3 Basic programming constructs

Julia has similarities with several popular programming languages such as Python, MATLAB, and R, to name a few.

A.3.1 Displaying something

Using `println` and `@printf`

```
print("Hey")
println("Hello")
```

```
using Printf
@printf("Hey")
@printf("Hello\n")
```

A.3.2 Variables

A variable, in Julia, is a name associated (or bound) to a value. It's useful when you want to store a value (that you obtained after some math, for example) for later use. For example:

```
# Assign the value 10 to the variable x
x = 10

# Doing math with x's value
x + 1

# Reassign x's value
x = 1 + 1

# You can assign values of other types, like strings of text
julia> x = "Hello World!"
```

Julia provides an extremely flexible system for naming variables. Variable names are case-sensitive, and have no semantic meaning (that is, the language will not treat variables differently based on their names).

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters (e.g. a subset of the Sm math symbols) are allowed. Subsequent characters may also include ! and digits (0-9 and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

The only explicitly disallowed names for variables are the names of built-in statements:

While Julia imposes few restrictions on valid names, it has become useful to adopt the following conventions:

Names of variables are in lower case.

Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.

Names of Types and Modules begin with a capital letter and word separation is shown with upper camel case instead of underscores.

Names of functions and macros are in lower case, without underscores.

Functions that write to their arguments have names that end in !. These are sometimes called "mutating" or "in-place" functions because they are intended to produce changes in their arguments after the function is called, not just return a value.

A.3.3 Mathematical operators

```
if a >= 1
  println("a is larger or equal to 1")
end
```

Example code 3

Plotting:

```
using PGFPlotsX
using LaTeXStrings
include("init_FD1d_grid.jl")
function my_gaussian(x::Float64; α=1.0)
  return exp( -α*x^2 )
end
function main()
  A = -5.0
  B =  5.0
  Npoints = 8
  x, h = init_FD1d_grid( A, B, Npoints )

  NptsPlot = 200
  x_dense = range(A, stop=5, length=NptsPlot)

  f = @pgf(
    Axis( {height = "6cm", width = "10cm" },
      PlotInc( {mark="none"}, Coordinates(x_dense, my_gaussian.(x_dense)) ),
      LegendEntry(L"f(x)"),
      PlotInc( Coordinates(x, my_gaussian.(x)) ),
      LegendEntry(L"Sampled $f(x)$"),
    )
  )
  pgfsave("TEMP_gaussian_1d.pdf", f)
end
main()
```

Appendix B

Lagrange basis functions

B.1 Lagrance-sinc function

$$L_i(x) = \frac{1}{\sqrt{h}} \frac{\sin\left[\pi(x - x_i)/h\right]}{\pi(x - x_i)/h} \tag{B.1}$$

$$D^{(2)}_{jl} = \begin{cases} \dfrac{1}{h^2}\dfrac{\pi^2}{6} & \text{for } j = l \\ \dfrac{(-1)^{j-l}}{h^2}\dfrac{1}{(x_j - x_l)^2} & \text{for } j \neq l \end{cases} \tag{B.2}$$

B.2 Periodic Lagrange function

For a given interval $[0, L]$, with $L > 0$, the grid points x_i appropriate for periodic Lagrange function are given by:

$$x_i = \frac{L}{2}\frac{2i - 1}{N} \tag{B.3}$$

with $i = 1, \ldots, N$. Number of points N should be an odd number.

The periodic cardinal functions $L_i^{\text{per}}(x)$, defined at grid point i are given by:

$$L_i^{\text{per}}(x) = \frac{1}{\sqrt{NL}} \sum_{n=1}^{N} \cos\left(\frac{\pi}{L}(2n - N - 1)(x - x_i)\right). \tag{B.4}$$

The expansion of periodic function in terms of Lagrange functions:

$$f(x) = \sum_{i=1}^{N} c_i L_i^{\text{per}}(x) \tag{B.5}$$

with expansion coefficients $c_i = \sqrt{L/N} f(x_i)$. When doing variational calculation, the cofficients c_i are the variational parameters. The actual function values $f(x_i)$ at grid points x_i is obtained by $f(x_i) = \sqrt{N/L}\, c_i$. The prefactor is sometimes abbreviated by $h = L/N$ and is also referred to as scaling factor.

Consider periodic potential in one dimension:

$$V(x + L) = V(x). \tag{B.6}$$

Floquet-Bloch theorem states that the wave function solution for periodic potentials can be written in the form:

$$\psi_k(x) = e^{\imath kx}\phi_k(x) \tag{B.7}$$

where function $\phi_k(x)$ and its first derivative $\phi'_k(x)$ have the same periodicity as $V(x)$ and k is a constant called the crystal momentum. Substituting this expression to Schrodinger equation we obtain:

$$\left[-\frac{\hbar^2}{2m}\left(\frac{\mathrm{d}^2}{\mathrm{d}x^2} + 2\imath k\frac{\mathrm{d}}{\mathrm{d}x} - k^2\right) + V(x)\right]\phi_k(x) = E\phi_k(k). \tag{B.8}$$

An alternative way of enforcing periodicity of the wave function is to require that:

$$\psi_k(x+L) = e^{\imath kL}\psi_k(x). \tag{B.9}$$

This condition follows from:

$$\begin{aligned}\psi_k(x+L) &= e^{\imath k(x+L)}\phi_k(x+L)\\ &= e^{\imath k(x+L)}\phi_k(x)\\ &= e^{\imath kL}e^{\imath kx}\phi_k(x)\\ &= e^{\imath kL}\psi_k(x)\end{aligned}$$

Using periodic cardinal the Schrodinger equation for periodic potential can be written as:

$$\sum_{j=1}^{N}\left[-\frac{\hbar^2}{2m}\left(D^{(2)}_{jl} + 2\imath kD^{(1)}_{jl} - k^2\delta_{jl}\right) + V(j)\delta_{jl}\right]\phi(j) = E\phi(l) \tag{B.10}$$

with $l = 1, \ldots, N$. $D^{(1)}_{jl}$ are matrix elements of the first derivatives:

$$D^{(1)}_{jl} = \begin{cases} 0 & j = l \\ -\dfrac{2\pi}{L}(-1)^{j-l}\left(2\sin\dfrac{\pi(j-l)}{N}\right)^{-1} & j \neq l \end{cases} \tag{B.11}$$

and $D^{(2)}_{jl}$ are matrix elements of the second derivatives, $N' = (N-1)/2$:

$$D^{(2)}_{jl} = \begin{cases} -\left(\dfrac{2\pi}{L}\right)^2\dfrac{N'(N'+1)}{3} & j = l \\ -\left(\dfrac{2\pi}{L}\right)^2(-1)^{j-l}\dfrac{\cos\left(\pi(j-l)/N\right)}{2\sin^2\left[\pi(j-l)/N\right]} & j \neq l \end{cases} \tag{B.12}$$

Note that, $D^{(1)}_{jl}$ is not symmetric, but $D^{(1)}_{jl} = -D^{(1)}_{lj}$. Meanwhile, the second derivative matrix $D^{(2)}_{jl}$ is symetric, i.e. $D^{(2)}_{jl} = D^{(2)}_{lj}$. With the above expressions, first and second derivative of periodic cardinals can be expressed as

$$\frac{\mathrm{d}}{\mathrm{d}x}L^{\mathrm{per}}_i(x) = \sum_{j=1}^{N}D^{(1)}_{ji}L^{\mathrm{per}}_j(x) \tag{B.13}$$

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}L^{\mathrm{per}}_i(x) = \sum_{j=1}^{N}D^{(2)}_{ji}L^{\mathrm{per}}_j(x) \tag{B.14}$$

The previous approach also can be extended to periodic potential in 3D:

$$V(\mathbf{r}) = V(x, y, z) = V\left(x + L_x, y + L_y, z + L_z\right)$$

Using periodic LF, Schrodinger equation can be casted into the following form:

$$\left[-\frac{\hbar^2}{2m}\left(\nabla^2 + 2\imath\mathbf{k}\cdot\nabla - \mathbf{k}^2\right) + V(\mathbf{r})\right]\phi_{\mathrm{k}}(\mathbf{r}) = E\ \phi_{\mathrm{k}}(\mathbf{r}) \tag{B.15}$$

B.3 Cluster Lagrange function

For a given interval $[A, B]$, with $B > A$, the grid points x_i appropriate for cluster Lagrange function are given by:

$$x_i = A + \frac{B-A}{N+1} i$$

where $i = 1, \ldots, N$. Number of points N can be either odd or even number.

The cluster Lagrange functions $L_i^{\mathrm{clu}}(x)$, defined at grid point i are given by:

$$L_i^{\mathrm{clu}}(x) = \frac{2}{\sqrt{(N+1)(B-A)}} \sum_{n=1}^{N} \sin\left(k_n(x_i - A)\right) \sin\left(k_n(x - A)\right). \tag{B.16}$$

where $k_n = \pi n/(B-A)$. The expansion of a function $f(x)$ in terms of cluster Lagrange functions:

$$f(x) = \sum_{i=1}^{N} c_i L_i^{\mathrm{clu}}(x) \tag{B.17}$$

with expansion coefficients $c_i = \sqrt{(B-A)/(N+1)} f(x_i)$. When doing variational calculation, the cofficients c_i are the variational parameters. The actual function values $f(x_i)$ at grid points x_i is obtained by $f(x_i) = \sqrt{(N+1)/(B-A)} c_i$.

Matrix elements $D_{jl}^{(2)}$ of the second derivatives for cluster Lagrange functions are

$$D_{jl}^{(2)} = \begin{cases} -\frac{1}{2}\left(\frac{\pi}{B-A}\right)^2 \frac{2(N+1)^2+1}{3} - \frac{1}{\sin^2\left[\pi j/(N+1)\right]} & j = l \\ -\frac{1}{2}\left(\frac{\pi}{B-A}\right)^2 (-1)^{j-l} \left[\frac{1}{\sin^2\left[\frac{\pi(j-l)}{2(N+1)}\right]} - \frac{1}{\sin^2\left[\frac{\pi(j+l)}{2(N+1)}\right]}\right] & j \neq l \end{cases} \tag{B.18}$$

For free or cluster boundary condition, we don't need $D_{jl}^{(1)}$.

Appendix C

Introduction to Octopus DFT code

Prerequisites:

- Autotools and GNU Make
- C, C++ and Fortran compilers
- Libxc

```
autoreconf --install
./configure --prefix=path_to_install
make
make install
```

Bibliography

[1] P. Hohenberg and W. Kohn. Inhomogenous electron gas. *Phys. Rev.*, 136:B864–871, 1964.

[2] W Kohn and L.J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140(4A):A1333–1138, 1965.

[3] List of quantum chemistry and solid-state physics software. `https://en.wikipedia.org/wiki/List_of_quantum_chemistry_and_solid-state_physics_software`.

[4] Julia programming language. `https://julialang.org/`.

[5] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409, 1952.

[6] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Available at `https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`.

[7] Conjugate gradient method. `https://en.wikipedia.org/wiki/Conjugate_gradient_method`.

[8] Practical DFT mini-course. `http://jdftx.org/PracticalDFT.html`.