TF4062: Implementing Density Functional Theory using Finite Difference Method

Iwan Prasetyo
Fadjar Fathurrahman

1 Introduction

Kohn-Sham equations:

$$\hat{H}_{\mathrm{KS}}\,\psi_i(\mathbf{r}) = \epsilon_i\,\psi_i(\mathbf{r}) \tag{1}$$

2 Schroedinger equation in 1d

We are interested in finding the bound states of 1d time-independent Schroedinger equation:

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

with the boundary conditions:

$$\lim_{x\rightarrow\pm\infty} \psi(x) = 0 \tag{3}$$

This boundary condition is relavant for non-periodic systems such as atoms and molecules.

2.1 Grid points

First 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 3 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 number of grid points. Various spatial quantities such as wave functions and potentials will be discretized on these grid points. The grid points x_i, $i = 1, 2, \ldots$ are chosen as:

$$x_i = x_{\mathrm{min}} + (i-1)h \tag{4}$$

where h is the spacing between the grid points:

$$h = \frac{x_{\mathrm{max}} - x_{\mathrm{min}}}{N-1} \tag{5}$$

The following Julia code can be used to initialize the grid points:

```julia
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
```

2.2 Approximating second derivative

Our next task is to find an approximation to the second derivative operator present in the Equation (2). One simple approximation that we can use is the 3-point (central) finite difference:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi_i \approx \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{h^2} \tag{6}$$

where we have the following notation have been used: $\psi_i = \psi(x_i)$. By taking $\{\psi_i\}$ as a column vector, the second derivative operation can be expressed as matrix multiplication:

$$\vec{\psi''} = \mathbb{D}^{(2)}\vec{\psi} \tag{7}$$

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

$$\mathbb{D}^{(2)} = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & -2 \end{bmatrix} \tag{8}$$

An example implementation can be found in the following function.

```
function build_D2_matrix_3pt( N::Int64, h::Float64 )
    mat = zeros(Float64,N,N)
    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
```

Before use this function to solve Schroedinger equation we will to test the operation in Equation (8) for a simple function which second derivative can be calculated analytically.

$$\psi(x) = \mathrm{e}^{-\alpha x^2} \tag{9}$$

which second derivative can be calculated as

$$\psi''(x) = \left(-2\alpha + 4\alpha^2 x^2\right)\mathrm{e}^{-\alpha x^2} \tag{10}$$

They are implemented in the following code

```
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
```

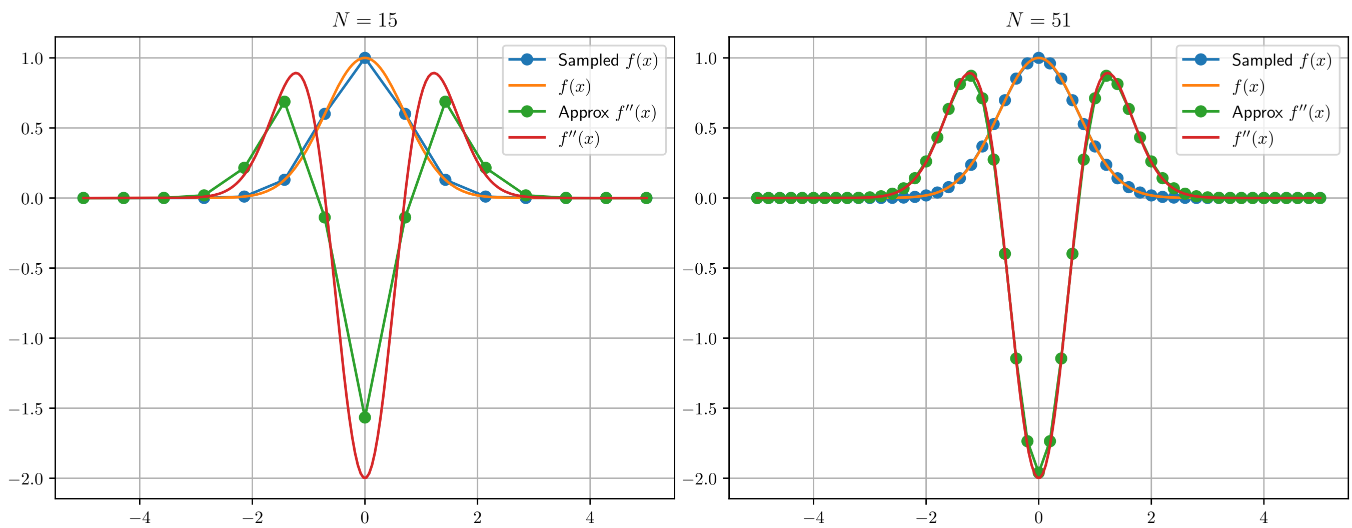


Figure 1: Finite difference approximation to a Gaussian function and its second derivative

In Figure 1, the comparison of analytical and numerical second derivative of a Gaussian function is shown. It can be seen clearly that the accuracy of numerical second derivate became better as the number of the grid points is increased.

2.3 Harmonic potential

Now that we know how to represent second derivative as a matrix, we are ready to solve the Schroedinger equation. We will start with a simple potential $V(x)$ for which we know the exact solutions, i.e. the harmonic potential:

$$V(x) = \frac{1}{2}\omega^2 x^2 \tag{11}$$

where ω is a parameter.

The Hamiltonian in the finite difference representation takes the following form:

$$\mathbb{H} = -\frac{1}{2}\mathbb{D}^{(2)} + \mathbb{V} \tag{12}$$

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

$$\mathbb{V}_{ij} = V(x_i)\delta_{ij} \tag{13}$$

and $\mathbb{D}^{(2)}$ is the second derivative matrix defined previously.

The following code calculates the harmonic potential with the default value of $\omega = 1$.

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

The following Julia snippet illustrates the steps of constructing the Hamiltonian matrix, starting from initialization of grid points, building the 2nd derivative matrix, and building the potential.

```
# Initialize the grid points
xmin = -5.0; xmax = 5.0
N = 51
x, h = init_FD1d_grid(xmin, xmax, N)
# Build 2nd derivative matrix
D2 = build_D2_matrix_3pt(N, h)
# Potential
Vpot = pot_harmonic.(x)
# Hamiltonian
Ham = -0.5*D2 + diagm( 0 => Vpot )
```

Once the Hamiltonian matrix has been constructed, we can find the solutions or the eigenvalues and eigenvectors by solving the eigenproblem. In Julia, we can do this by calling the `eigen` function of `LinearAlgebra` package which is part of the standard Julia library. The following snippets shows how this can be achieved.

```
# Solve the eigenproblem
evals, evecs = eigen( Ham )
# We will show the 5 lowest eigenvalues
Nstates = 5
@printf("Eigenvalues\n")
for i in 1:Nstates
    @printf("%5d %18.10f\n", i, evals[i])
end
```

We can compare our eigenvalues result with the analytical solution:

$$E_n = (2n-1)\frac{\hbar}{2}\omega, \quad n = 1, 2, 3, \ldots \tag{14}$$

The results are shown in the following table for $N = 51$.

n	Numerical	Exact	abs(error)
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

You may try to vary the number of N to achieve higher accuracy.

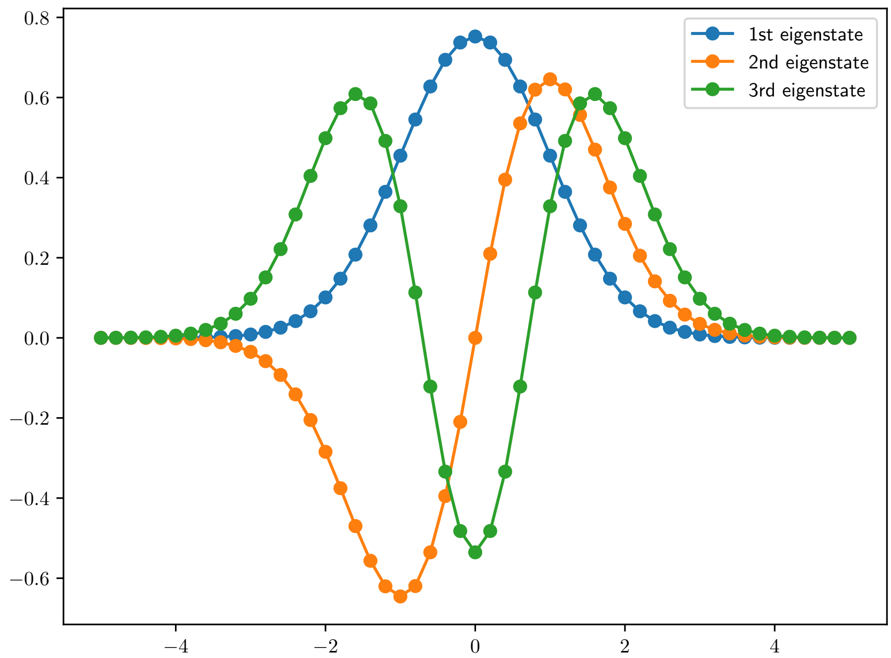


Figure 2: Eigenstates of harmonic oscillator

In addition to the eigenvalues, we also can visualize the eigenfunctions or eigenstates. The results are shown in Figure 2 for $N = 51$.

The full Julia program for this harmonic potential is given in `sch_1d/main_harmonic_01.jl`.

Note that calling `eigen` will give us N-pairs of eigenvalue-eigenfunctions where N is the dimension of the Hamiltonian matrix or in this case the number of grid points. We rarely needs all of these eigenpairs.

2.4 Higher order finite difference

To achieve more accurate result we can include more points in our calculations. However there is an alternative, namely by using more points or higher order formula to approximate second derivative An example is 5-point formula for central difference approximation to second derivative:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi_i \approx \frac{-\psi_{i+2} + 16\psi_{i+1} - 30\psi_i + 16\psi_{i-1} - \psi_{i-2}}{12h^2} \tag{15}$$

The finite difference coefficients can be found in the literature or various sources on the web (for example: `http://web.media.mit.edu/~crtaylor/calculator.html`).

We have provided Julia codes for calculating second derivative matrix using 5, 7, and 9 points with the name `build_D2_matrix_xpt.jl` where $x = 5, 7, 9$. You can repeat the calculation for harmonic oscillator potential with fixed number of N and and compare the eigenvalue results by using 3, 5, 7, and 9 points formula for second derivative matrix.

3 Schroedinger equation in 2d

Now we will turn out attention to higher dimensions, i.e 2d. The Schrodinger equation in 2d reads:

$$\left[-\frac{1}{2}\nabla^2 + V(x,y) \right] \psi(x,y) = E\,\psi(x,y) \tag{16}$$

where ∇^2 is the Laplacian operator:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{17}$$

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 $x_{\text{max}}, x_{\text{min}}$ for the X-domain and $y_{\text{max}}, y_{\text{min}}$ 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. That 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
    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}
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
fdgrid = 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

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

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

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

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

```
for ip = 1:fdgrid.Npoints
    @printf("%3d %8.3f %8.3f\n", ip, fdgrid.r[1,ip], fdgrid.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 = fdgrid.idx_xy2ip[i,j]
        @printf("[%8.3f,%8.3f] ", fdgrid.r[1,ip], fdgrid.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 16. 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, $\mathbb{D}^{(2)}_x$, y direction, $\mathbb{D}^{(2)}_x$, we can construct finite difference representation of the Laplacian operator $\mathbb{L}$ by using

$$\mathbb{L} = \mathbb{D}^{(2)}_x \otimes \mathbb{I}_y + \mathbb{I}_x \otimes \mathbb{D}^{(2)}_y \tag{18}$$

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( fdgrid::FD2dGrid; func_1d=build_D2_matrix_3pt )
    Nx = fdgrid.Nx
    hx = fdgrid.hx
    Ny = fdgrid.Ny
    hy = fdgrid.hy

    D2x = func_1d(Nx, hx)
    D2y = func_1d(Ny, hy)

    ∇² = kron(D2x, speye(Ny)) + kron(speye(Nx), D2y)
    return ∇²
end
```

In the Figure 3, an example to the approximation of 2nd derivative of 2d Gaussian function by using finite difference is shown.

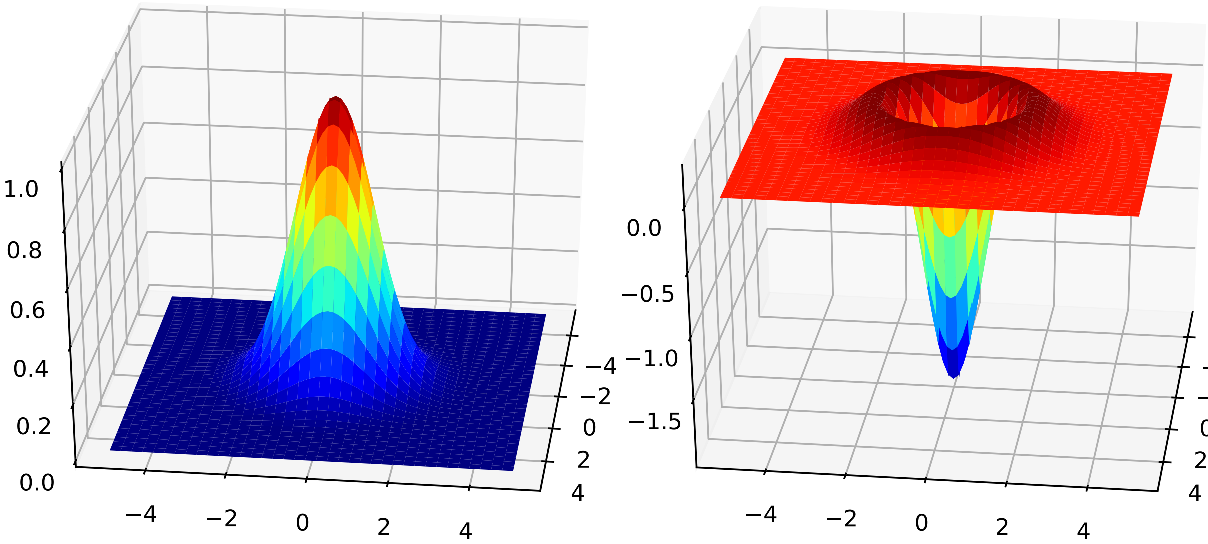


Figure 3: Two-dimensional Gaussian function and its finite difference approximation of second derivative

3.3 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( fdgrid )
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 `eigen` method 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 the eigenvalue and eigenfunction of the Hamiltonian, however it is not recommended nor practical for larger problem size.

Typically, we also do not need to solve for all eigenvalue and eigenfunction pairs. We only need to solve for several eigenpairs with lowest eigenvalues. In typical density functional theory calculations, we typically solve for $N_{\mathrm{electrons}}$ or $N_{\mathrm{electrons}}/2$ lowest states, where $N_{\mathrm{electrons}}$ 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.

In this short article, we will not discuss about these methods in depth. However, we have prepared several implementation of iterative diagonalization methods for your convenience:

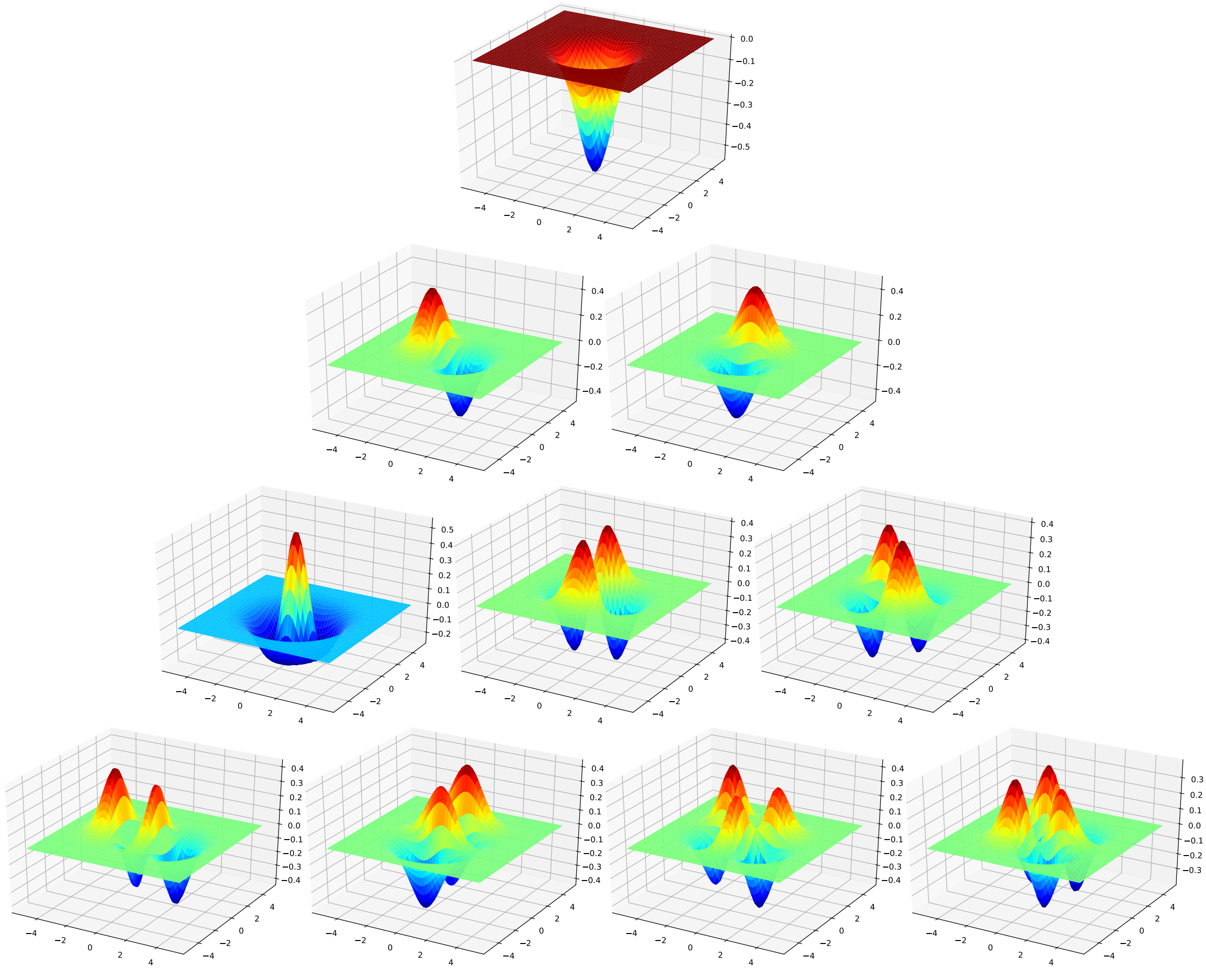
- `diag_Emin_PCG`
- `diag_davidson`
- `diag_LOBPCG`

Almost all iterative methods need a good preconditioner to function properly. In this talk, we will use several preconditioners that have been implemented in several packages in Julia such as incomplete LU and multigrid preconditioners.

Here we show an example of Julia program to solve the Schroedinger equation for two dimensional harmonic potentials. The complete program can be found in the `sch_2d/main_harmonic.jl`.

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

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

    # Preconditioner based on inverse kinetic
    prec = ilu(-0.5*∇2)

    Nstates = 10
    Npoints = Nx*Ny
    X = rand(Float64, Npoints, Nstates)
    ortho_sqrt!(X)

    evals = diag_LOBPCG!( Ham, X, prec, verbose=true )
    X = X/sqrt(fdgrid.dA) # renormalize the eigenfunctions

    @printf("\n\nEigenvalues\n")
    for i in 1:Nstates
        @printf("%5d %18.10f\n", i, evals[i])
    end
end
```

The eigenfunctions are shown in Figure 3.3.



Eigenvalues ($N_x = N_y = 50$):

```
     1        0.9999999862
     2        1.9999998768
     3        1.9999999392
     4        2.9999992436
     5        2.9999993054
     6        2.9999997845
     7        3.9999973668
     8        3.9999976649
     9        3.9999992565
    10        3.9999998030
```

Energy

```
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)
```

Energy:

$$E_{n_x+n_y} = \hbar\omega\,(n_x + n_y + 1) \tag{19}$$

4 Schroedinger equation in 3d

The 3d case of Schroedinger equation is a straightforward extension of the 2d case. The Schroedinger equation thus reads:

$$\left[-\frac{1}{2}\nabla^2 + V(x,y,z)\right]\psi(x,y,z) = E\,\psi(x,y,z) \tag{20}$$

where ∇^2 is the Laplacian operator:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{21}$$

We begin by defining a struct called `FD3dGrid` which is a straightforward generalization of `FD2dGrid`. The implementation of this struct can be found in the file `FD3d/FD3dGrid.jl`.

The Lagrangian operator in 3d also can be implemented by straightforward extension of 2d case.

```
const ⊗ = kron

function build_nabla2_matrix( fdgrid::FD3dGrid; func_1d=build_D2_matrix_3pt )

    D2x = func_1d(fdgrid.Nx, fdgrid.hx)
    D2y = func_1d(fdgrid.Ny, fdgrid.hy)
    D2z = func_1d(fdgrid.Nz, fdgrid.hz)

    IIx = speye(fdgrid.Nx)
    IIy = speye(fdgrid.Ny)
    IIz = speye(fdgrid.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.

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

Analytic solution for energy:

$$E_{n_x+n_y+n_z} = \hbar\omega\left(n_x + n_y + n_z + \frac{3}{2}\right) \tag{22}$$

Degeneracies:

$$g_n = \frac{(n+1)(n+2)}{2} \tag{23}$$

```
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
```

A Alternative solutions

```
function my_func()
    println("OK ...")
end
```