Implementing Density Functional Theory

Fadjar Fathurrahman
Hermawan Kresno Dipojono

December 24, 2019

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

Many-body (electrons-nuclei) Hamiltonian (in atomic unit):

$$\hat{\mathcal{H}} = -\sum_{I=1}^{N_a} \frac{1}{2M_I}\nabla_I^2 - \sum_{i=1}^{N_e} \frac{1}{2m}\nabla_i^2 + \sum_{I=1}^{N_a}\sum_{J\neq I}^{N_a} \frac{Z_I Z_J}{|\mathbf{R}_I - \mathbf{R}_J|} + \frac{1}{2}\sum_{i=1}^{N_e}\sum_{j\neq i}^{N_e} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} - \sum_{I=1}^{N_a}\sum_{i\neq i}^{N_e} \frac{Z_I}{|\mathbf{R}_I - \mathbf{r}_i|} \tag{1.1}$$

$\mathbf{R} = \{\mathbf{R}_I, I = 1, 2, \ldots, N_a\}$

$\mathbf{r} = \{\mathbf{r}_i, i = 1, 2, \ldots, N_e\}$

Many-body Schroedinger equation

$$\hat{\mathcal{H}}\,\Psi_n(\mathbf{R},\mathbf{r}) = \mathcal{E}_n\,\Psi_n(\mathbf{R},\mathbf{r}) \tag{1.2}$$

$$\Psi(\mathbf{R},\mathbf{r},t) = \sum_n \Theta_n(\mathbf{R},t)\Phi_n(\mathbf{R},r) \tag{1.3}$$

$\Theta_n(\mathbf{R},t)$ are wave functions describing dynamics of nuclear subsystem in each one of adiabatic electronic eigenstates $\Phi_n(\mathbf{R},r)$.

$$\hat{h}_e\,\Phi_n(\mathbf{R},\mathbf{r}) = E_n\,\Phi_n(\mathbf{R},\mathbf{r}) \tag{1.4}$$

Electronic Hamiltonian:

$$\hat{h}_e = \hat{T} + \hat{U}_{ee} + \hat{V}_{ne} = \mathcal{H} - \hat{T}_n - \hat{V}_{nn} \tag{1.5}$$

Kohn-Sham energy functional

Kohn-Sham equation [1] :

$$\left[-\frac{1}{2}\nabla^2 + V_{\text{KS}}(\mathbf{r})\right]\psi_i(\mathbf{r}) = \epsilon_i\,\psi_i(\mathbf{r}) \tag{1.6}$$

with Kohn-Sham potential:

$$V_{\text{KS}}(\mathbf{r}) = V_{\text{ion}}(\mathbf{r}) + V_{\text{Ha}}(\mathbf{r}) + V_{\text{xc}}(\mathbf{r}) \tag{1.7}$$

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

$$V_{\text{ion}}(\mathbf{r}) = \sum_{I=1} \frac{Z_I}{\mathbf{r} - \mathbf{R}_I} \tag{1.9}$$

Bibliography

[1] Test Kohn Sham

Chapter 2

Schroedinger equation in 1d

The equation we want to solve is (in Hartree atomic unit):

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

with the boundary conditions:

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

We will discretize the wave function, potentials, and various spatial quantities using regular grid within the finite computational interval $[x_{\text{min}}, x_{\text{max}}]$. This computational interval should be choosen such that the boundary condition 2.2 approximately satisfied.

We will choose the grid points x_i, $i = 1, 2, \ldots$ as:

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

where N is the number of grid points and h is the spacing between the grid points is:

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

The following 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)
    x = zeros(Float64,N)
    for i = 1:N
        x[i] = x_min + (i-1)*h
    end
    return x, h
end

init_FD1d_grid( X, N ) = init_FD1d_grid( X[1], X[2], N )
```

2.1 Approximating second derivative

With the following notation: $\psi_i = \psi(x_i)$, we can use 3-point finite difference to approximate second derivative of $\psi(x)$:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi_i = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{h^2} \tag{2.5}$$

Take $\{\psi_i\}$ as (column) vector, we can represent the second derivative operation as matrix multiplication:

$$\vec{\psi}'' = \mathbb{D}^{(2)}\vec{\psi} \tag{2.6}$$

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

$$\mathbb{D}^{(2)} = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & -2 \end{bmatrix} \tag{2.7}$$

An example implementation can be found in file `build_D2_matrix_3pt.jl`.

```julia
"""
Build second derivative matrix using 3-points centered
finite difference approximation.

# Arguments
- `N::Int64`: number of grid points
- `h::Float64`: spacing between grid points
"""
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
```

Test with Gaussian function:

$$\psi(x) = \mathrm{e}^{-\alpha x^2} \tag{2.8}$$

which second derivative can be calculated as

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

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
```

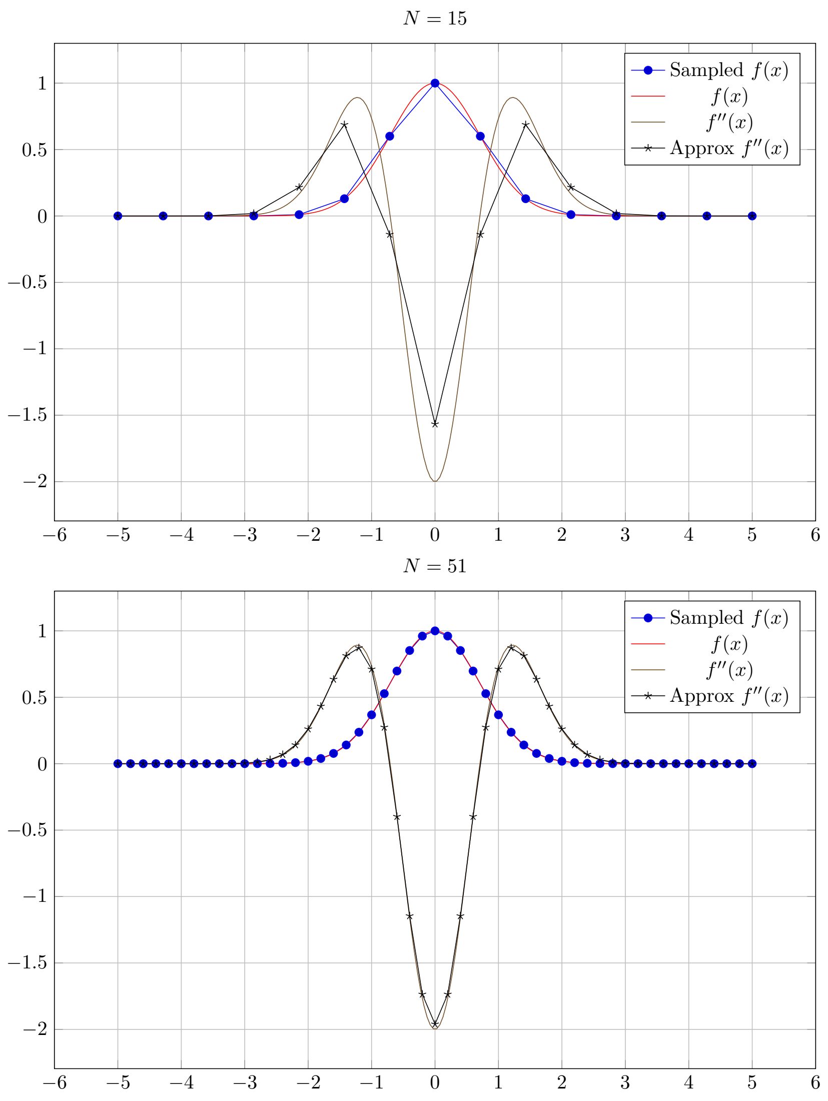

Figure 2.1: Finite difference approximation to a Gaussian function and its second derivative

2.2 Harmonic potential

We will start with a simple potential with known exact solution, namely the harmonic potential:

$$V(x) = \frac{1}{2}\omega^2 x^2 \tag{2.10}$$

The Hamiltonian in finite difference representation:

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

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

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

Code to solve harmonic oscillator:

```julia
using Printf
using LinearAlgebra
using PGFPlotsX
using LaTeXStrings

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()
    # 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 )
    # 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

    # 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
    f = @pgf Axis({ title="N="*string(N), height="10cm", width="15cm",
    ↪  xmajorgrids, ymajorgrids },
        PlotInc(Coordinates(x, evecs[:,1])),
        LegendEntry("1st eigenstate"),
        PlotInc(Coordinates(x, evecs[:,2])),
        LegendEntry("2nd eigenstate"),
        PlotInc(Coordinates(x, evecs[:,3])),
        LegendEntry("3rd eigenstate"),
    )
    pgfsave("IMG_main_harmonic_01_"*string(N)*".pdf", f)
end

main()
```

Compare with analytical solution.

Plot of eigenfunctions:

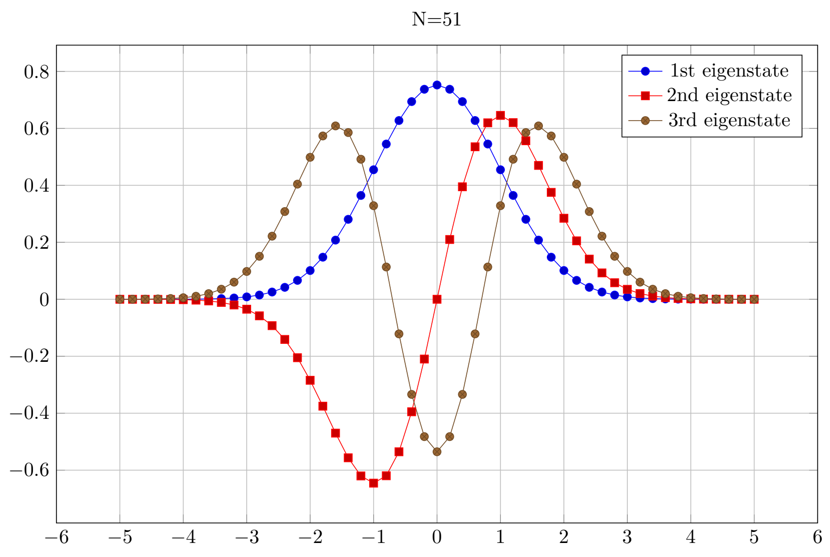

Figure 2.2: Eigenstates of harmonic oscillator

2.3 Higher order finite difference

To obtain higher accuracy

Implementing higher order finite difference.

2.4 Exercises

Gaussian potential

Chapter 3

Schroedinger equation in 2d

Schrodinger equation in 2d:

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

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 Finite difference grid in 2d

```
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

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
```

3.2 Laplacian operator

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

$$\mathbb{L} = \mathbb{D}^{(2)}_{x} \otimes \mathbb{I}_{y} + \mathbb{I}_{x} \otimes \mathbb{D}^{(2)}_{y} \tag{3.3}$$

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)`.

```
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
```

Example to the approximation of 2nd derivative of 2d Gaussian function

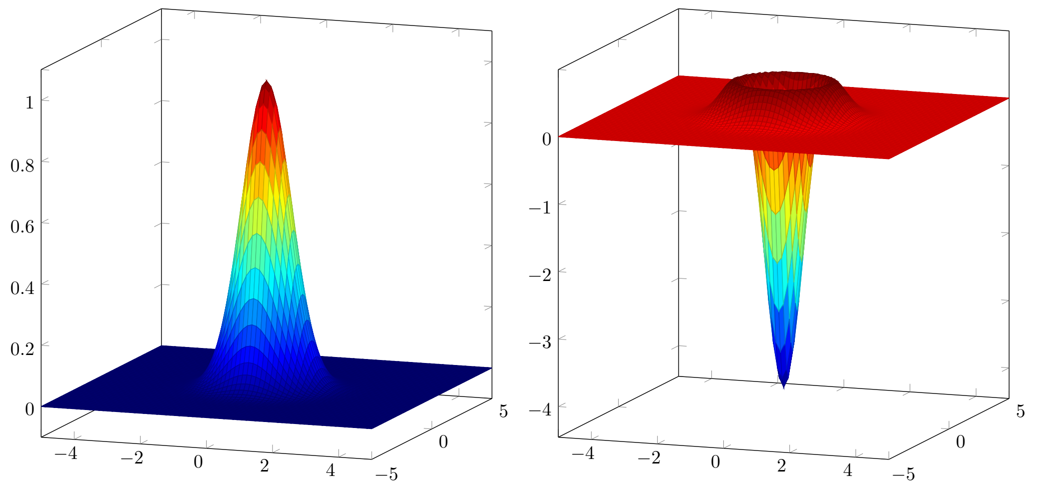

Figure 3.1: Two-dimensional Gaussian function and its finite difference approximation of second derivative

3.3 Iterative methods for eigenvalue problem

The Hamiltonian matrix:

```
∇2 = build_nabla2_matrix( fdgrid, func_1d=build_D2_matrix_9pt )
Ham = -0.5*∇2 + spdiagm( 0 => Vpot )
```

The Hamiltonian matrix size is large. The use `eigen` method to solve this eigenvalue problem is not practical. We also do not need to solve for all eigenvalues. We must resort to the so called iterative methods.

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 **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

$$\mathbb{L} = \mathbb{D}^{(2)}_{x} \otimes \mathbb{I}_{y} \otimes \mathbb{I}_{z} + \mathbb{I}_{x} \otimes \mathbb{D}^{(2)}_{y} \otimes \mathbb{I}_{z} + \mathbb{I}_{x} \otimes \mathbb{I}_{y} \otimes \mathbb{D}^{(2)}_{z} \tag{4.3}$$

Code

```
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
```

Chapter 5

Numerical solution of Poisson equation

We will turn our attention to the Poisson equation:

$$\nabla^2 V_H(\mathbf{r}) = 4\pi\rho(\mathbf{r}) \tag{5.1}$$

Introduction to conjugate gradient problem

3d dimensional problem

Chapter 6

Kohn-Sham equation part I

Using local potential only

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
```

To exit type

```
julia> exit()
```

A.2.2 Julia script file

In a text file with `.jl` extension.

You can experiment with Julia REPL by typing `julia` at terminal:

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

```
julia filename.jl
```

The following code

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

A.3 Basic programming construct

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

A.4 Mathematical operators

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

Example code 3

```
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

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
```

Index