Density Potential Functional Theory in position and momentum space and its implementation using the Machine Learning library PyTorch

A Dissertation presented to the
National University of Singapore

for the degree of Bachelor of Science (Hons.) in Physics

Ding Ruiqi

Supervisor: Prof. Berge Englert

March 2020

Abstract

This thesis studies the Density Potential Functional Theory (DPFT) introduced by Julian Schwinger and Berge Englert. Its application on spin polarized Fermi gas with magnetic dipole-dipole interaction is studied under the Thomas Fermi (TF) approximation in both position and momentum space. The main new contribution is developing **my own** orbital free (OF) DPFT code and releasing it as an official python packaged for interested users. My code used the state of the art Machine Learning library PyTorch to achieve multi-GPU acceleration. The magnetic dipole-dipole interaction has been implemented in $2D$ and $3D$, the result of which agrees well with physical expectation. The accuracy of OF-DPFT code has been compared with custom Kohn Sham DFT code. The state of the art DFT software VASP has been studied in order to adopt its strengths into my own code in the future development. Another innovation that applies a low pass filter on TF density has shown enhanced accuracy and will be further explored in the future.

Acknowledgement

I would first like to thank my supervisor Prof. Berge Englert. He guided me to explore all amazing aspects of Density Potential Functional Theory. Apart from teaching me various skills in mathematics, he also taught me to think like a physicist. I will always remember how he has patiently guided me through derivations, as well as interesting stories shared by him every time we have lunch. In earlier 2019, he has organized a conference named *Density Functionals for Many-Particle Systems: Mathematical Theory and Physical Applications of Effective Equations*, where he has invited prominent researcher in the field of DFT to deliver talks and workshops, notably Kieron Burke and Mel Levy. I was kindly invited to that conference as a guest student. Although it was hard to follow all of the content delivered by those well established professors, I was able to learn the key picture of DFT and light up my passion towards this amazing field. Most of all I thank him to allow me to participate in such profound research.

I would also like to thank all my group members for valuable discussions. In particular Dr. Matin Trappe, for he has guided me to take the first step in writing my own OFDPFT code. Under his influence, I also improved my ability to use the essential software for physicists: Mathematica. Throughout the final year, he has enlightened me time after time and provided valuable suggestions to my work. He is kind and patient, I enjoy collaborating with him and really look forward to working with him in the future code development.

I would also like to thank another two lecturer from the DFT module I took: Firstly Prof. Quek Su Ying, who gave us an interesting lecture on DFT focusing on its application in material discovery. Secondly Prof. Feng Yuan Ping, he introduced the idea of using machine learning in DFT which is a new exciting research hotspot. He was also a mentor to guide me learning the software VASP, which was crucial to help me understand the key elements of DFT.

Last but not least, I would like to thank Prof. Reiner Kree, who was my professor for the module *Advanced Topics in Biophysics* during my exchange in Georg-August-Universität Göttingen. He guided me to participate in a project where Deep reinforcement learning was used to study the motion of nanorobots in complex fluid. Without the knowledge and programming skills he taught and trained me about machine learning, this report would not be possible.

Contents

List of Figures

Chapter 1

Density functional theory

1.1 Hohenberg Kohn Theorem

We can use the standard textbook Variational Principle to find the ground state energy E_{GS} of a system:

$$E_{GS} = \min_{\Psi} \langle \Psi | H | \Psi \rangle \tag{1.1}$$

However, for a many particle system, the above many particle wave function has the form

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N) \tag{1.2}$$

Where $\boldsymbol{r}_i$ is the position of the ith electron. This is terrible for practical calculation: consider 10 electrons and a $10 \times 10 \times 10$ space grid. And suppose that a complex number takes 10 bytes of storage. Then the wave function alone takes a storage of $10^{3 \cdot 10} \cdot 10$ bytes $= 10^{22}$ GB. To save us from this disaster, the Hohenberg Kohn Variational Principle[1] states that the ground state energy is the minimum of some functional of the spatial density $n(\boldsymbol{r})$, it is natural to call this functional $E[n(\boldsymbol{r})]$:

$$E_{GS} = \min_{n} E[n(\boldsymbol{r})] \tag{1.3}$$

Now instead of the many particle wave function, we only need to find the density $n(\boldsymbol{r})$. For the same system, we now need of storage of: $10^3 \cdot 5$ bytes $= 5$ GB, where the 5 bytes is because the density is real instead of complex. More specifically, we made the following progress:

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N) \quad \text{to} \quad n(\boldsymbol{r}) \tag{1.4}$$

$$\boldsymbol{R}^{3N} \to \boldsymbol{C} \quad \text{to} \quad \boldsymbol{R}^3 \to R \tag{1.5}$$

1.2 Finding the Energy functional

The Hohenberg Kohn Theorem may sound powerful, but we do not know the exact form of $E[n]$, unlike wave functions that can be calculated using the Schrodinger equation. Therefore over the past decades, people have been trying to find approximations to the energy functional, which is naturally divided into three parts: the kinetic energy $T_{\text{kin}}[n]$, the energy $E_{\text{ext}}[n]$ from some external potential and the interaction energy $E_{\text{int}}[n]$ between particles.

$$E[n] = T_{\text{kin}}[n] + E_{\text{ext}}[n] + E_{\text{int}}[n] \tag{1.6}$$

Note that the famous exchange-correlation energy $E_{\text{xc}}[n]$ is considered as part of the interaction energy $E_{\text{int}}[n]$:

$$E_{\text{int}}[n] = E_{\text{direct}}[n] + E_{\text{xc}}[n] \tag{1.7}$$

$$E_{\text{xc}}[n] = E_{\text{exchange}}[n] + E_{\text{correlation}}[n] \tag{1.8}$$

The direct interaction energy $E_{\text{direct}}[n]$ is the part of interaction energy we know exactly such as Hartree (Coulomb) or dipole-dipole interaction. The unknown exchange-correlation energy can be modelled using the uniform electron gas, known as the local density approximation (LDA). The exchange energy for LDA is derived by Dirac,[2]

$$E_{\text{x}}^{\text{LDA}}[n] = -\frac{3}{4}(\frac{3}{\pi})^{1/3} \int d\boldsymbol{r}\ n^{4/3} \tag{1.9}$$

We will use Dirac's approach again later in a different form. In practical DFT programs, hybrid functionals that combine LDA with other approximations are used. For example, a popular one is B3LYP[3] :

$$E_{\mathrm{x}}^{\mathrm{B3LYP}} = 0.8E_{\mathrm{x}}^{\mathrm{LDA}} + 0.2E_{\mathrm{x}}^{\mathrm{HF}} + 0.72\Delta E_{\mathrm{x}}^{\mathrm{B88}} \tag{1.10}$$

$$E_{\mathrm{c}}^{\mathrm{B3LYP}} = 0.19E_{\mathrm{c}}^{\mathrm{VWN3}} + 0.81E_{\mathrm{c}}^{\mathrm{LYP}} \tag{1.11}$$

The kinetic energy functional $T[n]$ is the most important term since it usually has the same order as the total energy. Thus it has been the term that people try to approximate closely the most. In fact, it is where Orbital free formalism and Kohn Sham[4] formalism differ from each other.

1.3 Orbital free formalism

The starting point for approximation to $T[n]$ in Orbital free formalism is the Thomas-Fermi[5,6] model. Consider **uniform** electron gas, then the density in position space can be calculated by integrating the phase space density $\eta(\mu - H(\boldsymbol{p}, \boldsymbol{r}))$ in momentum space. Where η is the Heaviside unit step function and, μ is the chemical potential. The phase space density is 1 when $\mu - H(\boldsymbol{p}, \boldsymbol{r}) > 0$ and 0 otherwise. This leads to the definition of Fermi momentum p_F: the maximum magnitude of momentum $\boldsymbol{p}$ that satisfies $\mu - H(\boldsymbol{p}, \boldsymbol{r}) > 0$

$$n_{\mathrm{TF}}(\boldsymbol{r}) = 2\int d\boldsymbol{p}\ \eta(\mu - H(\boldsymbol{p}, \boldsymbol{r}))\frac{1}{(2\pi\hbar)^3} \tag{1.12}$$

$$= 2 \cdot \frac{4\pi}{3}p_F^3(\boldsymbol{r}) \cdot \frac{1}{(2\pi\hbar)^3} = \frac{1}{3\pi^2}k_F^3(\boldsymbol{r}) \tag{1.13}$$

Where the factor 2 is spin multiplicity for Fermions. Then the kinetic energy $T[n]$ can be represented using spatial density as follows:

$$T_{\mathrm{TF}}[n] = \int d\boldsymbol{r}d\boldsymbol{p}\ \frac{\boldsymbol{p}^2}{2m} = \int d\boldsymbol{r}\ 4\pi\frac{1}{2m}\frac{1}{5}p_F^5 \tag{1.14}$$

$$= \int d\boldsymbol{r}\ 4\pi\frac{1}{2m}\ \frac{1}{5}\ (\frac{3h^3}{8\pi}n)^{5/3} \equiv C_{\mathrm{TF}}\int d\boldsymbol{r}\ n^{5/3} \tag{1.15}$$

The above kinetic energy functional is exact only for a uniform (density) system such as metals. For a more general non-uniform system correction terms need to be added. Since the non-uniformity can be characterized by the gradient of the density, there exists a family of corrections known as the gradient expansions. It is worth to mention that one paper by Burke et al.[7] called *Generalized gradient approximation made simple* has been cited **112,438** times, probability the

most cited paper in the history of Physics. A simple example of the gradient expansion is the TF-Weizsacker[8] functional:

$$T_{\mathrm{TFW}}[n] = T_{\mathrm{TF}}[n] + \frac{1}{8}\int d\boldsymbol{r}\ \frac{|\nabla n|^2}{n} \tag{1.16}$$

Apart from the gradient expansions, another type of non-local correction is the convolution of density over some kernel. In the state of the art Orbital free DFT applications, the Wang-Teter[9] functional is commonly used:

$$T_{\mathrm{WT}}[n] = C_{\mathrm{TF}}\int d\boldsymbol{r}_a d\boldsymbol{r}_b\ n(\boldsymbol{r}_a)^{5/6}\ \underbrace{W(\boldsymbol{r}_a - \boldsymbol{r}_b)}_{\text{convolution kernel}}\ n(\boldsymbol{r}_b)^{5/6} \tag{1.17}$$

The main computational cost in Orbital free formalism is using the fast Fourier transform (FFT) to evaluate the kinetic energy functional and Coulomb interaction. However, conventional FFT algorithms face parallelization limitations due to core to core communication. Luckily, recently the Emily Carter[10] group designed a small-box FFT (SBFFT) algorithm that overcame this issue, achieving a state of the art result: calculating one million lithium atoms in around 2 minutes time.

1.4 Kohn Sham formalism

In Orbital free formalism, the kinetic energy functional is not accurate enough even with various corrections. We know that solving Schrodinger equations to get wave functions gives us the exact result, maybe we can somehow combine the Schrodinger approach and Density functional theory? Influenced by Hartree Fock method,[11] Kohn and Sham[4] first came up with this idea and reintroduced the use of wave functions known as Kohn Sham orbitals, the main technique of which is to convert many body wave function that depends on N particle coordinates into N wave functions of one particle coordinates:

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N) \quad \text{to} \quad \{\psi_1(\boldsymbol{r}), \psi_2(\boldsymbol{r}), ...\psi_N(\boldsymbol{r})\} \tag{1.18}$$

$$\boldsymbol{R}^{3N} \to \boldsymbol{C} \quad \text{to} \quad \boldsymbol{R}^3 \to N \times \boldsymbol{C} \tag{1.19}$$

This approach has proven to be more accurate than Orbital free formalism, but computationally more expensive for large particle systems since the cost scales with particle number. In Kohn Sham formalism the energy functional can be expressed as:

$$E[n] = T_{\text{kin}}[\psi_i(\boldsymbol{r})] + E_{\text{ext}}[n] + E_{\text{int}}[n] \tag{1.20}$$

In fact, not only the kinetic energy term, but also the interaction energy, can be improved by calculation involving the Kohn Sham orbitals. We are not going into further details here. The Kohn Sham orbitals $\{\psi_i(\boldsymbol{r})\}$ are calculated using the single particle Schrodinger equation ($\hbar = 1$):

$$\left[\frac{(i\nabla)^2}{2} + V_{\text{ext}}[n] + V_{\text{int}}[n]\right] \psi_i(\boldsymbol{r}) = \epsilon_i \psi_i(\boldsymbol{r}) \tag{1.21}$$

And from the Kohn Sham orbitals we can calculate the density:

$$n(\boldsymbol{r}) = \sum_i^N |\psi_i(\boldsymbol{r})|^2 \tag{1.22}$$

Chapter 2

Density potential functional theory

As the name suggests, in Density potential functional theory, we want to convert the total energy functional $E[n]$ from the Hohenberg Kohn Theorem, which is a functional of density only, to a functional of both density and the so-called effective potential using Legendre transformation. This idea is first introduced by Julian Schwinger,[12] and promoted by Berge Englert.[13–16] It is regarded as the most important contribution of Julian Schwinger to the field of density functional theory.

2.1 Note: notation convention

In previous works by Berge Englert, the convention is to use $n(\boldsymbol{r})$ to represent position space density, and use $\rho(\boldsymbol{p})$ to represent momentum space density. However, in this report the symbol n and ρ are interchangeably used, this is in order to emphasize the profound equivalency and connection between position and momentum density. The meanings should be easily understood from the context.

2.2 Position space formalism

As a first step we want to incorporate the conservation of particles into the formalism. Therefore it is convenient to use the Lagrange multiplier μ, which physically is the chemical potential, to add the particle number constraint to the energy functional:

$$E[n,\mu] = E_{\text{kin}} + \int d\boldsymbol{r}\; V_{\text{ext}} n + E_{\text{int}}[n] + \mu(N - \int d\boldsymbol{r}\; n) \tag{2.1}$$

The second step is to introduce a Legendre transformation. We define the effective potential V as the functional derivative of the kinetic energy functional and add the constant μ to it, so that it acts as an conjugate variable:

$$V := \mu - \frac{\delta E_{\text{kin}}}{\delta n} \tag{2.2}$$

Then the Legendre transform of E_{kin} is:

$$E_{\text{kin}}^{\text{LGD}} := E_{\text{kin}} - \int d\boldsymbol{r}\ n\ \frac{\delta E_{\text{kin}}}{\delta n} = E_{\text{kin}} + \int d\boldsymbol{r}\ n \cdot (V - \mu) \tag{2.3}$$

This means that now $E_{\text{kin}}^{\text{LGD}}$ is a functional of the conjugate variable $(V - \mu)$:

$$E_{\text{kin}}^{\text{LGD}} = E_{\text{kin}}^{\text{LGD}}[\frac{\delta E_{\text{kin}}}{\delta n}] = E_{\text{kin}}^{\text{LGD}}[V - \mu] \tag{2.4}$$

Now one just have to express the total energy $E[n]$ using $E_{\text{kin}}^{\text{LGD}}$:

$$E = E_{\text{kin}}[n] + \int d\boldsymbol{r}\ V_{\text{ext}} n + E_{\text{int}}[n] + \mu(N - \int d\boldsymbol{r}\ n) \tag{2.5}$$

$$= E_{\text{kin}}^{\text{LGD}}[V - \mu] + \int d\boldsymbol{r}\ (V_{\text{ext}} - V)n + E_{\text{int}}[n] + \mu N \tag{2.6}$$

In summary, we have successfully made the conversion:

$$E[n] \to E[n, \mu] \to E[n, \mu, \frac{\delta E_{\text{kin}}}{\delta n}] = E[n, \mu, V] \tag{2.7}$$

The variation of E with respect to the three variables n, μ, V gives

$$\frac{\delta E[n, \mu, V]}{\delta n} = 0 = V_{\text{ext}} - V + \frac{\delta E_{\text{int}}[n]}{\delta n} \tag{2.8}$$

$$\frac{\delta E[n, \mu, V]}{\delta \mu} = 0 = \frac{\delta E_{\text{kin}}^{\text{LGD}}[V - \mu]}{\delta \mu} + N \tag{2.9}$$

$$\frac{\delta E[n, \mu, V]}{\delta V} = 0 = \frac{\delta E_{\text{kin}}^{\text{LGD}}[V - \mu]}{\delta V} - n \tag{2.10}$$

2.3 Momentum space formalism

The momentum space formalism is essentially the same, which is based on Henderson's[17] proof of the momentum version of Hohenberg Kohn Theorem. The energy functional is:

$$E[n,\mu] = \int d\boldsymbol{p}\, T_{\text{kin}} n + E_{\text{ext}} + E_{\text{int}}[n] + \mu(N - \int d\boldsymbol{p}\, n) \tag{2.11}$$

In the Legendre transformation, we define the effective kinetic energy T as the functional derivative of the external potential energy functional and add the constant μ to it, so that it acts as an conjugate variable:

$$T := \mu - \frac{\delta E_{\text{ext}}}{\delta n} \tag{2.12}$$

Then the Legendre transform of E_{ext} is:

$$E_{\text{ext}}^{\text{LGD}} := E_{\text{ext}} - \int d\boldsymbol{p}\, n\, \frac{\delta E_{\text{ext}}}{\delta n} = E_{\text{ext}} + \int d\boldsymbol{p}\, n \cdot (T - \mu) \tag{2.13}$$

This means $E_{\text{ext}}^{\text{LGD}}$ is a functional of the conjugate variable $(T - \mu)$:

$$E_{\text{ext}}^{\text{LGD}} = E_{\text{ext}}^{\text{LGD}}[\frac{\delta E_{\text{ext}}}{\delta n}] = E_{\text{ext}}^{\text{LGD}}[T - \mu] \tag{2.14}$$

Now one just have to express the total energy $E[n]$ using $E_{\text{ext}}^{\text{LGD}}$:

$$E = \int d\boldsymbol{p}\, T_{\text{kin}} n + E_{\text{ext}} + E_{\text{int}}[n] + \mu(N - \int d\boldsymbol{p}\, n) \tag{2.15}$$

$$= \int d\boldsymbol{p}\, (T_{\text{kin}} - T) n + E_{\text{ext}}^{\text{LGD}} + E_{\text{int}}[n] + \mu N \tag{2.16}$$

In summary, we have successfully made the conversion:

$$E[n] \to E[n,\mu] \to E[n,\mu,\frac{\delta E_{\text{ext}}}{\delta n}] = E[n,\mu,T] \tag{2.17}$$

The variation of E with respect to the three variables n, μ, T gives

$$\frac{\delta E[n,\mu,T]}{\delta n} = 0 = T_{\text{kin}} - T + \frac{\delta E_{\text{int}}[n]}{\delta n} \tag{2.18}$$

$$\frac{\delta E[n,\mu,T]}{\delta \mu} = 0 = \frac{\delta E_{\text{ext}}^{\text{LGD}}[T-\mu]}{\delta \mu} + N \tag{2.19}$$

$$\frac{\delta E[n,\mu,T]}{\delta T} = 0 = \frac{\delta E_{\text{ext}}^{\text{LGD}}[T-\mu]}{\delta T} - n \tag{2.20}$$

2.4 The significance of DPFT

In summary, using Legendre transform we turned the total energy functional of density only to a functional of both density and the **conjugate variable** of kinetic energy / external potential energy.

- In position space, we take the Legendre transform of the kinetic energy functional and call it the effective potential.
- In momentum space, we take the Legendre transform of the external potential energy functional and call it the effective kinetic energy.

The significance of DPFT is that, for a long time people have been trying to find approximations of various energies and express them in terms of density. For example the Thomas-Fermi kinetic energy

$$T_{\text{TF}}[n] = C_{\text{TF}} \int d\boldsymbol{r}\; n^{5/3} \tag{2.21}$$

These approximations are hard to find. But in DPFT this may not be a problem, since density and potential have equal footing now, we can express density in terms various energies instead:

$$n(\boldsymbol{r}) = -\frac{\delta E_{\text{kin}}^{\text{LGD}}[V-\mu]}{\delta V(\boldsymbol{r})} \tag{2.22}$$

$$n(\boldsymbol{p}) = -\frac{\delta E_{\text{ext}}^{\text{LGD}}[T-\mu]}{\delta(\boldsymbol{p})} \tag{2.23}$$

Various explorations in this direction have been made by my group.[18,19] The most recent one of which[19] uses Suzuki-Trotter approximation.[20] The main idea is to start with Kohn Sham density

$n(\boldsymbol{r}) = 2\langle \boldsymbol{r}|\eta(\mu - H(\boldsymbol{P},\boldsymbol{R}))|\boldsymbol{r}\rangle$, where $\eta()$ is the Heaviside step function, μ is the chemical potential, $H(\boldsymbol{P},\boldsymbol{R})$ is the single particle Hamiltonian and $|\boldsymbol{r}\rangle$ is the eigenvector of position operator $\boldsymbol{R}$. We can rewrite the density using Fourier transform:

$$n(\boldsymbol{r}) = 2\langle \boldsymbol{r}| \int_{-\infty}^{\infty} \frac{ds}{2\pi} e^{is[\mu - H(\boldsymbol{P},\boldsymbol{R})]} \tilde{\eta}(s) |\boldsymbol{r}\rangle \tag{2.24}$$

$$= 2\int_{-\infty}^{\infty} \frac{ds}{2\pi} e^{is\mu} \tilde{\eta}(s) \langle \boldsymbol{r}| \underbrace{e^{-isH(\boldsymbol{P},\boldsymbol{R})}}_{U(s)} |\boldsymbol{r}\rangle \tag{2.25}$$

Thus the problem of finding approximation to $n(\boldsymbol{r})$ has been converted to finding the approximation to the unitary evolution operator $U(s)$, in which the Suzuki-Trotter approximation can be applied.

Chapter 3

Spin polarized Fermi gas with magnetic dipole-dipole interaction

In this chapter we will apply the Density Potential functional theory and derive the various energy functionals for a specific system, namely spin polarized Fermi gas with magnetic dipole-dipole interaction. Two approximations are used: Dirac[2]'s approximation and Thomas Fermi approximation. We begin by introducing the Wigner function, which can be used to derive various density matrices.

3.1 Wigner function and densities

We denoted the momentum space one-particle density by $\rho(\boldsymbol{p})$, it is normalized to the total particle number $N = \int d\boldsymbol{p}\ \rho(\boldsymbol{p})$. Consider the one particle Wigner function $\nu(\boldsymbol{r}, \boldsymbol{p})$ without a specific form at the moment, which will later be approximated by the Thomas-Fermi model. We can use the Wigner function to express various density variables:

- Spatial one particle density matrix. In $n^{(1)}$ below, we use the superscript $^{(1)}$ to denote one particle. In $\boldsymbol{r}_1$ below, we use subscript $_1$ to denote position 1 of the particle:

$$n^{(1)}(\boldsymbol{r}_1; \boldsymbol{r}_2) = \int \frac{d\boldsymbol{p}}{(2\pi\hbar)^3}\ \nu(\frac{\boldsymbol{r}_1 + \boldsymbol{r}_2}{2}, \boldsymbol{p}) e^{i\frac{\boldsymbol{p}}{\hbar}\cdot(\boldsymbol{r}_1 - \boldsymbol{r}_2)} \tag{3.1}$$

- Spatial one particle density.

$$n(\boldsymbol{r}) = n^{(1)}(\boldsymbol{r}; \boldsymbol{r}) = \int \frac{d\boldsymbol{p}}{(2\pi\hbar)^3}\ \nu(\boldsymbol{r}, \boldsymbol{p}) \tag{3.2}$$

- Momentum space one particle density.

$$\rho(\boldsymbol{p}) = \int \frac{d\boldsymbol{r}}{(2\pi\hbar)^3}\, \nu(\boldsymbol{r},\boldsymbol{p}) \tag{3.3}$$

3.2 Dirac's approximation for two particle density matrix

The two particle density matrix can be written down using an approximation introduced by Dirac.[2] In $\boldsymbol{r}_{1a}$ below, we use subscripts $_{1a}$ to denote position 1 of particle (a) :

$$n^{(2)}(\boldsymbol{r}_{1a},\boldsymbol{r}_{1b};\boldsymbol{r}_{2a},\boldsymbol{r}_{2b}) = n^{(1)}(\boldsymbol{r}_{1a};\boldsymbol{r}_{2a})\; n^{(1)}(\boldsymbol{r}_{1b};\boldsymbol{r}_{2b}) - n^{(1)}(\boldsymbol{r}_{1a};\boldsymbol{r}_{1b})\; n^{(1)}(\boldsymbol{r}_{2b};\boldsymbol{r}_{2a}) \tag{3.4}$$

Quite intuitively, the above expression accounts for the effect of exchanging positions of the two particles. For our system, particle (a) can only be at one place, thus $\boldsymbol{r}_{1a} = \boldsymbol{r}_{2a}$ and we can drop the number subscripts $1, 2$ that represents position. The resulting two terms can be interpreted as so called direct term and exchange term:

$$n^{(2)}(\boldsymbol{r}_a,\boldsymbol{r}_b;\boldsymbol{r}_a,\boldsymbol{r}_b) = n^{(1)}(\boldsymbol{r}_a;\boldsymbol{r}_a)\; n^{(1)}(\boldsymbol{r}_b;\boldsymbol{r}_b) - n^{(1)}(\boldsymbol{r}_a;\boldsymbol{r}_b)\; n^{(1)}(\boldsymbol{r}_b;\boldsymbol{r}_a) \tag{3.5}$$

$$= \underbrace{n(\boldsymbol{r}_a)\; n(\boldsymbol{r}_b)}_{\text{direct term}} - \underbrace{n^{(1)}(\boldsymbol{r}_a;\boldsymbol{r}_b)\; n^{(1)}(\boldsymbol{r}_b;\boldsymbol{r}_a)}_{\text{exchange term}} \tag{3.6}$$

3.3 Thomas Fermi approximation for Wigner function

The Wigner function can be approximated by the Thomas Fermi approximation, which is the simplest approximation used in orbital free DFT, but it serves as the basis for further modifications. The following expressions are essentially the same as equation [1.13], except we have used a spin multiplicity of 1 instead of 2, since we will be discussing about **spin-polarized** Fermions. For position space, we have:

$$\nu(\boldsymbol{r},\boldsymbol{p}) = \eta(\hbar[6\pi^2 n(\boldsymbol{r})]^{1/3} - p) \tag{3.7}$$

Where η is the Heaviside unit step function and we will be using $p \equiv |\boldsymbol{p}|$. In analogy, we have in momentum space:

$$\nu(\boldsymbol{r},\boldsymbol{p}) = \eta(\hbar[6\pi^2 \rho(\boldsymbol{p})]^{1/3} - r) \tag{3.8}$$

3.4 Energy functionals in momentum space

3.4.1 Kinetic energy

Unlike in position space, the kinetic energy can be easily expressed in momentum space:

$$E_{\text{kin}} = \int d\boldsymbol{p}\ \frac{\boldsymbol{p}^2}{2M}\rho(\boldsymbol{p}) \tag{3.9}$$

Using the following formula for functional derivative originated from Euler–Lagrange equation

$$F[n] = \int f(\boldsymbol{r}, n(\boldsymbol{r}), \nabla n(\boldsymbol{r}))\ d\boldsymbol{r} \tag{3.10}$$

$$\Rightarrow \frac{\delta F}{\delta n} = \frac{\partial f}{\partial n} - \nabla \cdot \frac{\partial f}{\partial \nabla n} \tag{3.11}$$

Taking the functional derivative, we have:

$$\frac{\delta E_{\text{kin}}}{\delta \rho} = \frac{\boldsymbol{p}^2}{2M} \tag{3.12}$$

3.4.2 External potential energy

For external potential energy, we use the favourite harmonic trap:

$$E_{\text{ext}} = \int d\boldsymbol{r}\ \frac{1}{2}M\omega^2 r^2 n(\boldsymbol{r}) \tag{3.13}$$

$$= \int d\boldsymbol{r}\ \frac{1}{2}M\omega^2 r^2 \int \frac{d\boldsymbol{p}}{(2\pi\hbar)^3}\ \eta(\hbar[6\pi^2\rho(\boldsymbol{p})]^{1/3} - r) \tag{3.14}$$

$$= \int \frac{d\boldsymbol{p}}{(2\pi\hbar)^3}\ \frac{1}{2}M\omega^2 \int_0^{\hbar[6\pi^2\rho(\boldsymbol{p})]^{1/3}} r^2\ r^2 \sin\theta dr d\theta d\phi \tag{3.15}$$

$$= \int \frac{d\boldsymbol{p}}{(2\pi\hbar)^3}\ \frac{1}{2}M\omega^2\ 4\pi\frac{1}{5}\hbar^5[6\pi^2\rho(\boldsymbol{p})]^{5/3} \tag{3.16}$$

$$= \int \frac{d\boldsymbol{p}}{20\pi^2}\ M(\hbar\omega)^2[6\pi^2\rho(\boldsymbol{p})]^{5/3} \tag{3.17}$$

Taking the functional derivative, we have:

$$\frac{\delta E_{\text{ext}}}{\delta \rho} = \frac{1}{20\pi^2}\ M(\hbar\omega)^2\frac{5}{3}[6\pi^2\rho(\boldsymbol{p})]^{2/3}6\pi^2 \tag{3.18}$$

$$= \frac{1}{2}M(\hbar\omega)^2[6\pi^2\rho(\boldsymbol{p})]^{2/3} \tag{3.19}$$

Using this we can express the density in terms of the effective kinetic potential $T := \mu - \frac{\delta E_{\text{ext}}}{\delta \rho}$:

$$\rho(\boldsymbol{p}) = \frac{1}{6\pi^2}\left[\frac{2(\mu - T)}{M(\hbar\omega)^2}\right]^{3/2} \tag{3.20}$$

3.4.3 Dipole-dipole interaction energy

Next, we consider the dipole-dipole interaction energy E_{dd}:

$$E_{\text{dd}} = \frac{1}{2}\int d\boldsymbol{r}_a d\boldsymbol{r}_b \; U_{\text{dd}}(\boldsymbol{r}_a - \boldsymbol{r}_b) n^{(2)}(\boldsymbol{r}_a, \boldsymbol{r}_b; \boldsymbol{r}_a, \boldsymbol{r}_b) \tag{3.21}$$

Where

$$U_{\text{dd}}(\boldsymbol{r}) = \frac{\mu_0}{4\pi}\left[\frac{\boldsymbol{\mu}^2}{r^3} - 3\frac{(\boldsymbol{\mu}\cdot\boldsymbol{r})^2}{r^5} - \frac{8\pi}{3}\boldsymbol{\mu}^2\delta(\boldsymbol{r})\right] \tag{3.22}$$

Using $\nabla\nabla\frac{1}{r} = \frac{3}{r^5}\boldsymbol{r}\boldsymbol{r} - \frac{1}{r^3} - \frac{4\pi}{3}I\delta(\boldsymbol{r})$ we can rewrite:

$$U_{\text{dd}}(\boldsymbol{r}) = \frac{\mu_0}{4\pi}\boldsymbol{\mu}\cdot\left[-\nabla\nabla\frac{1}{r} - 4\pi I\delta(\boldsymbol{r})\right]\cdot\boldsymbol{\mu} \tag{3.23}$$

We want the above expression since it can be Fourier transformed easily to get the momentum space representation:

$$U_{\text{dd}}(\boldsymbol{k}) = \frac{\mu_0}{4\pi}\boldsymbol{\mu}\cdot\left[-(i\boldsymbol{k})(i\boldsymbol{k})\frac{4\pi}{k^2} - 4\pi I\right]\cdot\boldsymbol{\mu} \tag{3.24}$$

The exchange term

For the exchange term, we have:

$$E^{\text{ex}}_{\text{dd}} = -\frac{1}{2}\int d\boldsymbol{r}_a d\boldsymbol{r}_b \; U_{\text{dd}}(\boldsymbol{r}_a - \boldsymbol{r}_b) n^{(1)}(\boldsymbol{r}_a; \boldsymbol{r}_b)\; n^{(1)}(\boldsymbol{r}_b; \boldsymbol{r}_a) \tag{3.25}$$

$$= -\frac{1}{2}\int d\boldsymbol{r}_m d\boldsymbol{s} \; U_{\text{dd}}(\boldsymbol{s}) n^{(1)}(\boldsymbol{r}_m + \frac{\boldsymbol{s}}{2}; \boldsymbol{r}_m - \frac{\boldsymbol{s}}{2})\; n^{(1)}(\boldsymbol{r}_m - \frac{\boldsymbol{s}}{2}; \boldsymbol{r}_m + \frac{\boldsymbol{s}}{2}) \tag{3.26}$$

$$= -\frac{1}{2}\int d\boldsymbol{r}_m d\boldsymbol{s} \; U_{\text{dd}}(\boldsymbol{s}) \int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^6}\; \nu(\boldsymbol{r}, \boldsymbol{p}_a)\nu(\boldsymbol{r}, \boldsymbol{p}_b) e^{i\frac{\boldsymbol{p}_a - \boldsymbol{p}_b}{\hbar}\cdot\boldsymbol{s}} \tag{3.27}$$

Under Thomas Fermi approximation, the term $\nu(\boldsymbol{r}, \boldsymbol{p}_a)\nu(\boldsymbol{r}, \boldsymbol{p}_b)$ only depends on the magnitudes $|\boldsymbol{p}_a|, |\boldsymbol{p}|_2$. And since $\boldsymbol{p}_a - \boldsymbol{p}_b$ takes value in the entire space, we can conclude that the integral only

depends on the magnitude $|\boldsymbol{s}|$. Thus we can use $U_{\rm dd}(\boldsymbol{s}) = \frac{\mu_0}{4\pi}\left[-\frac{8\pi}{3}\boldsymbol{\mu}^2\delta(\boldsymbol{s})\right]$ since the first two terms in $U_{\rm dd}(\boldsymbol{s})$ vanish upon integration over the solid angle. Therefore we have in position space:

$$E_{\rm dd}^{\rm ex} = -\frac{1}{2}\int d\boldsymbol{r}_m \frac{\mu_0}{4\pi}\left[-\frac{8\pi}{3}\boldsymbol{\mu}^2\right] n^2(\boldsymbol{r}_m) \tag{3.28}$$

And in momentum space with Thomas Fermi approximation again:

$$E_{\rm dd}^{\rm ex} = -\frac{1}{2}\int d\boldsymbol{r}_m \frac{\mu_0}{4\pi}\left[-\frac{8\pi}{3}\boldsymbol{\mu}^2\right]\int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^6}\,\nu(\boldsymbol{r},\boldsymbol{p}_a)\nu(\boldsymbol{r},\boldsymbol{p}_b) \tag{3.29}$$

$$= -\frac{1}{2}\int d\boldsymbol{r}_m \frac{\mu_0}{4\pi}\left[-\frac{8\pi}{3}\boldsymbol{\mu}^2\right]\int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^6}\,\eta(\hbar[6\pi^2\rho(\boldsymbol{p}_a)]^{1/3} - r)\eta(\hbar[6\pi^2\rho(\boldsymbol{p}_b)]^{1/3} - r) \tag{3.30}$$

$$= -\frac{1}{2}\frac{\mu_0}{4\pi}\left[-\frac{8\pi}{3}\boldsymbol{\mu}^2\right]\int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^6}\,\frac{4\pi}{3}r_m^3\,\big|_{r_m=\hbar[6\pi^2\rho_<]^{1/3}} \tag{3.31}$$

$$= -\frac{1}{2}\frac{\mu_0}{4\pi}\left[-\frac{8\pi}{3}\boldsymbol{\mu}^2\right]\int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^3}\,\rho_< \tag{3.32}$$

One immediately realizes that **this exchange term has the effect of canceling out with the third delta function term in equation [3.22]** as we shall see in equations [3.37, 3.38]. And $\rho_<$ is defined as:

$$\rho_< = \min\{\rho(\boldsymbol{p}_a), \rho(\boldsymbol{p}_b)\} \tag{3.33}$$

The direct term

The direct term can be evaluated similarly using the above mentioned Fourier transform of $U_{\rm dd}$:

$$E_{\rm dd}^{\rm dir} = \frac{1}{2}\int d\boldsymbol{r}_a d\boldsymbol{r}_b\, U_{\rm dd}(\boldsymbol{r}_a - \boldsymbol{r}_b) n(\boldsymbol{r}_a)\, n(\boldsymbol{r}_b) \tag{3.34}$$

$$= \frac{1}{2}\int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^3}\,\frac{\mu_0}{4\pi}\left[(\boldsymbol{\mu}\cdot\boldsymbol{k})^2\frac{4\pi}{k^2} - 4\pi\boldsymbol{\mu}^2\right]_{k=\frac{\boldsymbol{p}_a-\boldsymbol{p}_b}{\hbar}}\,\rho_< \tag{3.35}$$

Combining the exchange term and direct term

Combining the exchange term and direct term for the interaction energy we have:

$$E_{\rm dd} = E_{\rm dd}^{\rm dir} + E_{\rm dd}^{\rm ex} = \frac{\mu_0}{2}\int \frac{d\boldsymbol{p}_a d\boldsymbol{p}_b}{(2\pi\hbar)^3}\,\left[\frac{(\boldsymbol{\mu}\cdot\boldsymbol{k})^2}{k^2} - \frac{1}{3}\boldsymbol{\mu}^2\right]_{k=\frac{\boldsymbol{p}_a-\boldsymbol{p}_b}{\hbar}}\,\rho_< \tag{3.36}$$

Taking the functional derivative, we have the dipole-dipole interaction potential in momentum space:

$$V_{\text{dd}}^{\text{mom}} := \frac{\delta E_{\text{dd}}}{\delta \rho(\boldsymbol{p}_a)} = \frac{\mu_0}{2} \int \frac{d\boldsymbol{p}_b}{(2\pi\hbar)^3} \left[\frac{(\boldsymbol{\mu} \cdot \boldsymbol{k})^2}{\boldsymbol{k}^2} - \frac{1}{3}\boldsymbol{\mu}^2 \right]_{\boldsymbol{k}=\frac{\boldsymbol{p}_a - \boldsymbol{p}_b}{\hbar}} \eta[\rho(\boldsymbol{p}_b) - \rho(\boldsymbol{p}_a)] \tag{3.37}$$

Note that we have used the notation U_{dd} for the actual potential and V_{dd} that is defined as the functional derivative of the energy functional. Again since we know that **the exchange term has the effect of canceling out with the third delta function term in equation [3.22]**. We can write down the position space version easily:

$$V_{\text{dd}}^{\text{pos}} := \frac{\delta E_{\text{dd}}}{\delta \rho(\boldsymbol{r}_a)} = \frac{\mu_0}{4\pi \boldsymbol{r}^3} \int d\boldsymbol{r}_b \left[-3\frac{(\boldsymbol{\mu} \cdot \boldsymbol{r})^2}{\boldsymbol{r}^2} + \boldsymbol{\mu}^2 \right]_{\boldsymbol{r}=\boldsymbol{r}_a - \boldsymbol{r}_b} \tag{3.38}$$

3.5 The integral equation

By demanding the functional derivative of the total energy functional with respect to density to vanish: $\frac{\delta E}{\delta \rho} = 0$, we can arrive at the following integral equation:

$$0 = -\mu + \frac{\boldsymbol{p}^2}{2M} + \frac{1}{2} M(\hbar\omega)^2 [6\pi^2 \rho(\boldsymbol{p})]^{2/3} + \tag{3.39}$$

$$\frac{\mu_0 \boldsymbol{\mu}^2}{2} \int \frac{d\boldsymbol{k}}{(2\pi)^3} \left[\frac{(\hat{\boldsymbol{\mu}} \cdot \boldsymbol{k})^2}{k^2} - \frac{1}{3} \right] \eta[\rho(\boldsymbol{p} - \hbar\boldsymbol{k}) - \rho(\boldsymbol{p})] \tag{3.40}$$

We want to first attempt solving this equation analytically. But the step function part $\eta[\rho(\boldsymbol{p} - \hbar\boldsymbol{k}) - \rho(\boldsymbol{p})]$ is hard to deal with and many attempts have failed, thus I had to let it be 1 by unwillingly making the **assumption** that the density $\rho(\boldsymbol{p})$ is **isotropic**. Together we introduce the substitution $\boldsymbol{p} - \hbar\boldsymbol{k} = \boldsymbol{q}$ so the integral term $V_{\text{dd}}^{\text{mom}}$ becomes

$$V_{\text{dd}}^{\text{mom,iso}} = -\frac{1}{(2\pi\hbar)^3} \int_{|\boldsymbol{q}|<|\boldsymbol{p}|} d\boldsymbol{q} \left[\frac{[\hat{\boldsymbol{\mu}} \cdot (\boldsymbol{p} - \boldsymbol{q})]^2}{(\boldsymbol{p} - \boldsymbol{q})^2} - \frac{1}{3} \right] \tag{3.41}$$

Note that $\boldsymbol{p}$ and $\hat{\boldsymbol{\mu}}$ are constants in the integration. Therefore without loss of generalization, in a (x, y, z) coordinates system we can choose $\boldsymbol{p}$ to be along z axis and $\hat{\boldsymbol{\mu}}$ to be in the x-z plane. Let the angle between $\boldsymbol{p}$ and $\hat{\boldsymbol{\mu}}$ to be α we have:

$$\boldsymbol{p} = \Big(0, 0, p\Big) \quad \hat{\boldsymbol{\mu}} = \Big(\sin\alpha, 0, \cos\alpha\Big) \quad \boldsymbol{q} = q\Big(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta\Big) \tag{3.42}$$

This integration can be solved using Mathematica:

```
"   pVec = {0, 0, p};
    muHat = {Sin[alpha], 0, Cos[alpha]};
    qVec = q * {Sin[theta]*Cos[phi], Sin[theta]*Sin[phi], Cos[theta]};
    pMinusQ = pVec - qVec;
    f := q^2 * Sin[theta] * ((muHat.pMinusQ)^2 / pMinusQ.pMinusQ - 1/3)
```

And we get the following output:

$$\frac{1}{9}\pi p^3(3\cos(2\alpha)+1) \tag{3.43}$$

Thus the density under the isotropic assumption is:

$$\rho(\boldsymbol{p}) = \frac{1}{6\pi^2}\left(\frac{1}{2}M(\hbar\omega)^2\right)^{-3/2}\left[\underbrace{\mu}_{V_{\text{chem}}}\underbrace{-\frac{\boldsymbol{p}^2}{2M}}_{V_{\text{kin}}}\underbrace{+\frac{\mu_0\boldsymbol{\mu}^2p^3}{144\,\hbar^3\pi^2}(3\cos(2\alpha)+1)}_{V_{\text{dd}}^{\text{mom,iso}}}\right]^{3/2} \tag{3.44}$$

The term $V_{\text{dd}}^{\text{mom,iso}}$ which is the result of integrating equation [3.41] after making the isotropic assumption, has the familiar look of the spherical harmonic Y_2^0:

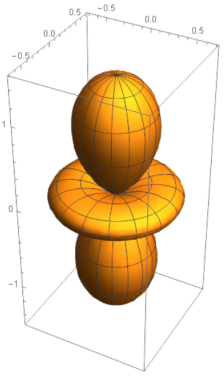


Figure 3.1: $V_{\text{dd}}^{\text{mom,iso}}$ has the familiar look of the spherical harmonic Y_2^0

We note that the density $\rho(\boldsymbol{p})$ not only depends on magnitude p of $\boldsymbol{p}$, but also the angle α, i.e., the direction of $\boldsymbol{p}$. This **contradicts** with the previous isotropic assumption, thus we regard this as a proof by contradiction and conclude: **there exists no isotropic solution $\rho(\boldsymbol{p})$ to the integral equation**. Hence this system will have be studied numerically using my own DPFT code in the next chapter.

Chapter 4

My own DPFT code development

Throughout the past few years Dr. Martin Trappe has been working on the development of DPFT code for our group. His code has been used by all of our group members in their works. However, I set up the goal to develop my own code. I have had several experiences writing code for machine learning. In the field of machine learning there have been huge efforts made by numerous prominent computer scientists in the development of parallelization. Therefore it would be wise to use a package that provides GPU/TPU acceleration out of the box. I have chosen PyTorch which is the state of the art library developed by Facebook.

4.1 Kohn Sham vs DPFT-TF in 1D

As a first step I implemented Kohn Sham DFT and orbital free DPFT code with Thomas Fermi approximation in 1D position space. This is in order to get the simplest form of a correctly working code, and the Kohn Sham code is written to verify such correctness. As mentioned earlier the Kohn Sham DFT and orbital free DPFT mainly differ from the calculation of density. Whereas the *self consistent loop* and *interaction energy* are the same for both. Therefore we first discuss the implementation of these two and compare the density implementations later.

4.1.1 Both: the self consistent loop

The self consistent (SC) loop is the classical approach used in DFT. As the name suggests, we iteratively calculate the interaction energy and density from one another until the density converges. More specifically, first we set the initial density to zeros, this is equivalent to setting a zero initial

interaction potential. We then use the total potential to calculate the new density. Finally we mix the new density and the old one and use it to calculate the new interaction potential. The loop repeats as such. The implementation in Python is as follows:

```
def forward(self,Vext,method): # self consistent loop
  l = self.config['loop']
  rho = np.zeros_like(self.x)
  for i in range(l['Imax']):
    Vx,Vh = getVint(self,rho)
    oldRho = rho
    if method == 'KS':
      rho,N,E,psi = getRhoKS(self,Vx+Vh+Vext)
    elif method == 'DPFT-TF':
      rho,N,E,psi = getRhoDPFT(self,Vx+Vh+Vext)
    if np.mean(np.abs(oldRho-rho)) < l['precision']:
      print('forward break at {}'.format(i)); break
  return Vx,Vh,rho,N,E,psi
```

4.1.2 Both: the interaction energy

For interaction energy calculation, we only consider two terms out of simplicity. They are the exchange energy under local density approximation and the Hartree (Coulomb) energy. We are considering Hartree interaction instead of dipole-dipole interaction because the 1D code is just to test out the idea and get a properly working prototype code. The dipole-dipole interactions are implemented in 2D and 3D where the concept of magnetic dipole vector $\boldsymbol{\mu}$ actually makes sense.

$$E_H[n(\boldsymbol{r})] = \frac{1}{2}\int d\boldsymbol{r}d\boldsymbol{r}'\ \frac{n(\boldsymbol{r})n(\boldsymbol{r}')}{|\boldsymbol{r}-\boldsymbol{r}'|+\epsilon} \tag{4.1}$$

$$E_{\mathrm{x}}^{\mathrm{LDA}}[n] = -\frac{3}{4}(\frac{3}{\pi})^{1/3}\int d\boldsymbol{r}\ n^{4/3} \tag{4.2}$$

Where the small quantity ϵ has been added to the $\frac{1}{r}$ Hartree energy to avoid infinity near the origin. Note that the Hartree energy and in general any interaction energy has the form of correlation $f(\boldsymbol{r},\boldsymbol{r}')$ that involves two coordinates. They can be implemented easily in python. In

the following **x** is a 1D array (vector). And we have used **x[None,:]** and **x[:,None]** to represent $\boldsymbol{r}$ and $\boldsymbol{r}'$ respectively. The keyword **None** represents expanding an extra dimension, thus the operations between **x[None,:]** and **x[:,None]** are automatically **broadcasted** and form a 2D array (matrix).

```
def getVint(o,rho):
  Vx = — (3/np.pi)**(1/3) * rho**(1/3)
  Vh = np.sum(rho[None,:]/np.sqrt((o.x[None,:]—o.x[:,None])**2+o.dx),
              axis=—1) * o.dx
  return Vx,Vh
```

We demonstrate the correctness of the above method by comparing it with a **for-loop** in 3D. Since the **for-loop** goes through each element of the array, we know that it definitely produces the correct result. Note that we do not want to use the **for-loop** for production code since it cannot be accelerated in Python, whereas the **array operation: x[None,:]** is accelerated by Numpy/PyTorch automatically. The code below produces **True** and justifies the viability.

```
def forloopInt(n): # we know that the for—loop is definitely correct
  x = np.linspace(—0.5,0.5,n)
  I = np.zeros([n,n,n,n,n,n])
  for i in range(n):
    for j in range(n):
      for k in range(n):
        for a in range(n):
          for b in range(n):
            for c in range(n):
              I[i][j][k][a][b][c] = (x[i]—x[a])**2+(x[j]—x[b])**2+(x[k]—x[c])**2
  return I


def arrayInt(n):
  x = np.linspace(—0.5,0.5,n)
  xx,yy,zz = np.meshgrid(x,x,x,sparse=True,indexing='ij')
  I = (xx[None,None,None,:,:,:]—xx[:,:,:,None,None,None])**2 \
    + (yy[None,None,None,:,:,:]—yy[:,:,:,None,None,None])**2 \
```

```
    + (zz[None,None,None,:,:,:]-zz[:,:,:,None,None,None])**2
  return I

print(np.all(forloopInt(10)==arrayInt(10)))  # output: True
```

4.1.3 Kohn Sham: the Laplacian matrix

In the Kohn Sham code I used the idea of representing the Laplacian as a matrix. The potential energy is represented in diagonal matrix and added to the Laplacian matrix, this forms the Hamiltonian whose eigenvectors are the wave functions we seek: $\left[\frac{(i\nabla)^2}{2} + V_{\text{ext}}[n] + V_{\text{int}}[n]\right]\psi_i(\boldsymbol{r}) = \epsilon_i\psi_i(\boldsymbol{r})$. The first step would be to implement the Laplacian ∇^2, which can be discretized into a matrix M as follows in $1D$:

$$\nabla^2\psi = \frac{d^2\psi}{dx^2} = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{h^2} \equiv M\psi \tag{4.3}$$

where h is the grid size. And in $3D$ we have:

$$\nabla^2\psi = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} + \frac{\partial^2\psi}{\partial z^2} = \tag{4.4}$$

$$\frac{\psi_{i+1,j,k} + \psi_{i-1,j,k} + \psi_{i,j+1,k} + \psi_{i,j-1,k} + \psi_{i,j,k+1} + \psi_{i,j,k-1} - 6\psi_{i,j,k}}{h^2} \equiv M\psi \tag{4.5}$$

For a $2 \times 2 \times 2$ grid, this looks like:

$$M\psi = \begin{pmatrix} -6 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -6 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -6 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & -6 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -6 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & -6 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & -6 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & -6 \end{pmatrix} \begin{pmatrix} \psi_{1,1,1} \\ \psi_{2,1,1} \\ \psi_{1,2,1} \\ \psi_{2,2,1} \\ \psi_{1,1,2} \\ \psi_{2,1,2} \\ \psi_{1,2,2} \\ \psi_{2,2,2} \end{pmatrix} \tag{4.6}$$

The matrix M can be generated using the following python code:

```
from scipy.sparse import spdiags,eye,kron,linalg


def laplacian1d(N):
  e = np.ones(N)
  Laplacian = spdiags([e,—2*e,e], [—1,0,1], N, N)
  return Laplacian.toarray()


def laplacian3d(nx,ny,nz,sparse=0):
  N = nx*ny*nz
  ex = np.ones(nx); ey = np.ones(ny); e = np.ones(N)
  Ix = eye(nx); Iy = eye(ny); Iz = eye(nz)
  Lx = spdiags([ex,—3*ex,ex], [—1,0,1], nx, nx)
  Ly = spdiags([ey,—3*ey,ey], [—1,0,1], ny, ny)
  Lxy = kron(Iy,Lx)+kron(Ly,Ix)
  L = spdiags([e,e], [—nx*ny, nx*ny], N, N)
  Laplacian = (kron(Iz,Lxy)+L)
  if sparse: return Laplacian
  else: return Laplacian.toarray()
```

4.1.4 Kohn Sham: the density

The Kohn Sham density $n(\boldsymbol{r}) = \sum_i^N |\psi_i(\boldsymbol{r})|^2$ can be implemented as follows. The **o.Hkin** (Laplacian with some constant) are calculated in the **class** initialization method. Similarly, all quantities that do not change are calculated once at initialization. After finding eigenvalues and eigenvectors (KS orbitals) of the Hamiltonian, these orbitals are filled according to total particle number.

```
def getRhoKS(o,V):
  d = o.config['rho']
  E,psi = np.linalg.eigh(o.Hkin + np.diagflat(V))
  I = np.sum(psi**2,axis=0) * o.dx
  psi = psi/np.sqrt(I)[None,:]
  Nk = [2 for _ in range(d['N']//2)]
```

```
  if d['N'] % 2: Nk.append(1)
  rho = np.zeros_like(psi[:,0])
  for Nk_,psi_ in zip(Nk,psi.T):
    rho += Nk_ * psi_**2
  return rho,d['N'],E,psi
```

4.1.5 DPFT-TF: the density

Recall the third of the three self consistent equations [2.10] of DPFT in position space formalism: $n = \frac{\delta E_{\rm kin}^{\rm LGD}[V-\mu]}{\delta V}$. And with the Thomas Fermi approximation we have:

$$E_{\rm kin}^{\rm LGD}[V-\mu] = \int \frac{d\boldsymbol{r}d\boldsymbol{p}}{(2\pi\hbar)^3}[\frac{\boldsymbol{p}^2}{2m} + V(\boldsymbol{r}) - \mu]\ \eta(\mu - \frac{\boldsymbol{p}^2}{2m} - V) \tag{4.7}$$

Therefore in $3D$ and $1D$ the densities are respectively:

$$n(\boldsymbol{r}) = \int \frac{d\boldsymbol{p}}{(2\pi\hbar)^3} 1\cdot\eta(\mu - \frac{\boldsymbol{p}^2}{2m} - V) = \frac{1}{6\pi^2\hbar^3}P^3|_{P=\sqrt{2m(\mu-V)}} \tag{4.8}$$

$$n(r) = \int \frac{dp}{2\pi\hbar} 1\cdot\eta(\mu - \frac{p^2}{2m} - V) = \frac{P}{\pi\hbar}|_{P=\sqrt{2m(\mu-V)}} \tag{4.9}$$

They can be implemented as follows. The main difference in the DPFT code below compared with Kohn Sham code above is an additional for-loop that calculates the correct chemical potential μ so that the resulting density gives the correct total particle number. In Kohn Sham code it is easier since one simply fills the KS orbitals two particles at a time (for Fermions). Whereas in orbital free DPFT we find the chemical potential using **bisection method**, i.e., by testing many possible values of μ until we find the correct one. Therefore the density is calculated in each iteration using the internal function **getRhoN**:

```
def getRhoDPFT(o,V):
  def getRhoN(mu,V):
      muMinusV = mu - V; muMinusV[muMinusV<0] = 0
      rho = np.power(muMinusV,0.5); N = np.sum(rho) * o.dx
      return rho,N
  muMax = muMin = np.min(V); trueN = o.config['rho']['N']
```

```
while getRhoN(muMax,V)[1] < trueN: muMax = muMax*2
for i in range(o.config['loop']['Imax']):
  muMid = (muMax+muMin)/2
  rho,N = getRhoN(muMid,V)
  if(N > trueN): muMax = muMid
  else: muMin = muMid
  if 1-muMin/muMax < o.config['loop']['precision']:
    print('getRhoDPFT break at {}'.format(i)); break
return rho,N,0,0
```

4.1.6 Both: the result comparison

We first consider the Harmonic external potential, we can see that the Kohn Sham and DPFT-TF results agrees quite well with each other:

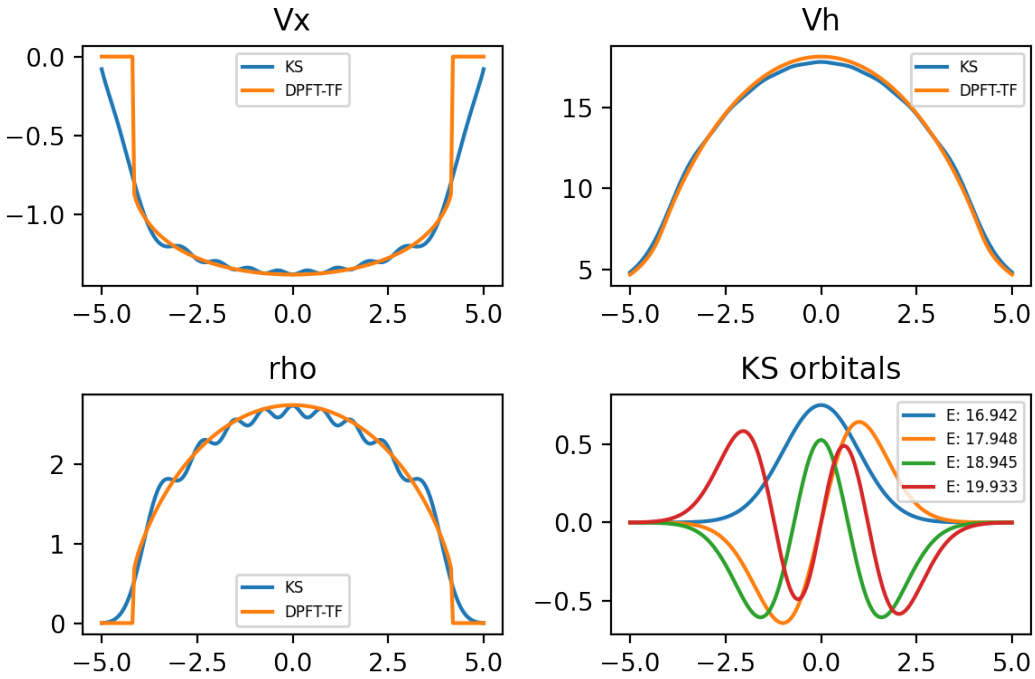
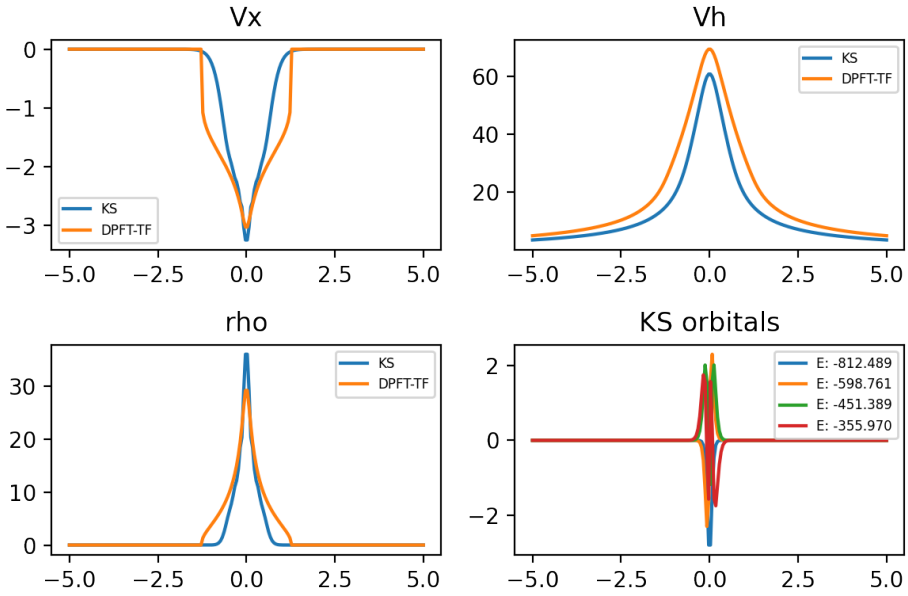


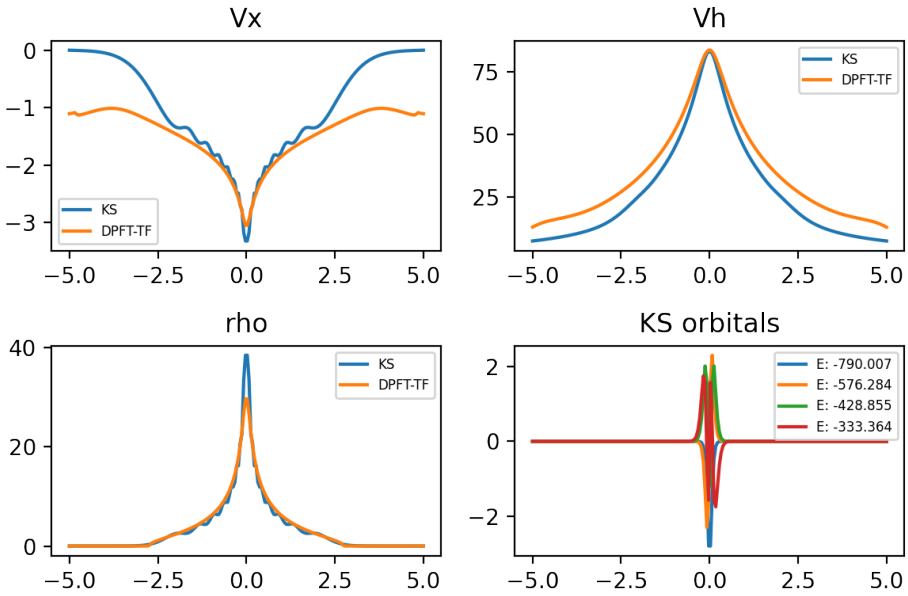
Figure 4.1: Kohn Sham vs DPFT-TF in 1D: $V_{\text{ext}} = r^2$, $N_{\text{particle}} = 18$

Next we consider the Coulomb external potential. This time the Kohn Sham and DPFT-TF results agree well with each other at large particle number $N = 36$, but differ significantly at

$N = 18$. Since we know the Kohn Sham is the most accurate, this result justified the commonly accepted observation: **Thomas Fermi approximation works better with larger particle numbers**.



(a) $N_{\text{particle}} = 18$



(b) $N_{\text{particle}} = 36$

Figure 4.2: Kohn Sham vs DPFT-TF in 1D: $V_{\text{ext}} = -\frac{Z}{r+\epsilon}$

4.2 DPFT in 2D using PyTorch Multi-GPU acceleration

I have successfully implemented the orbital free DPFT with Thomas-Fermi implementation, and compared it with custom Kohn Sham DFT code in 1D. Now I just have to modify the 1D code into 2D and add the Multi-GPU acceleration feature. The first and simplest modifications are **the computation grid** and **the Hartree interaction energy**:

```
# the computation grid
x = np.linspace(s['x'][0],s['x'][1],s['x'][2]); dx = x[1] - x[0]
y = np.linspace(s['y'][0],s['y'][1],s['y'][2]); dy = y[1] - y[0]
o.dV = dx * dy
xx, yy = np.meshgrid(x,y,sparse=True,indexing='ij')

# the Hartree interaction energy
VhKernel = ((xx[None,None,:,:]-xx[:,:,None,None])**2 \
            + (yy[None,None,:,:]-yy[:,:,None,None])**2 + 1e-2
  )**0.5 * o.dV
Vh = rho[None,None,:,:] / o.VhKernel
Vh = np.sum(Vh,axis=-1); Vh = np.sum(Vh,axis=-1);
```

4.2.1 Conversion from Numpy to PyTorch

Next we take four steps to make the conversion from Numpy to PyTorch.

In[1]:- The first step is to import the package and declare the GPU device:

```
import torch as tc
GPU = tc.device('cuda:0' if tc.cuda.is_available() else 'cpu')
```

- The second step is to convert Numpy arrays to PyTorch tensors, and load them into GPU memory:

```
o.VhKernel = tc.from_numpy(VhKernel).to(GPU)
rho = tc.zeros(self.xx.shape).to(GPU)
Vext = tc.from_numpy(Vext).to(GPU)
```

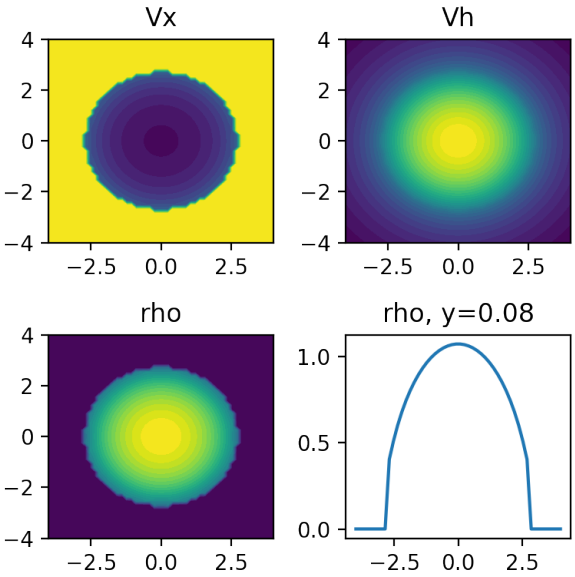
- The third step is to convert Python class to PyTorch module

```
class DPFT2D(tc.nn.Module):
  def __init__(self,config):
    super(DPFT2D,self).__init__()
    ......
  def forward(self,Vext): # self consistent loop
    ......
    return Vx.cpu().numpy(),Vh.cpu().numpy(),rho.cpu().numpy(),N.cpu().numpy()
```

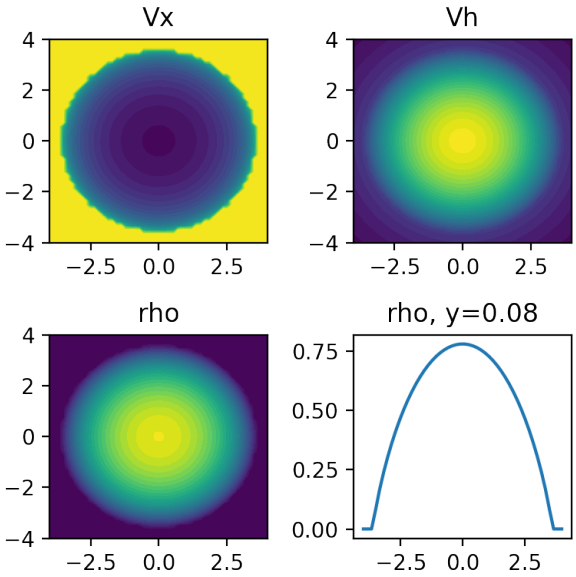
- The fourth step is to use multiple GPU as many as accessible by the computing device:

```
dft = DPFT2D(config)
if tc.cuda.device_count() > 1: dft = tc.nn.DataParallel(dft)
dft.to(GPU); print('using {} GPUs !'.format(tc.cuda.device_count()))
```

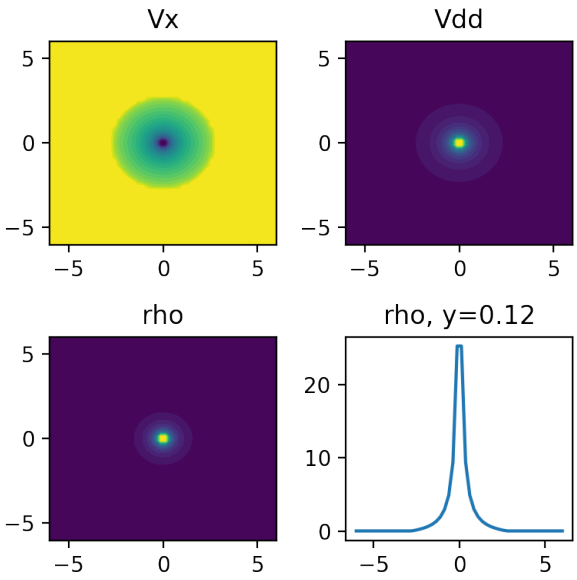
4.2.2 The result



(a) $V_{\text{ext}} = r^2$, $\epsilon = 10^{-3}$ for V_{h}



(b) $V_{\text{ext}} = r^2$, $\epsilon = 10^{-5}$ for V_{h}



(c) $V_{\text{ext}} = -\frac{Z}{r+\epsilon}$, $\epsilon = 10^{-5}$ for both V_{ext}, V_{h}

Figure 4.3: DPFT in 2D using PyTorch Multi-GPU acceleration

The above result with 50×50 grid was very quickly generated. The exact time for running the full calculation will not be discussed here in 2D, but will be discussed in detail for the 3D case with the more complex dipole-dipole interaction in momentum space. We can see that with a smaller value $\epsilon = 10^{-5}$, the Hartree potential $E_H[n(\boldsymbol{r})] = \frac{1}{2}\int d\boldsymbol{r}d\boldsymbol{r}' \, \frac{n(\boldsymbol{r})n(\boldsymbol{r}')}{|\boldsymbol{r}-\boldsymbol{r}'|+\epsilon}$ has larger values for $\boldsymbol{r} \approx \boldsymbol{r}'$.

Since Hartree energy represents the Coulomb repulsion between electrons, thus the density with smaller ϵ is more **expanded**. We can also see that the density at a fixed y value has exactly the same shape as in 1D, this is expected since both $V_{\text{ext}} = r^2$ and $V_{\text{ext}} = -\frac{Z}{r+\epsilon}$ are isotropic.

4.3 DPFT2D with dipole-dipole interaction V_{dd}

4.3.1 Momentum space dipole-dipole interaction

Recall the momentum space dipole-dipole interaction $V_{\text{dd}}^{\text{mom}}$ from equation [3.37]:

$$V_{\text{dd}}^{\text{mom}} := \frac{\delta E_{\text{dd}}}{\delta \rho(\boldsymbol{p}_a)} = \frac{\mu_0}{2} \int \frac{d\boldsymbol{p}_b}{(2\pi\hbar)^3} \left[\frac{(\boldsymbol{\mu} \cdot \boldsymbol{k})^2}{\boldsymbol{k}^2} - \frac{1}{3}\boldsymbol{\mu}^2 \right]_{\boldsymbol{k}=\frac{\boldsymbol{p}_a - \boldsymbol{p}_b}{\hbar}} \eta[\rho(\boldsymbol{p}_b) - \rho(\boldsymbol{p}_a)] \tag{4.10}$$

We implement it in two parts:

- The following constructs the interaction correlation **kernel** since it is unchanged during the loop. Therefore we run the following code once in the initialization method of our class:

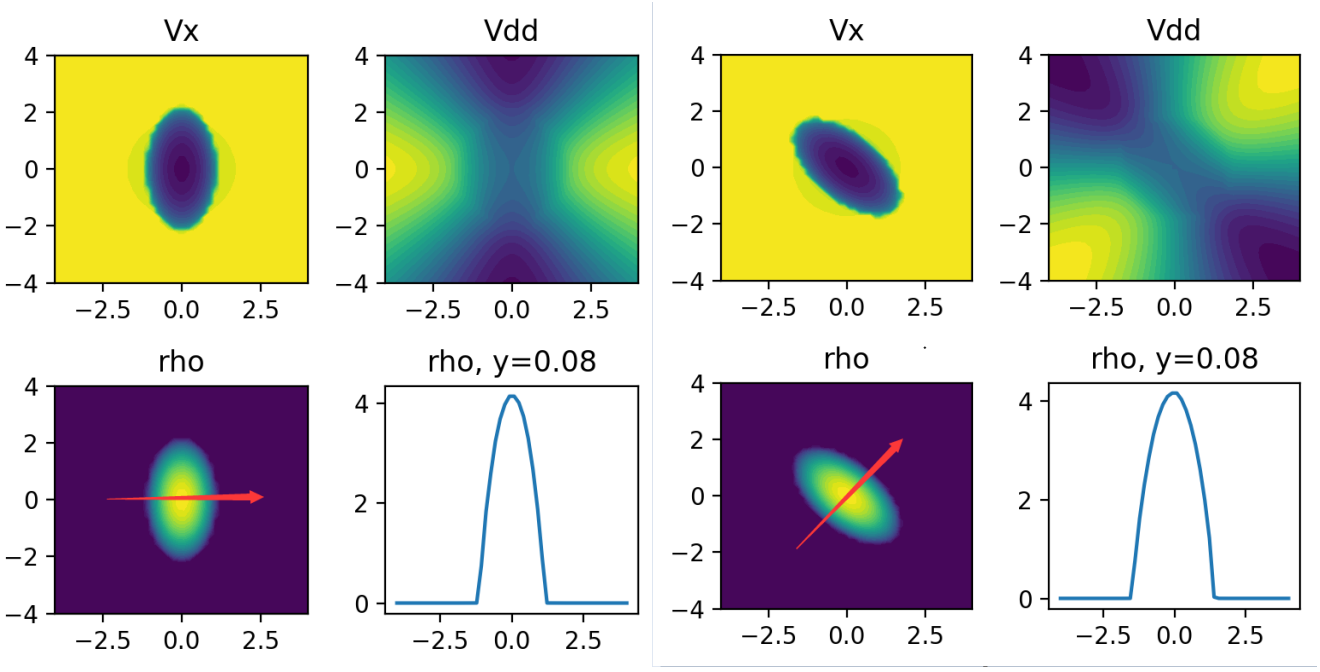
```
const = c['mu0']/16/(np.pi*c['hbar'])**3 * o.dV
numerator = (
    c['mu'][0] * (xx[None,None,:,:]—xx[:,:,None,None]) +
    c['mu'][1] * (yy[None,None,:,:]—yy[:,:,None,None])
)**2 + 1e—5
denominator = (xx[None,None,:,:]—xx[:,:,None,None])**2 \
  + (yy[None,None,:,:]—yy[:,:,None,None])**2 + 1e—5
mu2over3 = (c['mu'][0]**2 + c['mu'][1]**2)/3
VddKernel = const * (numerator/denominator — mu2over3)
o.VddKernel = tc.from_numpy(VddKernel).to(GPU)
```

- Then we run the following during the self consistent loop. Note that the Heaviside unit step function can be easily implemented using **the mask method**: the expression **rhoDiff** $<$ **0** generates an array of **True/False** values, and the expression **rhoDiff**[**rhoDiff** $<$ **0**] selects the elements in **rhoDiff** corresponding to **True**.

```
Vx = — (3/np.pi)**(1/3) * rho**(1/3)
rhoDiff = rho[None,None,:,:]—rho[:,:,None,None]
rhoDiff[rhoDiff<0] = 0; rhoDiff[rhoDiff>0] = 1
Vdd = rhoDiff * o.VddKernel
return Vx,Vdd.sum(—1).sum(—1)
```

4.3.2 Momentum space result

We use the constants $\hbar = 1, m = 1, \mu_0 = 500$ and the Harmonic external potential. By comparing the result between two different direction of the magnetic momentum $\boldsymbol{\mu}$, we arrive at the result that **the momentum space density is squeezed along the magnetic moment $\boldsymbol{\mu}$**. The physical interpretation behind this will be clear after comparing it with the position space result in the next section. Note that in the following the direction of $\boldsymbol{\mu}$ is labeled as red arrow:



(a) $\boldsymbol{\mu} = (0, 1)$ momentum space

(b) $\boldsymbol{\mu} = (0.7,\ 0.7)$ momentum space

Figure 4.4: The momentum space density is squeezed along the magnetic moment $\boldsymbol{\mu}$

4.3.3 Momentum space VS position space

Recall the position space dipole-dipole interaction $V_{\text{dd}}^{\text{pos}}$ from equation [3.38]:

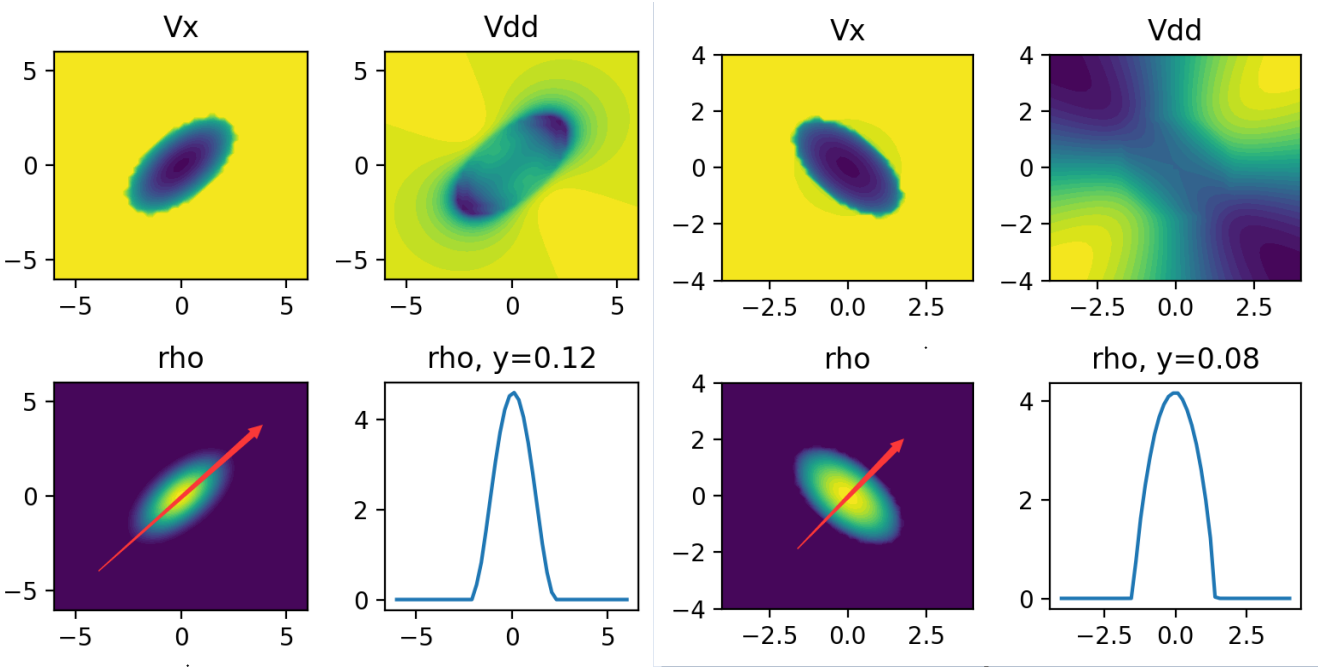
$$V_{\text{dd}}^{\text{pos}} := \frac{\delta E_{\text{dd}}}{\delta \rho(\boldsymbol{r}_a)} = \frac{\mu_0}{4\pi \boldsymbol{r}^3} \int d\boldsymbol{r}_b \left[-3\frac{(\boldsymbol{\mu} \cdot \boldsymbol{r})^2}{\boldsymbol{r}^2} + \boldsymbol{\mu}^2 \right]_{\boldsymbol{r}=\boldsymbol{r}_a-\boldsymbol{r}_b} \tag{4.11}$$

We implement it similarly:

```
# in the initialization method
const = c['mu0']/4/np.pi * o.dV
Vdd_x_Kernel = const * (mu2/(r**3 + 3e-3) - 6*muDotR2/(r**5 + 3e-3))
o.Vdd_x_Kernel = tc.from_numpy(Vdd_x_Kernel).to(GPU)

# during the self consistent loop
Vint = rho[None,None,:,:] * o.Vdd_x_Kernel
```

Note the term $3\frac{(\boldsymbol{\mu}\cdot\boldsymbol{r})^2}{\boldsymbol{r}^2}$ has been changed to $6\frac{(\boldsymbol{\mu}\cdot\boldsymbol{r})^2}{\boldsymbol{r}^2}$ in the implementation so that the **stretching effect** shown below is easier to view. The result using 3 also has the stretching effect but it is not obviously visible from the figure. We didn't have to make such an unwilling change in momentum space to view the **squeezing effect**. Now since one realize that **the expression [4.10] for momentum space and [4.11] for position space is essentially the same except for the step function term** $\eta[\rho(\boldsymbol{p}_b) - \rho(\boldsymbol{p}_a)]$, we can conclude that **in momentum space the squeezing effect coming from the term** $\left[\frac{(\boldsymbol{\mu}\cdot\boldsymbol{k})^2}{\boldsymbol{k}^2} - \frac{1}{3}\boldsymbol{\mu}^2\right]$ **is enhanced by the step function term** $\eta[\rho(\boldsymbol{p}_b) - \rho(\boldsymbol{p}_a)]$.



(a) position space

(b) momentum space

Figure 4.5: Position space density is stretched along $\boldsymbol{\mu}$, momentum space density is squeezed

Note how V_{dd} in momentum and position space differ from each other. Since in both spaces $\boldsymbol{\mu}$ vector has been chosen to be $(0.7,\ 0.7)$, from this we can try to explain these stretching/squeezing physically:

- In position space the density is stretched along $\boldsymbol{\mu}$, since the magnets tend to align head to tail with one another.

- In momentum space the density is squeezed along $\boldsymbol{\mu}$. This is physically clear from the relation $k = \frac{2\pi}{\lambda}$, and since the position space density profile is the same as the position space wave function profile, which is in turn the same as the wavelength λ profile, thus one would expect the effect in momentum space and position space to be the **inverse** of each other. This is also clear by looking at mathematical expressions: $V_{\text{dd}}^{\text{mom}}$ is essentially negative $V_{\text{dd}}^{\text{pos}}$ enhanced by the step function term. An alternative physical explanation from the particle perspective is also plausible: It is easier for an atom to move perpendicular to $\boldsymbol{\mu}$ than along $\boldsymbol{\mu}$ due to attraction and repulsion from neighbouring atoms, thus momentums perpendicular to $\boldsymbol{\mu}$ is favoured.

4.4 DPFT in 3D using PyTorch Multi-GPU acceleration

The 3D code can be upgraded easily from 2D. Although there is no manual hard work, the computer has to do much more calculation and provide much more memory. In the entire code developing process, I am using the popular Google Colab platform, which provides a limited amount of RAM and only a single GPU due to the vast amount of global users. Therefore, for the position space calculation, I was only able to run my code on a $25 \times 25 \times 25 = 15625$ grid points before running out of RAM. Note this in fact corresponds to $15625^2 = 244140625$ points for the 6D interaction energy **before contraction**. And for the slightly more complicated momentum space interaction energy, the limit is $20 \times 20 \times 20$. Despite the computing limitations, the result in 3D agrees well with 2D and is clearly visualizable. For the magnetic moment $\boldsymbol{\mu} = (0.7,\ 0.7,\ 0)$ lying on xy plane, the position space density is stretched along $\boldsymbol{\mu}$ whereas the momentum density is squeezed along $\boldsymbol{\mu}$. This is physically expected since the interaction energies have **rotational symmetry** around $\boldsymbol{\mu}$, **therefore the 2D case for $\boldsymbol{\mu} = (0.7,\ 0.7)$ must be the same as 3D case for $\boldsymbol{\mu} = (0.7,\ 0.7,\ 0)$**

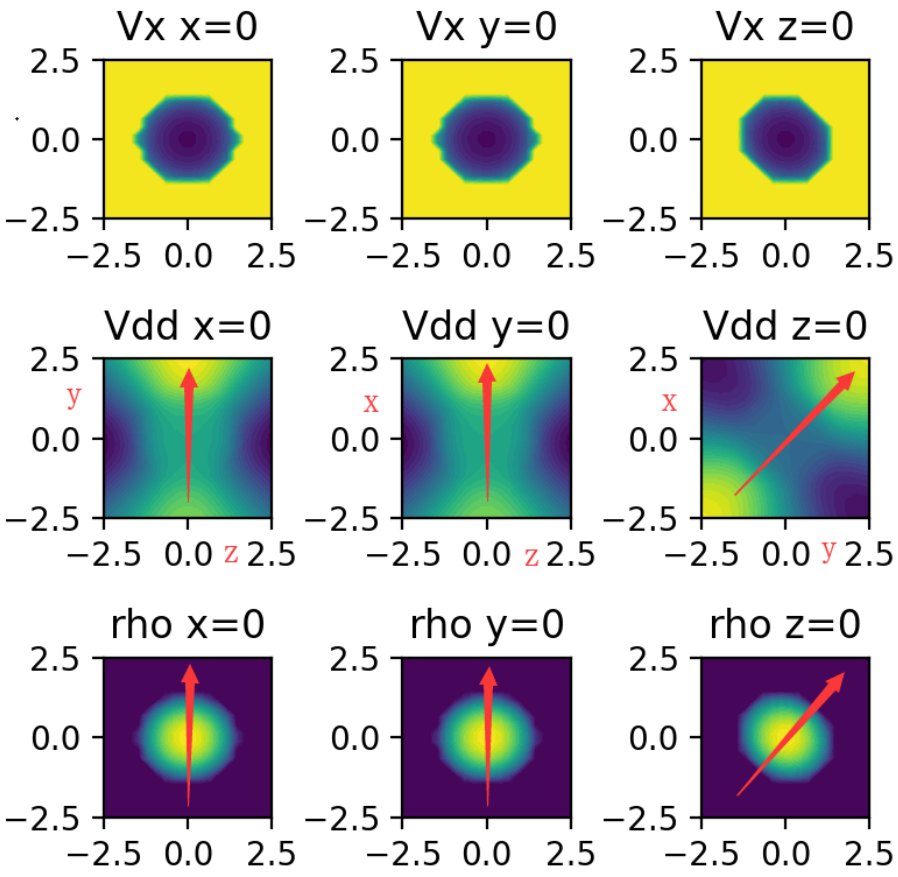


Figure 4.6: The result in 3D agrees well with 2D: momentum space

The three columns in the figures correspond to cross sections of the 3D quantities in $yz,\ xz,\ xy$ planes respectively. Since the stretching/squeezing effects are not obvious for ones in $yz,\ xz$ density plots, the reader is suggested to refer to the clearer V_{dd} plot instead. Note that the pattern we see in $yz,\ xz$ V_{dd} plots are expected since $\boldsymbol{\mu} = (0.7,\ 0.7,\ 0)$ has components along x and y directions. Also note that the densities in the position space plot below seems squeezed instead of stretched, but this is due to the aspect ratio of the plot, one should again refer to the V_{dd} plot.

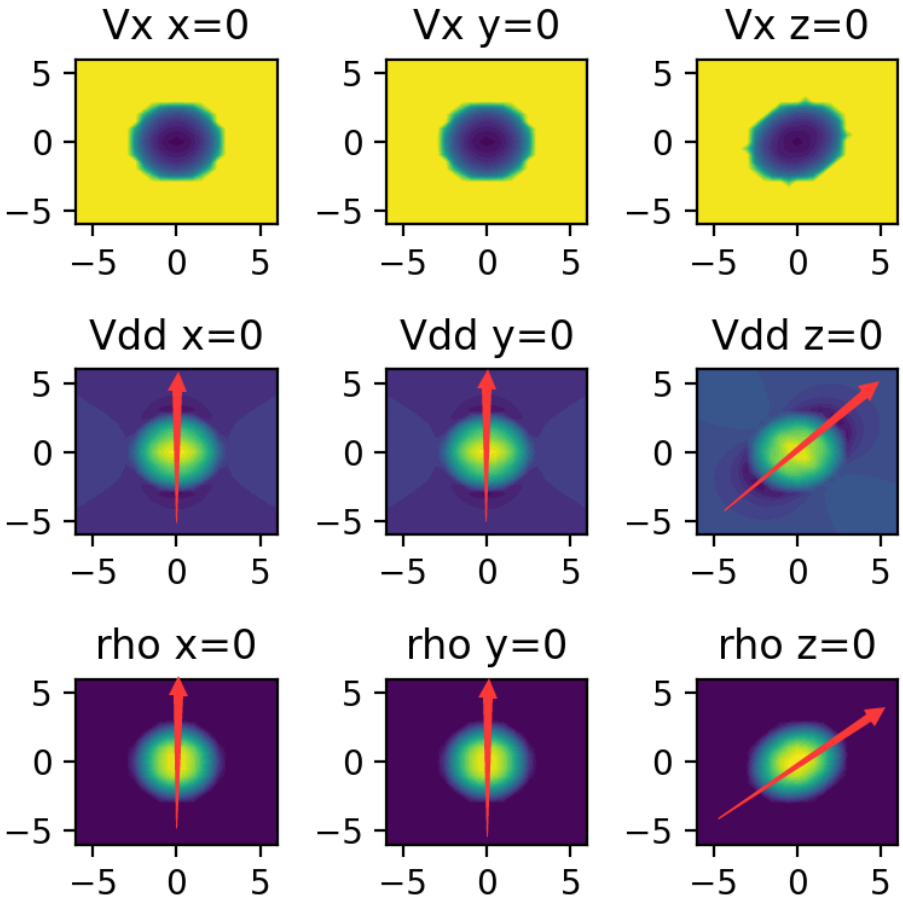
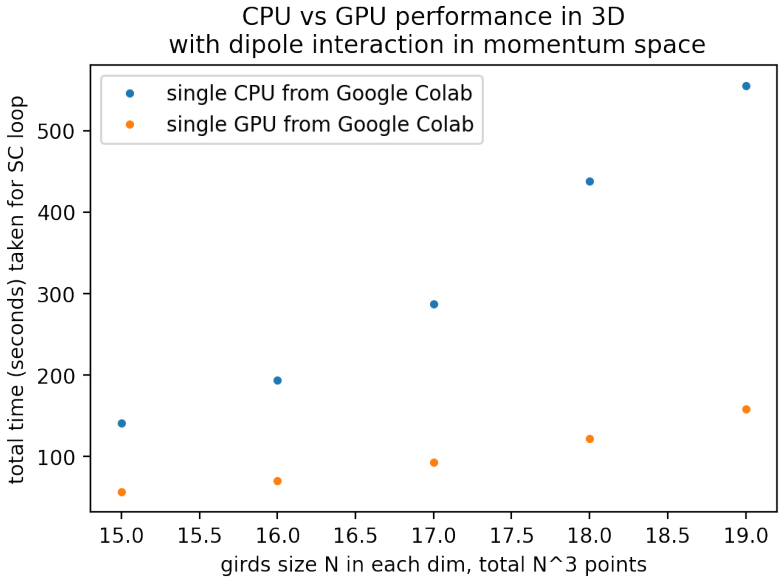


Figure 4.7: The result in 3D agrees well with 2D: position space

4.5 The performance

I compared CPU vs GPU performance in the 3D case with dipole interaction in momentum space, using exactly the same settings as the previous section. the details can be found in the appendix 3 the **config** code. This has been chosen for comparison because 3D is much more demanding than 1D and 2D, since the interaction energy which essentially is a correlation between the density with

some kernel, scales as N_{grid}^6 in 3D space. I have chosen momentum space since the dipole dipole interaction has a more complicated form in it than position space. We can see that, although the longest calculation in either case only takes a few minutes to run, the GPU is still very useful and provides much shorter run time at larger grid sizes.



(a) momentum space

Figure 4.8: CPU vs GPU performance in 3D

4.6 Python package distribution

The code from 1D to 3D has been published to `pypi.org` as an official python package. A brief documentation has been developed for user guidance. The results presented in this report are provided as examples.

- The package can be found from link `https://pypi.org/project/PyDPFT/`
- The GitHub repository is `https://github.com/tesla-cat/PyDPFT`

4.7 VASP and its enlightenment on my code

In order to improve my DPFT code, I spent quite some time learning how the state of the art DFT code is developed. I have chosen *Vienna Ab initio Simulation Package* (VASP)[21] and have

learned a lot of great features that enhances performance and provides more functionality. I will try to add these nice features into my own code as a future agenda.

In the $1D$ Kohn Sham code a single operation to calculate the eigenvalues and eigenvectors was extremely fast, therefore even the entire self-consistent loop takes merely seconds to run on a $1D$ gird of 200 points. However in $3D$ implementation the amount of calculations is vastly increased. Even with PyTorch GPU acceleration it took 159 seconds to run a single diagonalization operation in a $20 \times 20 \times 20$ grid:

```
H = tc.from_numpy(laplacian3d(nx=20,ny=20,nz=20)).to(gpu)
start_time = time.time()
E, Psi = tc.eig(H,eigenvectors=True)
print("--- %s seconds ---" % (time.time() - start_time))
print(E.shape,'\n',Psi.shape)
# The output is:
--- 159.46788477897644 seconds ---
torch.Size([8000, 2])    torch.Size([8000, 8000])
```

We can see that it took 159 seconds to find 8000 eigenvalues $\mathbf{E}$, each eigenvalue is composed of a real and an imaginary part, hence we have an array of shape $[8000, 2]$. We also have 8000 eigenvectors ψ and each eigenvector is composed of 8000 real numbers, hence $[8000, 8000]$. This approach of turning the Laplace operator into a matrix and then finding its eigenvalues is clearly not practical in 3D. The practical method used in the state of the art DFT programs is basis expansion. For example in VASP a core algorithm is known as projector augmented wave (PAW)[22] , where the plain waves[23] have been chosen as basis vectors:

$$\psi_i(\boldsymbol{r}) = \sum_{\boldsymbol{k}} c_{i,\boldsymbol{k}} \frac{1}{\sqrt{\Omega}} e^{i\boldsymbol{k}\cdot\boldsymbol{r}} = \sum_{\boldsymbol{k}} c_{i,\boldsymbol{k}} |\boldsymbol{k}\rangle \tag{4.12}$$

Substitute this into the single particle Schrodinger equation and multiply by $\langle \boldsymbol{l}|$:

$$\sum_{\boldsymbol{k}} \langle \boldsymbol{l}|H|\boldsymbol{k}\rangle c_{i,\boldsymbol{k}} = \epsilon_i c_{i,\boldsymbol{l}} \tag{4.13}$$

Therefore, the idea is that in order to find the Kohn Sham orbital $\psi_i(\boldsymbol{r})$, we find the eigenvectors of the matrix that is the representation of the Hamiltonian operator in plane wave basis. For kinetic

operator $\frac{(i\nabla)^2}{2}$ we have

$$\langle \boldsymbol{l}|\frac{(i\nabla)^2}{2}|\boldsymbol{k}\rangle = \frac{1}{2}\boldsymbol{k}^2\ \delta_{\boldsymbol{lk}} \tag{4.14}$$

For potential operator $V = V_{\text{ext}} + V_{\text{int}}$ we consider Fourier transform, which is essentially representing the V in plane wave basis

$$V(\boldsymbol{r}) = \int d\boldsymbol{g}\ V(\boldsymbol{g})e^{i\boldsymbol{g}\cdot\boldsymbol{r}} \quad V(\boldsymbol{g}) = \frac{1}{\Omega}\int d\boldsymbol{r}\ V(\boldsymbol{r})e^{-i\boldsymbol{g}\cdot\boldsymbol{r}} \tag{4.15}$$

$$\langle \boldsymbol{l}|V|\boldsymbol{k}\rangle = \int d\boldsymbol{g}\ \langle \boldsymbol{l}|V(\boldsymbol{g})e^{i\boldsymbol{g}\cdot\boldsymbol{r}}|\boldsymbol{k}\rangle = \int d\boldsymbol{g}\ V(\boldsymbol{g})\delta_{\boldsymbol{l},\boldsymbol{k}+\boldsymbol{g}} \tag{4.16}$$

For crystal lattice, $\boldsymbol{g}$ only take discrete values in the reciprocal lattice $\boldsymbol{g} \to \boldsymbol{G}_i$. Therefore we make the definition $\boldsymbol{l} - \boldsymbol{k} = \boldsymbol{g} \equiv \boldsymbol{G}_m - \boldsymbol{G}_n$, then the matrix elements of the Hamiltonian becomes $\langle \boldsymbol{l}|H|\boldsymbol{k}\rangle \to \langle \boldsymbol{k}+\boldsymbol{G}_m|H|\boldsymbol{k}+\boldsymbol{G}_n\rangle$, Finally the single particle Schrodinger equation

$$\sum_{\boldsymbol{k}}\langle \boldsymbol{l}|H|\boldsymbol{k}\rangle c_{i,\boldsymbol{k}} = \epsilon_i c_{i,\boldsymbol{l}} \quad \text{becomes} \tag{4.17}$$

$$\sum_n\ \left[\frac{1}{2}(\boldsymbol{k}+\boldsymbol{G}_n)^2\ \delta_{mn} + V(\boldsymbol{G}_m - \boldsymbol{G}_n)\right] c_{i,n} = \epsilon_i c_{i,m} \tag{4.18}$$

For Hartree part of the interaction potential we can use the familiar Fourier transform:

$$V_H[n(\boldsymbol{r})] = \int d\boldsymbol{r}'\ \frac{n(\boldsymbol{r}')}{|\boldsymbol{r}-\boldsymbol{r}'|} \tag{4.19}$$

$$V_H[n(\boldsymbol{G})] = \frac{1}{\Omega}\int d\boldsymbol{r}\ V_H[n(\boldsymbol{r})]e^{-i\boldsymbol{G}\cdot\boldsymbol{r}} = \frac{4\pi}{\boldsymbol{G}^2}n(\boldsymbol{G}) \quad n(\boldsymbol{G}) = \frac{1}{\Omega}\int d\boldsymbol{r}'\ n(\boldsymbol{r}')e^{-i\boldsymbol{G}\cdot\boldsymbol{r}'} \tag{4.20}$$

The external potential in the case of nucleus-electron Coulomb interaction is essentially the same as above. The Coulomb potential however, results in many zeros near the nucleus, requiring more plain waves to describe it. Thus in real applications the method of pseudopotentials or projector augmented waves are used. This idea of using basis expansion in orbital free DPFT has been discussed between Dr. Martin Trappe and myself. Since it doesn't use actual space but basis functions, the computation cost is much smaller than both Dr. Martin's and my own DPFT code. **Therefore we have set up the goal to somehow incorporate basis expansion into orbital free DPFT in the future.**

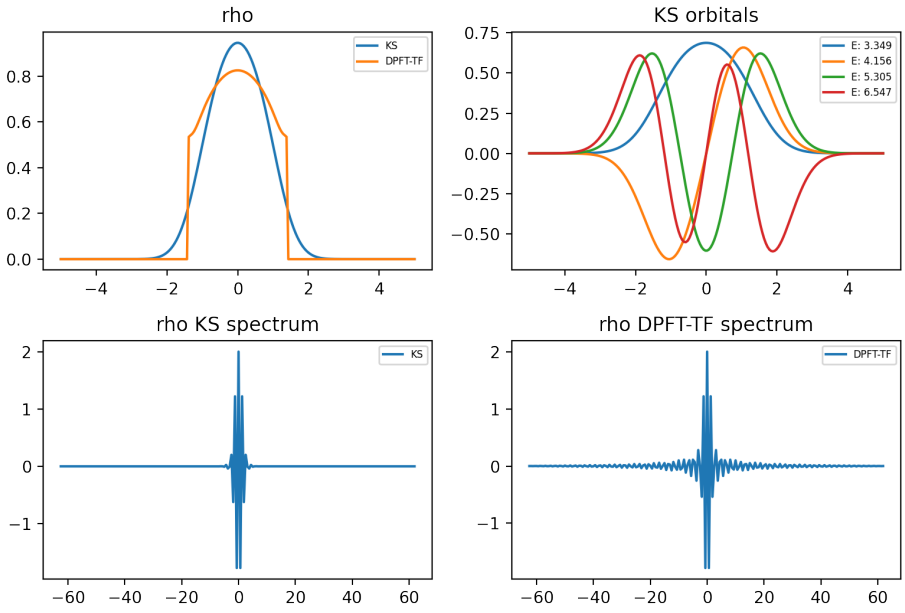
Chapter 5

An initial exploration: filtering the TF density

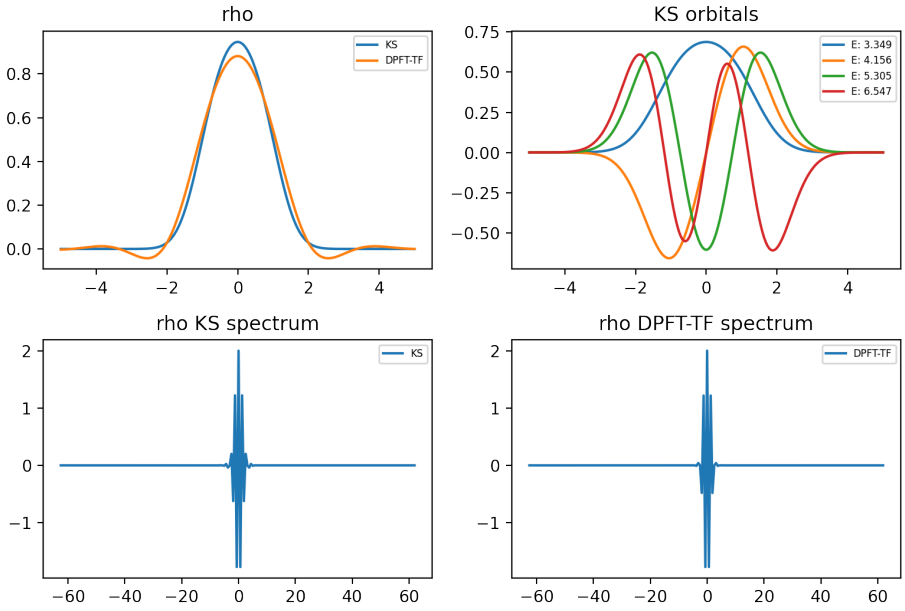
I had the following observation when playing with my codes: For larger particle numbers (N) TF density agrees much better with KS density. On the other hand for larger N, more KS orbitals (modes) are involved. Therefore I made the following guess:

- For small particle numbers, only few orbitals are involved in KS, but TF corresponds to the large N limit, i.e., has more modes. Therefore, we should apply a low pass filter on TF density to filter out larger modes.
- But for larger particle numbers, both KS and TF involve many modes, they agree well with each other so neither filtering is needed, nor would it help.

The following compares the TF density before and after applying a fourth order Butterworth low pass filter:

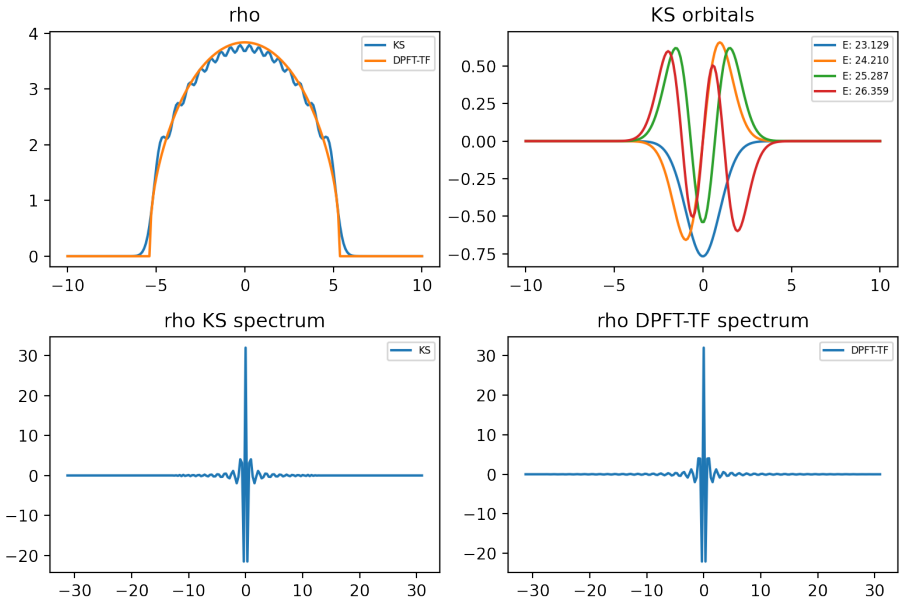


(a) original

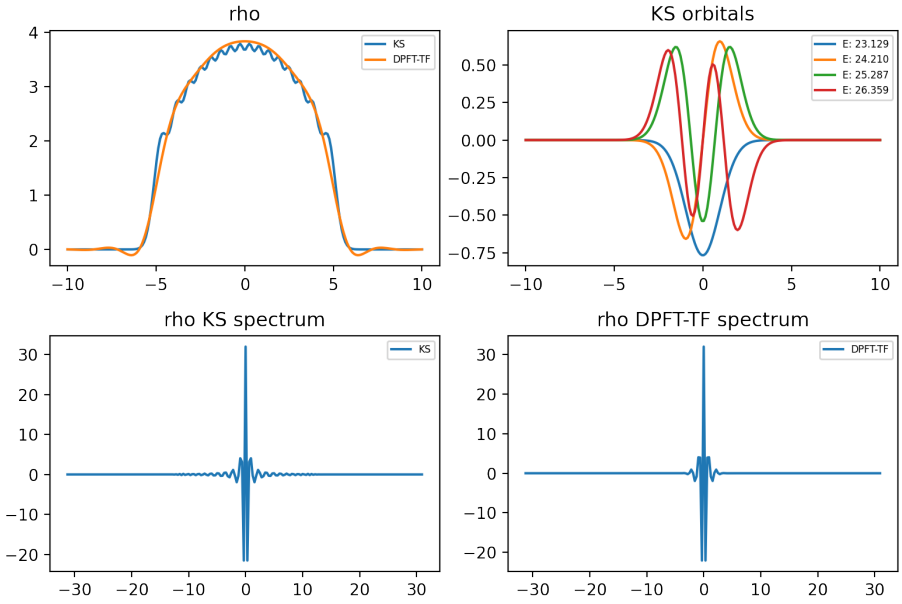


(b) filtered

Figure 5.1: For small particle numbers ($N = 2$), filtering the TF density improves the density



(a) original



(b) filtered

Figure 5.2: For larger particle numbers ($N = 32$), filtering is not needed

The physical reasoning is: The discontinuity in the density results from the discontinuity of the step function in TF approximation. The step function is in momentum (frequency) domain, consequently results in high frequency responses in position (time) domain. Therefore by applying a low pass filter on the density removes the high frequency responses and removes the discontinuity in TF density. This is an initial exploration and will be further examined and justified in the future.

Chapter 6

Summary

This thesis studied the Density Potential Functional Theory (DPFT) introduced by Julian Schwinger and Berge Englert in both position and momentum space. It is implemented with my own code using the state of the art Machine Learning library PyTorch.

In chapter 1, the classical Density functional theory (DFT) has been discussed. The Kohn Sham formalism that uses wave functions (orbitals) has been compared with the orbital free (OF) formalism. With the goal to justify the use of orbital free approach in our project.

In chapter 2, Density potential functional theory has been discussed. The idea of which is to use a Legendre transform to convert the energy functional of density only to a functional of both density and the **conjugate variable** of kinetic energy / external potential energy.

In chapter 3, the application of DPFT on spin polarized Fermi gas with magnetic dipole-dipole interaction is studied under the Thomas Fermi (TF) and Dirac approximation. The various energy functionals of this system, especially the dipole-dipole interaction functional E_{dd} have been derived for both position and momentum space.

In chapter 4, my main new contribution is discussed, which is developing **my own** orbital free DPFT code and releasing it as an official python packaged for interested users. My code has been compared with custom Kohn Sham DFT code in 1D and its correctness has been verified. Its main feature is the use of the state of the art Machine Learning library PyTorch to achieve multi-GPU acceleration. The performance of a single CPU and a single GPU was compared and the GPU demonstrated great speed enhancement. The magnetic dipole-dipole interaction has been implemented in $2D$ and $3D$, the result of which agrees well with physical expectation. For position space, since magnets tend to align head to tail with each other, the density profile of the Fermi

gas is **stretched** along the magnetic dipole momentum $\boldsymbol{\mu}$. For momentum space, both physical and mathematical observations suggested a inverse relationship between momentum and position space, therefore the density profile is **squeezed** along $\boldsymbol{\mu}$. Furthermore, the state of the art DFT software VASP has been studied in order to adopt its strengths into my own code in the future development, specifically we would like to try to incorporate the use of basis expansion method in to DPFT, which was proposed by Georg Kresse and has been proven very successful.

In chapter 5, another innovation that applies a low pass filter on TF density has shown enhanced accuracy and will be further explored in the future.

Bibliography

[1] Pierre Hohenberg and Walter Kohn. Inhomogeneous electron gas. *Physical review*, 136(3B):B864, 1964.

[2] Paul AM Dirac. Note on exchange phenomena in the Thomas atom. 26(3):376–385, 1930.

[3] Takeshi Yanai, David P Tew, and Nicholas C Handy. A new hybrid exchange–correlation functional using the Coulomb-attenuating method (CAM-B3LYP). *Chemical Physics Letters*, 393(1-3):51–57, 2004.

[4] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Physical review*, 140(4A):A1133, 1965.

[5] Llewellyn H Thomas. The calculation of atomic fields. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 23, pages 542–548. Cambridge University Press, 1927.

[6] Enrico Fermi. Un metodo statistico per la determinazione di alcune priorieta dell'atome. *Rend. Accad. Naz. Lincei*, 6(602-607):32, 1927.

[7] John P Perdew, Kieron Burke, and Matthias Ernzerhof. Generalized gradient approximation made simple. *Physical review letters*, 77(18):3865, 1996.

[8] C.F. von Weizsäcker. On the theory of nuclear masses. *Journal for physics*, 96(7-8):431–458, 1935.

[9] Lin-Wang Wang and Michael P Teter. Kinetic-energy functional of the electron density. *Physical Review B*, 45(23):13196, 1992.

[10] Mohan Chen, Xiang-Wei Jiang, Houlong Zhuang, Lin-Wang Wang, and Emily A Carter. Petascale orbital-free density functional theory enabled by small-box algorithms. *Journal of chemical theory and computation*, 12(6):2950–2963, 2016.

[11] Douglas R Hartree. The wave mechanics of an atom with a non-Coulomb central field. Part I. Theory and methods. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 24, pages 89–110. Cambridge University Press, 1928.

[12] Berthold-Georg Englert. Julian Schwinger and the semiclassical atom. *arXiv preprint arXiv:1907.04751*, 2019.

[13] Berthold-Georg Englert. Energy functionals and the Thomas-Fermi model in momentum space. *Physical Review A*, 45(1):127, 1992.

[14] Marek Cinal and Berthold-Georg Englert. Energy functionals in momentum space: exchange energy, quantum corrections, and the Kohn-Sham scheme. *Physical Review A*, 48(3):1893, 1993.

[15] Krzysztof Góral, Berthold-Georg Englert, and Kazimierz Rzażewski. Semiclassical theory of trapped fermionic dipoles. *Physical Review A*, 63(3):033606, 2001.

[16] Bess Fang and Berthold-Georg Englert. Density functional of a two-dimensional gas of dipolar atoms: Thomas-Fermi-Dirac treatment. *Physical Review A*, 83(5):052517, 2011.

[17] George A Henderson. Variational theorems for the single-particle probability density and density matrix in momentum space. *Physical Review A*, 23(1):19, 1981.

[18] Martin-Isbjörn Trappe, Yink Loong Len, Hui Khoon Ng, Cord Axel Müller, and Berthold-Georg Englert. Leading gradient correction to the kinetic energy for two-dimensional fermion gases. *Physical Review A*, 93(4):042510, 2016.

[19] Thanh Tri Chau, Jun Hao Hue, Martin-Isbjörn Trappe, and Berthold-Georg Englert. Systematic corrections to the Thomas-Fermi approximation without a gradient expansion. *New Journal of Physics*, 20(7):073003, 2018.

[20] Naomichi Hatano and Masuo Suzuki. Finding exponential product formulas of higher orders. In *Quantum annealing and other optimization methods*, pages 37–68. Springer, 2005.

[21] G Kresse and J Furthmüller. Software VASP, vienna (1999). *Phys. Rev. B*, 54(11):169, 1996.

[22] Georg Kresse and Daniel Joubert. From ultrasoft pseudopotentials to the projector augmented-wave method. *Physical review B*, 59(3):1758, 1999.

[23] Georg Kresse and Jürgen Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Physical review B*, 54(16):11169, 1996.

Appendix A

Kohn Sham and OF-DPFT in 1D

A.1 Import

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import spdiags,eye,kron,linalg
```

A.2 Functions

```
def laplacian1d(N):
  e = np.ones(N)
  Laplacian = spdiags([e,—2*e,e], [—1,0,1], N, N)
  return Laplacian.toarray()


def initVariables(o):
  s = o.config['space']
  c = o.config['const']
  o.x = np.linspace(s['x0'],—s['x0'],s['N'])
  o.dx = o.x[1] — o.x[0]
  L = laplacian1d(s['N']) / o.dx**2
  o.Hkin = —c['hbar']**2/(2*c['m']) * L
```

```
def getVint(o,rho):
  Vx = -(3/np.pi)**(1/3) * rho**(1/3)
  Vh = np.sum(rho[None,:]/np.sqrt((o.x[None,:]-o.x[:,None])**2+o.dx),
              axis=-1) * o.dx
  return Vx + Vh


def getRhoKS(o,V):
  d = o.config['rho']
  E,psi = np.linalg.eigh(o.Hkin + np.diagflat(V))
  I = np.sum(psi**2,axis=0) * o.dx
  psi = psi/np.sqrt(I)[None,:]
  Nk = [2 for _ in range(d['N']//2)]
  if d['N'] % 2: Nk.append(1)
  rho = np.zeros_like(psi[:,0])
  for Nk_,psi_ in zip(Nk,psi.T):
    rho += Nk_ * psi_**2
  return rho,d['N'],E,psi


def getRhoDPFT(o,V):
  mu = 42; ratio = 0
  for i in range(o.config['loop']['Imax']):
    muMinusV = mu - V
    muMinusV[muMinusV<0] = 0
    rho = np.power(muMinusV,0.5)
    N = np.sum(rho) * o.dx
    ratio = N/o.config['rho']['N']
    if abs(ratio-1) < 1e-3:
      print('getRhoDPFT break at {}'.format(i)); break
    mu = mu / ratio
  return rho,N,0,0
```

A.3 Main class

```
class DFT1D:
  def __init__(self,config):
    self.config = config
    initVariables(self)

  def forward(self,Vext,method): # self consistent loop
    l = self.config['loop']
    rho = np.zeros_like(self.x)
    for i in range(l['Imax']):
      Vint = getVint(self,rho)
      oldRho = rho
      if method == 'KS':
        rho,N,E,psi = getRhoKS(self,Vint + Vext)
      elif method == 'DPFT-TF':
        rho,N,E,psi = getRhoDPFT(self,Vint + Vext)
      if np.mean(np.abs(oldRho-rho)) < l['precision']:
        print('forward break at {}'.format(i)); break
    return Vint,rho,N,E,psi
```

A.4 Usage

```
config = {
    'space':{'x0':-5,'N':200},
    'loop':{'Imax':1000,'precision':1e-6},
    'rho':{'N':18},
    'const':{'hbar':1,'m':1},
}
methods = ['KS','DPFT-TF']
```

```
dft = DFT1D(config)
Vext = dft.x ** 2


plt.figure(dpi=200);
ax1=plt.subplot(221); ax1.title.set_text('Vint')
ax2=plt.subplot(222); ax2.title.set_text('rho')
ax3=plt.subplot(223); ax3.title.set_text('KS orbitals')
for m in methods:
  Vint,rho,N,E,psi = dft.forward(Vext,m)
  ax1.plot(dft.x,Vint,label=m)
  ax2.plot(dft.x,rho,label=m)
  if E is not 0:
    for i in range(4):
      ax3.plot(dft.x,psi[:,i], label=f"E: {E[i]:.3f}")
p={'size': 6}
ax1.legend(prop=p); ax2.legend(prop=p); ax3.legend(prop=p);
plt.tight_layout(); plt.show()
```

Appendix B

DPFT in 2D using PyTorch Multi-GPU acceleration

B.1 Import

```
import numpy as np
import matplotlib.pyplot as plt
import torch as tc
GPU = tc.device('cuda:0' if tc.cuda.is_available() else 'cpu')
```

B.2 Functions

```
def initVariables(o):
  s = o.config['space']; c = o.config['const']
  x = np.linspace(s['x'][0],s['x'][1],s['x'][2]); dx = x[1] - x[0]
  y = np.linspace(s['y'][0],s['y'][1],s['y'][2]); dy = y[1] - y[0]
  o.dV = dx * dy;
  xx, yy = np.meshgrid(x,y,sparse=True,indexing='ij')
  o.x, o.y = x, y; o.xx, o.yy = xx, yy

  r = ( (xx[None,None,:,:]-xx[:,:,None,None])**2 \
```

```
        + (yy[None,None,:,:]—yy[:,:,None,None])**2 )**0.5


    VhKernel = o.dV / (r + 10**—2.5)
    o.VhKernel = tc.from_numpy(VhKernel).to(GPU)


    const = c['mu0']/16/(np.pi*c['hbar'])**3 * o.dV
    muDotR2 = (
        c['mu'][0] * (xx[None,None,:,:]—xx[:,:,None,None]) +
        c['mu'][1] * (yy[None,None,:,:]—yy[:,:,None,None])
    )**2
    mu2 = c['mu'][0]**2 + c['mu'][1]**2
    Vdd_p_Kernel = const * (muDotR2/(r**2 + 10**—2.5) — mu2/3)
    o.Vdd_p_Kernel = tc.from_numpy(Vdd_p_Kernel).to(GPU)


    const = c['mu0']/4/np.pi * o.dV
    Vdd_x_Kernel = const * (mu2/(r**3 + 3e—3) — 6*muDotR2/(r**5 + 3e—3))
    o.Vdd_x_Kernel = tc.from_numpy(Vdd_x_Kernel).to(GPU)


def getVint(o,rho):
    Vx = — (3/np.pi)**(1/3) * rho**(1/3)
    VintName = 'Dipole—x—space'
    if VintName == 'Hartree':
        Vint = rho[None,None,:,:] * o.VhKernel
    elif VintName == 'Dipole—p—space':
        rhoDiff = rho[None,None,:,:]—rho[:,:,None,None]
        rhoDiff[rhoDiff<0] = 0; rhoDiff[rhoDiff>0] = 1
        Vint = rhoDiff * o.Vdd_p_Kernel
    elif VintName == 'Dipole—x—space':
        Vint = rho[None,None,:,:] * o.Vdd_x_Kernel
    return Vx,Vint.sum(—1).sum(—1)
```

```
def getRhoDPFT(o,V):
  def getRhoN(mu,V):
      muMinusV = mu - V; muMinusV[muMinusV<0] =0
      rho = muMinusV
      N = tc.sum(rho) * o.dV
      return rho,N
  muMax = 1; muMin = tc.min(V); trueN = o.config['rho']['N']
  while getRhoN(muMax,V)[1] < trueN: muMax = muMax*2
  for i in range(o.config['loop']['Imax']):
    muMid = (muMax+muMin)/2
    rho,N = getRhoN(muMid,V)
    if(N > trueN): muMax = muMid
    else: muMin = muMid
    if 1-muMin/muMax < o.config['loop']['precision']:
      print('getRhoDPFT break at {}'.format(i))
      break
  return rho,N
```

B.3 Main class

```
class DPFT2D(tc.nn.Module):
  def __init__(self,config):
    super(DPFT2D,self).__init__()
    self.config = config
    initVariables(self)

  def forward(self,Vext): # self consistent loop
    l = self.config['loop']
    rho = tc.zeros(self.xx.shape).to(GPU)
    Vext = tc.from_numpy(Vext).to(GPU)
```

```
    for i in range(l['Imax']):
      Vx,Vh = getVint(self,rho)
      oldRho = rho
      rho,N = getRhoDPFT(self,Vx+Vh+Vext)
      rho = (1-l['mix']) * oldRho + l['mix'] * rho
      if tc.mean(tc.abs(oldRho-rho)) < l['precision']:
        print('='*20+'CONVERGED AFTER {} !'.format(i)+'='*20); break
    return Vx.cpu().numpy(),Vh.cpu().numpy(),rho.cpu().numpy(),N.cpu().numpy()
```

B.4 Usage

```
def plot(dft,Vext,Vx,Vh,rho,Nparticle):
  print('Nparticle = ',Nparticle)
  print('Vh = ',Vh)
  plt.figure(dpi=200,figsize=(4,4));
  ax1=plt.subplot(221); ax1.title.set_text('Vx')
  ax2=plt.subplot(222); ax2.title.set_text('Vdd')
  ax3=plt.subplot(223); ax3.title.set_text('rho')
  y0 = int(Nspace/2)
  ax4=plt.subplot(224); ax4.title.set_text('rho, y={y:.2f}'.format(y=dft.y[y0]))
  resolution = 30
  ax1.contourf(dft.x,dft.y,Vx,resolution)
  ax2.contourf(dft.x,dft.y,Vh,resolution)
  ax3.contourf(dft.x,dft.y,rho,resolution)

  ax4.plot(dft.x,rho[:][y0]); print(dft.y[y0])
  plt.tight_layout(); plt.show()
  plt.contourf(dft.x,dft.y,Vext,resolution); plt.show()

Nspace = 50
```

```
x0 = -6
config = {
    'space':{'x':(x0,-x0,Nspace),'y':(x0,-x0,Nspace)},
    'loop':{'Imax':1000,'precision':1e-6,'mix':0.05},
    'rho':{'N':18},
    'const':{'hbar':1,'m':1,'mu0':5,'mu':[0.7,0.7]},
}
mu0 = [1,8,12]

def test(config):
  dft = DPFT2D(config)
  if tc.cuda.device_count() > 1: dft = tc.nn.DataParallel(dft)
  dft.to(GPU)
  print('using {} GPUs !'.format(tc.cuda.device_count()))

  Vext = dft.xx**2 + dft.yy**2
  #Vext = -1/(dft.xx**2 + dft.yy**2 )**0.5
  Vx,Vh,rho,Nparticle = dft.forward(Vext)
  plot(dft,Vext,Vx,Vh,rho,Nparticle)

for mu0_ in mu0:
  print(mu0_)
  config['const']['mu0'] = mu0_
  test(config)
```

Appendix C

DPFT in 3D using PyTorch Multi-GPU acceleration

C.1 Import

```
import numpy as np
import matplotlib.pyplot as plt
import torch as tc
GPU = tc.device('cuda:0' if tc.cuda.is_available() else 'cpu')
```

C.2 Functions

```
def initVariables(o):
  s = o.config['space']; c = o.config['const']
  x = np.linspace(s['x'][0],s['x'][1],s['x'][2]); dx = x[1] - x[0]
  y = np.linspace(s['y'][0],s['y'][1],s['y'][2]); dy = y[1] - y[0]
  z = np.linspace(s['z'][0],s['z'][1],s['z'][2]); dz = z[1] - z[0]
  o.dV = dx * dy * dz;
  xx, yy, zz = np.meshgrid(x,y,z,sparse=True,indexing='ij')
  o.x, o.y, o.z = x, y, z; o.xx, o.yy, o.zz = xx, yy, zz
```

```
  r = ( (xx[None,None,None,:,:,:]—xx[:,:,:,None,None,None])**2
      + (yy[None,None,None,:,:,:]—yy[:,:,:,None,None,None])**2
      + (zz[None,None,None,:,:,:]—zz[:,:,:,None,None,None])**2 )**0.5

  VhKernel = o.dV / (r + 10**—2.5)
  o.VhKernel = tc.from_numpy(VhKernel).to(GPU)

  const = c['mu0']/16/(np.pi*c['hbar'])**3 * o.dV
  muDotR2 = (
      c['mu'][0] * (xx[None,None,None,:,:,:]—xx[:,:,:,None,None,None]) +
      c['mu'][1] * (yy[None,None,None,:,:,:]—yy[:,:,:,None,None,None]) +
      c['mu'][2] * (zz[None,None,None,:,:,:]—zz[:,:,:,None,None,None])
  )**2
  mu2 = c['mu'][0]**2 + c['mu'][1]**2 + c['mu'][2]**2
  Vdd_p_Kernel = const * (6*muDotR2/(r**2 + 10**—2.5) — mu2/3)
  o.Vdd_p_Kernel = tc.from_numpy(Vdd_p_Kernel).to(GPU)

  const = c['mu0']/4/np.pi * o.dV
  Vdd_x_Kernel = const * (mu2/(r**3 + 3e—3) — 6*muDotR2/(r**5 + 3e—3))
  o.Vdd_x_Kernel = tc.from_numpy(Vdd_x_Kernel).to(GPU)

def getVint(o,rho):
  Vx = — (3/np.pi)**(1/3) * rho**(1/3)
  VintName = 'Dipole—p—space'
  if VintName == 'Hartree':
    Vint = rho[None,None,None,:,:,:] * o.VhKernel
  elif VintName == 'Dipole—p—space':
    rhoDiff = rho[None,None,None,:,:,:]—rho[:,:,:,None,None,None]
    rhoDiff[rhoDiff<0] = 0; rhoDiff[rhoDiff>0] = 1
    Vint = rhoDiff * o.Vdd_p_Kernel
  elif VintName == 'Dipole—x—space':
```

```
    Vint = rho[None,None,None,:,:,:] * o.Vdd_x_Kernel
  return Vx,Vint.sum(-1).sum(-1).sum(-1)



def getRhoDPFT(o,V):
  def getRhoN(mu,V):
      muMinusV = mu - V; muMinusV[muMinusV<0] =0
      rho = muMinusV**1.5
      N = tc.sum(rho) * o.dV
      return rho,N
  muMax = 1; muMin = tc.min(V); trueN = o.config['rho']['N']
  while getRhoN(muMax,V)[1] < trueN: muMax = muMax*2
  for i in range(o.config['loop']['Imax']):
    muMid = (muMax+muMin)/2
    rho,N = getRhoN(muMid,V)
    if(N > trueN): muMax = muMid
    else: muMin = muMid
    if 1-muMin/muMax < o.config['loop']['precision']:
      print('getRhoDPFT break at {}'.format(i))
      break
  return rho,N
```

C.3 Main class

```
class DPFT3D(tc.nn.Module):
  def __init__(self,config):
    super(DPFT3D,self).__init__()
    self.config = config
    initVariables(self)


  def forward(self,Vext): # self consistent loop
```

```
    l = self.config['loop']
    rho = tc.zeros(self.xx.shape).to(GPU)
    Vext = tc.from_numpy(Vext).to(GPU)
    for i in range(l['Imax']):
      Vx,Vh = getVint(self,rho)
      oldRho = rho
      rho,N = getRhoDPFT(self,Vx+Vh+Vext)
      rho = (1-l['mix']) * oldRho + l['mix'] * rho
      if tc.mean(tc.abs(oldRho-rho)) < l['precision']:
        print('='*20+'CONVERGED AFTER {} !'.format(i)+'='*20); break
      tc.cuda.empty_cache()
    return Vx.cpu().numpy(),Vh.cpu().numpy(),rho.cpu().numpy(),N.cpu().numpy()
```

C.4 Usage

```
def plot(dft,Vext,Vx,Vh,rho,Nparticle):
  print('Nparticle = ',Nparticle)
  plt.figure(dpi=200,figsize=(4,4));
  ax11=plt.subplot(331); ax11.title.set_text('Vx x=0')
  ax12=plt.subplot(332); ax12.title.set_text('Vx y=0')
  ax13=plt.subplot(333); ax13.title.set_text('Vx z=0')

  ax21=plt.subplot(334); ax21.title.set_text('Vdd x=0')
  ax22=plt.subplot(335); ax22.title.set_text('Vdd y=0')
  ax23=plt.subplot(336); ax23.title.set_text('Vdd z=0')

  ax31=plt.subplot(337); ax31.title.set_text('rho x=0')
  ax32=plt.subplot(338); ax32.title.set_text('rho y=0')
  ax33=plt.subplot(339); ax33.title.set_text('rho z=0')
```

```
  x0 = int(Nspace/2)
  resolution = 30
  ax11.contourf(dft.y,dft.z,Vx[x0,:,:],resolution)
  ax12.contourf(dft.x,dft.z,Vx[:,x0,:],resolution)
  ax13.contourf(dft.x,dft.y,Vx[:,:,x0],resolution)

  ax21.contourf(dft.y,dft.z,Vh[x0,:,:],resolution)
  ax22.contourf(dft.x,dft.z,Vh[:,x0,:],resolution)
  ax23.contourf(dft.x,dft.y,Vh[:,:,x0],resolution)

  ax31.contourf(dft.y,dft.z,rho[x0,:,:],resolution)
  ax32.contourf(dft.x,dft.z,rho[:,x0,:],resolution)
  ax33.contourf(dft.x,dft.y,rho[:,:,x0],resolution)
  plt.tight_layout(); plt.show()

Nspace = 20

x0 = -2.5
config = {
    'space':{'x':(x0,-x0,Nspace),'y':(x0,-x0,Nspace),'z':(x0,-x0,Nspace)},
    'loop':{'Imax':1000,'precision':1e-6,'mix':0.05},
    'rho':{'N':18},
    'const':{'hbar':1,'m':1,'mu0':100,'mu':[0.7,0.7,0]},
}

def test(config):
  dft = DPFT3D(config)
  if tc.cuda.device_count() > 1: dft = tc.nn.DataParallel(dft)
  dft.to(GPU)
  print('using {} GPUs !'.format(tc.cuda.device_count()))
```

```
  Vext = dft.xx**2 + dft.yy**2 + dft.zz**2
  #Vext = -1/(dft.xx**2 + dft.yy**2 )**0.5
  Vx,Vh,rho,Nparticle = dft.forward(Vext)
  return dft,Vext,Vx,Vh,rho,Nparticle


dft,Vext,Vx,Vh,rho,Nparticle = test(config)
plot(dft,Vext,Vx,Vh,rho,Nparticle)
```