Universiteit Utrecht

Ferromagnetism in ABC-trilayer graphene

Richard Olsen

Supervisors:
Prof. Cristiane de Morais Smith
Ralph van Gelderen M.Sc

Ferromagnetism in ABC-trilayer graphene

Richard Olsen

Supervisors: Prof. Cristiane de Morais Smith and Ralph van Gelderen M.Sc

26. August 2012

Abstract

Graphene was first realized in its free form in 2004 by the physicists Andre Geim and Konstantin Novoselov. First believed to only exist as a theoretical system, this material has some very unique properties. It is a two dimensional material with an extremely high charge mobility. It turns out that stacking several layers of graphene on top of each other causes the properties of this material to change. This thesis focuses on the ferromagnetic properties of graphene. In the first few chapters, the derivations of the ferromagnetic properties of both monolayer and bilayer graphene will be treated extensively based on existing literature. The results that are presented in the literature will be reproduced by the use of numerical methods. Finally, based on this, a similar set of numerical calculations will be done for ABC-stacked trilayer graphene, which will result in a phase diagram in the electron-electron coupling versus doping plane. Finally, the effects of Coloumb screening will be investigated. Due to the "flatter" low energy bands of ABC-trilayer graphene, it will be shown that this material exhibits stronger ferromagnetism than that seen in monolayer, bilayer and ABA-trilayer graphene. This is experimentally very exciting due to the difficulty of producing graphene samples with very low doping, which makes it a big challenge to detect ferromagnetism in monolayer graphene, bilayer graphene and ABA-trilayer graphene which undergoes a phase transition to paramagnetism at relatively low levels of doping. The increased level of doping needed to make ABC-trilayer graphene paramagnetic makes its realization in the ferromagnetic phase more manageable.

Contents

1 Introduction **1**
1.1 History and applications of graphene 1
1.2 Properties of graphene 2
1.3 Current status of ABC-trilayer graphene 4
1.4 Outline of the thesis 5

2 Preliminaries **7**
2.1 Numerical integration 7
2.1.1 Finite elements and Newton-Cotes 7
2.1.2 Double exponential integration 11
2.1.3 Duffy transformations of Coulomb integrals 13
2.2 Numerical minimization 14
2.2.1 Minimization by second order interpolation 14
2.3 Numerical diagonalization 15
2.3.1 Jacobi diagonalization 15
2.4 Numerical function inverse 22
2.4.1 Function interpolation 1D 22
2.4.2 Function inverse by linear interpolation 23
2.5 Numerical root search 23
2.5.1 Binary search for roots 23
2.6 Ferromagnetic instability 25
2.6.1 The exchange integral 25
2.6.2 Ferromagnetic instability 26

3 Monolayer graphene **29**
3.1 Band structure 29
3.1.1 The structure of Graphene 29
3.1.2 Tight binding Hamiltonian 31
3.1.3 Graphene and the Schroedinger equation 32
3.1.4 Single particle dispersion relation 34
3.1.5 Alternative method for calculating the dispersion 37
3.2 Ferromagnetism in monolayer graphene 39
3.2.1 Low energy dispersion relation 39
3.2.2 The monolayer field operator 40
3.2.3 Average kinetic energy 40
3.2.4 Exchange energy due to Coulomb interactions 42

3.2.5 Phase transitions 44

4 Bilayer graphene **50**
4.1 Band structure 50
4.1.1 Bilayer Hamiltonian 50
4.1.2 Bilayer dispersion 52
4.2 Ferromagnetism in bilayer graphene 53
4.2.1 Dispersion, low energy approximation 53
4.2.2 Average kinetic energy 55
4.2.3 Exchange energy due to Coloumb interactions 56
4.2.4 The exchange energy difference 63
4.2.5 The kinetic energy difference 67
4.2.6 Bilayer phase diagram 67

5 ABC-Trilayer graphene **72**
5.1 Band structure 72
5.1.1 ABC-trilayer Hamiltonian 72
5.1.2 ABC-trilayer dispersion 73
5.2 Ferromagnetism in ABC-trilayer graphene 74
5.2.1 Average kinetic energy 74
5.2.2 Exchange energy due to Coulomb interactions 75
5.2.3 The kinetic and exchange energy differences 78
5.2.4 ABC-trilayer phase diagram - no screening 80
5.3 Effects of screening 80
5.3.1 The ABC-trilayer action in the language of path integrals 82
5.3.2 Momentum space Feynman rules 83
5.3.3 Random phase approximation 86

6 Conclusions **88**

A Introduction to solid state physics **91**
A.1 Crystal Structures 91
A.1.1 Definition of a crystal structure 91
A.1.2 Symmetries 92
A.1.3 The notion of a primitive cell 93
A.1.4 ABA and ABC lattice structures 94
A.1.5 Indexing of planes 95
A.2 Diffraction and the Reciprocal Lattice 96
A.2.1 Bragg scatteing 96
A.2.2 The reciprocal lattice 96
A.2.3 The scattering amplitude 97
A.2.4 The diffraction condition 98
A.2.5 The first Brillouin zone 98
A.3 Band structures 99
A.3.1 Nearly free electron model 99
A.3.2 The Kronig-Penney model - square potential 100
A.3.3 General periodic potential model 101
A.3.4 The Bloch theorem 102
A.3.5 Metals and insulators 102
A.4 Fermi surfaces 103
A.4.1 Definition 103
A.4.2 Reduced Zone Scheme 103
A.4.3 Periodic and Extended Zone Scheme 104
A.4.4 Calculating energy bands - tight binding method 104

A.4.5 Calculating energy bands - Wigner-Seitz method . . . 106
A.4.6 Calculating energy bands - pseudopotential method . . . 107
A.4.7 Electron orbits . . . 107

B The Brillouin zone of Graphene **108**

C The Dirac points **110**

D Moving the singularities **113**

E C++ utility classes and algorithms **120**
E.1 The CCplx class . . . 121
E.2 The CCplxMatrix class . . . 123
E.3 The CReFunc1D class . . . 132
E.4 The CReFunc2D class . . . 136
E.5 The CCplxFuncMatrix2D class . . . 138
E.6 The CIntegralDE1D class . . . 144
E.7 The CIntegralDE3D class . . . 146
E.8 The CCmdLineParser class . . . 150
E.9 The CTimer class . . . 152

F Bilayer C++ code **153**
F.1 The CBiLayerDispersion class . . . 153
F.2 The CEnergy class . . . 161
F.3 The CExchEnergy class . . . 161
F.4 The CExchEnergyAD class . . . 167
F.5 The CExchEnergyND class . . . 170
F.6 The CKinEnergyAD class . . . 173
F.7 The CKinEnergyND class . . . 174
F.8 The CBiLayerPhasePoint class . . . 175
F.9 The `main()` entry point . . . 182

G ABC-Trilayer C++ code **186**
G.1 The CTriLayerDispersion class . . . 186
G.2 The CEnergy class . . . 196
G.3 The CExchEnergy class . . . 197
G.4 The CExchEnergyND class . . . 202
G.5 The CKinEnergyND class . . . 205
G.6 The CTriLayerPhasePoint class . . . 206
G.7 The `main()` entry point . . . 213

Bibliography **217**

CHAPTER 1

Introduction

1.1 History and applications of graphene

The pencil was invented as early as 1564. Its tip is made out of carbon atoms which are arranged into stacks of one-atomic thick layers called graphene. These honeycomb structured layers are weakly coupled together through van der Waals forces which allow the graphite to be deposited onto a sheet of paper as the pencil is pressed against it.

For a long time it was assumed that graphene did not exist in its free state [1]. This all changed in 2004, when a single layer of graphene was isolated and observed in its free form by Novoselov et al. [2]. The method used to isolate the graphene sheet was surprisingly simple. First graphite was deposited onto a piece of Scotch tape by simply rubbing graphite onto the tape. The deposited graphite was subsequently divided by repeatedly using the tape to peel off layers of graphite until the layers were sufficiently thin. However, the difficult task is to actually observe a single layer of graphene and distinguish this layer from the many multilayer debris created. After depositing the debris of graphene layers from the graphite onto a chosen SiO_2 substrate, a single layer of graphene will exhibit a unique optical pattern, thus enabling its observation with an ordinary optical microscope using a monochromatic light source [3, 4, 5] or by using techniques such as Raman microscopy [6]. Such experimental tools were not available much before the experimental realization of single sheets of graphene and in 2010 Andre Geim and Konstantin Novoselov earned the Nobel prize in physics for their experiments on graphene.

Before 2004, graphene was used as a theoretical model to describe the properties of graphite by first deriving the formalism needed for one layer of graphite. This was already done in an article by Wallace [7] in 1947, which laid the foundation for explaining the electrical conductivity, thermal conductivity, diamagnetic susceptibility and optical absorption of graphite. Since the realization of free graphene sheets in 2004, there has been a great interest in graphene and much research on the subject has been done since then. For example, new methods of producing graphene have been developed. One such method is that of chemical vapor deposition (CVD) on thin nickel layers [8, 9], which can produce graphene sheets up to a few centimeters. The CVD technique has also been extended to produce graphene on flexible substrates [10]. This is done in a few steps: (i) using CVD to grow graphene onto a thin flexible Cu sheet (ii) the adhesion of polymer supports to the graphene (iii) etching away the copper substrate (iv) depositing the remaining graphene onto the required target. This technique can produce graphene films over half a meter in the diagonal direction. Quite recently, it was also shown that graphene can be

created synthetically by using low temperature scanning tunneling microscopy and spectroscopy to manipulate carbon monoxide molecules at a copper surface [11].

One might first expect that a single layer of graphene would simply behave as graphite. However, this is a completely incorrect assumption. In fact, even a sheet of monolayer graphene versus two graphene sheets stacked on top of each other (bilayer graphene) have completely different properties, which is already apparent from their differing dispersion relations [1]. Explicit calculations of the ferromagnetic properties for single and bilayer graphene is done in later chapters, based on the work of Peres et al. [12] and Nilsson et al. [13].

The unique properties of graphene has opened up for possible applications of this remarkable material. One such application lies in the production of highly flexible touch panel displays [10]. In the production of touch panel displays transparent electrodes are needed to supply the individual pixels, which with todays technology is made using indium tin oxide (ITO) with an optical transparency of $\sim 90\%$. Since this coating only can withstand a very low amount of mechanical stress, it is extremely difficult to produce flexible screens. Therefore, graphene with its potentially superior conductivity and flexibility is a good candidate for the production of flexible flat panel displays.

Transistors made out of graphene have been a very attractive application since the realization of graphene in its free state. This is related to the high charge carrier mobility in the material [14] which can lead to transistors capable of extremely high frequencies. However, graphene is in principle a gapless material and, so far, only very small band gaps can be formed due to modification such as "rolling" the graphene in carbon nanotubes [15]. The small gap is highly undesirable, but it only applies to pure graphene transistors. In 2009, Lin et al. reported on a 100GHz transistor made by epitaxially forming graphene onto a Si wafer [14] which is referred to as a graphene FET (as apposed to the common metal oxide semiconductor FET i.e. MOS FET). At the time, a MOS FET was capable of $\sim$ 40GHz using the same gate length.

Another possible application of graphene is the detection of individual gas molecules [16] which is made possible by the 2D structure of graphene and thus its complete volume being exposed to surface adsorbates, combined with high conductivity leading to low noise. Lastly, as a possible application of graphene, one can mention the possibility of graphene quantum dots [17, 18], which can be used in linear and nonlinear optical devices due to the size-dependent absorption cross-section of a quantum dot [19]. The absorption spectra of quantum dots enables them to be used as an alternative to organic dyes and fluorescent proteins, which are much brighter and more stable against photobleaching. The tunable properties also makes them well suited for DNA detection and cell sorting and tracking [20].

1.2 Properties of graphene

As already mentioned, the properties of graphene differ a lot from the properties of graphite, and even the properties of monolayer graphene differs drastically from the properties of bilayer graphene. One of the most remarkable properties of monolayer graphene is its linear low energy dispersion and the fact that the corresponding two linear bands form a Dirac cone [1]. This results in the valence electrons behaving as massless Dirac fermions, where the Fermi velocity is calculated to be $v_F \approx 1 \cdot 10^6$m/s, which is $\sim 0.003c$, where c is the speed of light. Thus, the valence electrons of graphene are relativistic in nature. This unusual behavior ultimately leads to effects like the anomalous integer quantum Hall effect when graphene is subjected to a magnetic field [21, 22]. Once another layer of graphene is added, the low energy dispersion becomes quadratic. If yet another layer is added and stacked in an ABA fashion, where A and B refer to to different nodes on the honeycomb lattice, the low energy dispersion consists of a combination of linear and quadratic bands [23]. It might be tempting to believe that a change in orientation of the top layer

in ABA-stacked graphene, such that it becomes ABC-stacked, would not alter the dispersion. However, this is not correct. The low energy dispersion of ABC-stacked graphene is in fact cubic [24].

The honeycomb lattice structure is formed by sp^2 hybridization between one s orbital and two p orbitals [1]. Since each p orbital has an extra electron, the electronic band structure of a pure graphene sample will be half filled, which is an important property in determining the ferromagnetic behavior of graphene. Different filling of the bands results in either an insulator, semimetal, semiconductor or conductor and in the 1947 paper by Wallace [7] it was shown that graphene is a semimetal. This is also true for ABA-stacked graphene, but not for ABC-stacked graphene which behaves as a semiconductor [23]. It is also interesting to note that, unlike an ordinary metal, the magnetic susceptibility of graphene is temperature dependent [25].

Another important property, that follows directly from the single particle dispersion, is the density of states of graphene [26]. For monolayer graphene the density of states vanishes at low energies [1] which results in a very low Coloumb screening in the material. For bilayer graphene the density of states is constant and hence leads to very small screening effects, which is also true for ABA-stacked graphene due to the combination of linear and quadratic low energy electronic bands. However, the density of states for ABC-stacked graphene originates from a cubic low energy dispersion and will hence increase rapidly as the energy approaches zero. This makes ABC-trilayer graphene highly receptive to Coloumb screening, as will be demonstrated in this thesis by the effects of screening on the ferromagnetic behavior of the material.

In a 1966 article by Mermin and Wagner [27], it was proved that a two dimensional material with sufficiently short range interaction cannot be subject to long range order. Thus, due to the true two dimensional nature of graphene, long range order of the carbon atoms is only possible at zero temperature. This is yet another property that makes graphene a truly interesting material to study, since natural defects, like "graphene nanobubbles", open up for applications such as tunable optical lenses and the reverese effects (i.e. that the band structure is dependent on the curvature of graphene) can be used for band structure engineering [28]. When such a "nanobubble" is grown on a platinum surface the strain causes pseudo-magnetic fields up to 300T, which can be used in the study of charge carriers in previously inaccessible magnetic field regimes [29].

The property that will be the main focus of this thesis is that of ferromagnetism. As with all other properties that have been covered so far, ferromagnetism in graphene is highly dependent on the number of graphene layers and how they are stacked. In 2005, it was shown by Peres et al. [12], by using a tight binding model, that monolayer graphene exhibit phase transitions from a paramagnetic to a ferromagnetic phase as a function of doping and electron-electron coupling. The transitions occur both as first order and as second order transitions, depending on the investigated doping region. However, the electron-electron coupling of graphite is in the paramagnetic region of the phase diagram. Thus, by assuming approximately the same value for graphene, one can presume that it will remain paramagnetic.

The ferromagnetic properties of bilayer graphene were investigated by Nilsson et al. [13] by using a tight binging model similar to what was applied for monolayer graphene. Contrary to what was found in the monolayer case, bilayer graphene only exhibits first order phase transitions from the paramagnetic to the ferromagnetic state. Also, the material will be ferromagnetic even with zero doping at electron-electron coupling values comparable to that of graphite.

Adding another layer of graphene yields a change in the phase diagram. For ABA-stacking, the ferromagnetic effects will be reduced compared to bilayer graphene. This was shown by van Gelderen et al. in 2011 [23] using a tight binding model. However, the material will still remain ferromagnetic at zero doping. Here, we intend to investigate the ferromagnetic properties of ABC-stacked trilayer graphene.

1.3 Current status of ABC-trilayer graphene

As is the case for monolayer and bilayer graphene, ABC-stacked trilayer graphene has also been realized experimentally. Experiments done by Jhang et al. [30] show that a perpendicular electric field can open a band gap in ABC-stacked graphene, while the same electric field will cause a band overlap in ABA-stacked graphene. The band gap in ABC-stacked graphene has been observed to be as large as 120mV [31] which has not been observed in ABA-stacked graphene. This result was predicted theoretically by Avetisyan et al. [32] based on a tight binding approach within a self-consistent Hartree approximation. As was stated earlier, this is an extremely important property needed in order to create transistors.

A current can be induced by an electric field. Given an electric field, the current density is determined by the conductivity tensor. Similarly, a temperature gradient can, in some materials, induce a current. Then, for a given temperature gradient, the current density is determined by the thermoelectric conductivity tensor. The thermoelectric conductivity of monolayer graphene was measured by Zuev et al. [33] in 2009, while it was measured for bilayer graphene in 2010 by Wang et al. [34]. For monolayer graphene the experiments were performed both with an applied magnetic field of 8.8T and without a magnetic field, while for bilayer graphene the experiments were done with an applied magnetic field of up to 15T. In 2012, Ma et al. [35] performed numerical calculations of the thermoelectric conductivity of both ABA-stacked graphene and ABC-stacked graphene. The calculations were done using a tight binding model. An interlayer bias voltage and an applied strong perpendicular magnetic field was included in the calculations. It was shown that the thermoelectric properties were strongly dependent on the stacking order.

In a 2012 paper by Bala et al. [36], chiral tunneling effects were investigated in both ABA-stacked and ABC-stacked graphene. It was found that ABC-stacked graphene is a much better electron collimator than ABA-trilayer graphene. This is yet another example of why ABC-stacked trilayer graphene is a rich and worthwhile material to study.

An extensive research into the band structure of ABC-trilayer was done in 2009 by Mikito Koshino and Edward McCann [37]. This was achieved using an effective mass approximation. It was found that the electron and hole bands touching at zero energy support chiral quasiparticles characterized by Berry's phase $N\pi$ for N layers. The Berry's phase is a geometrical phase factor that is characteristic of a quantum system in an eigenstate, that is slowly transported around a circuit by varying parameters in the Hamiltonian [38].

The phonon spectra of ABC-stacked graphene was investigated theoretically using density functional theory in 2008 [39]. It was later investigated experimentally by using infrared absorption spectroscopy, where the intensities have been found to be much stronger than that of bilayer graphene [40].

Using magnetic fields up to 60T, there has been evidence of the integer quantum Hall effect in trilayer graphene. The Hall resistivity plateaus have been reproduced by assuming an ABC-stacking order of the graphene samples by using a self-consistent Hartree calculation [41]. It has been suggested that the differences in the quantum Hall effect between ABC- and ABA- stacking might be used to identify the stacking order of high-quality trilayer samples [42].

Another experiment that illuminates the dependence of stacking sequence of trilayer graphene was completed in 2010 by Mak et al. [43]. By using infrared absorption spectroscopy, it was shown that the optical conductivity spectra for ABC- and ABA- stacking order differs considerably. Theoretical calculations of the optical conductivity were done using a tight binding model within the single-particle excitation picture [44], and they support the experimental findings.

Yet another interesting feature of ABC-stacked graphene is related to its edge states. It has

been found that edge states are generally absent at multilayer armchair terminated ribbons, but that external electric fields between layers give rise to chiral edge states in ABC-stacked graphene in zigzag edge terminations. This is due to the electric field opening an energy gap that yields a valley-Hall effect [45]. Also, the surface of finite ABC stacks can produce an energy gap in the bulk-like states due to breaking of translational symmetry (while an ideal ABC bulk is a semimetal). Thus, the ground state of ABC-stacked graphene is found to be topologically nontrivial, even without an external electric field, unlike what is the case for AB-type stacks [46]. It turns out that ideal ABC-stacked graphene is a topological insulator that has surface states only at the Fermi level and it develops a tunable band gap when subjected to an external electric field [47]. This is in contrast to AB-type stacked graphene.

Other experiments that have been done on ABC-stacked trilayer graphene include high resolution transmission microscopy of ABC-stacked trilayer graphene on a SiC surface [48]. This type of measurement provides information on the interlayer distances in the graphene sample.

1.4 Outline of the thesis

Before delving into the details of multilayer graphene, it is appropriate to say a few words on the notation used throughout the thesis. In a stack of graphene sheets, it is customary to assign a label to each of the carbon atoms that appear right on top of each other. Usually, for bilayer graphene the lower carbon atom would be labeled a, referring to an a-sublattice and the upper carbon atom by b. However, in this thesis, both these carbon atoms will be labeled a. This is done since the primary article on ferromagnetism in bilayer graphene by Nilsson et al. [13] follows this notation. In order to keep the thesis consistent, this notation is extended to ABC-stacked trilayer graphene. This will become clearer in the following chapters.

The work done in this thesis relies heavily on numerical calculations. It is therefore important to have a good insight into the accompanying numerical methods. Most of chapter 2 is devoted to a review of these techniques, which is found in the literature on the subject, followed by a small review on ferromagnetism. In chapter 3 the basic properties of monolayer graphene are calculated, following the review outlined by Castro Neto et al. [1]. Next, the ferromagnetic properties of graphene are investigated using the same approach as Peres et al. [12]. At this point, every calculation is analytic in nature. In chapter 4 of the thesis, we move into the realm of bilayer graphene, where the expressions for kinetic energy and exchange energy are derived, following Nilsson et al. [13]. Further calculations in the article by Nilsson et al. are analytic and lead to a phase diagram in the electron-electron coupling versus doping plane. These calculations are not repeated in this thesis. Instead, these calculations and the resulting phase diagram have been reproduced numerically. The program is shown in appendices E and F.

Performing the calculations of the bilayer phase diagram is a challenging task. This is mainly due to the Coloumb potentials and their singularities in Fourier space. Integrating such a potential causes some problems. It is not simply a matter of performing a sum over the volume elements in the integration domain. This is in theory possible, but it would require too many partitions of the integration region which would lead to extremely long computation times. In order to perform the integrations in a reasonable amount of time, the integral must be transformed into one that is suitable for numerical integration. There are also many possibilities for the choice of the integration algorithm. Not all algorithms can handle integrands that tend to infinity at the boundary.

The phase diagram for bilayer graphene has been evaluated numerically in two ways; first, by using the analytic expressions of the single particle dispersion relation and second, by calculating the dispersion (and the density of states) by numerical diagonalization of the bilayer Hamiltonian. The numerical diagonalization becomes necessary when extending the numerical calculations done for bilayer graphene to calculations of the corresponding phase diagram of ABC-trilayer graphene.

The reason for choosing a numerical diagonalization is twofold. Firstly, the low energy approximation of the ABC-trilayer Hamiltonian is a 6×6 matrix that is very complicated to diagonalize exactly, even if only nearest neighbor hopping parameters are included in the Hamiltonian. Secondly, it is easier to extend the calculations to include more hopping parameters in the case of numerical diagonalization.

In chapter 5, the expressions for the kinetic energy and the exchange energy of bilayer graphene that were done based on existing literature, are extended to the case of ABC-stacked trilayer graphene. The additional potential between the outer layers of the graphene stack will lead to a matrix potential that is not difficult to diagonalize, but leads to a complicated analytic expression. One of the challenges is to find a way of rewriting this matrix in a form that will lead to an equivalent expression for the exchange energy as that of bilayer graphene. This is necessary in order to be able to use the algorithms of the bilayer program as basis for the ABC-trilayer program. The resulting program is shown in appendices E and G.

In the first part of chapter 5, the effects of screening are not included. The resulting phase diagram shows enhanced ferromagnetism compared to both bilayer graphene and ABA-stacked trilayer graphene. The phase transition is first order at all points, while the material remains ferromagnetic at zero doping. Since it is very difficult to produce pure graphene, with no doping, it is reasonable to believe that ferromagnetism is more easily detected in an ABC-trilayer graphene sample than in any other type of sample with the same number of layers or fewer layers. This makes this result very interesting for future experiments.

In the remaining parts of chapter 5, the effects of screening are outlined by renormalizing the Coloumb potential for ABC-trilayer by utilizing a random phase approximation.

Appendix A of the thesis is provided to give a review of solid state physics, covering the most important features related to ferromagnetism. This is done as a convenience for readers that are unfamiliar with the subject.

A big challenge in writing down the algorithms and the corresponding program that calculate a phase diagram is to optimize the algorithms. It is therefore curious to note that, at the time of writing, a Core i7 Quad core processor running at 55.5 Gflops is used to perform the calculations. It uses 21 days to calculate one phase diagram. Using three such processors it now takes approximately 7 days to calculate one phase diagram. In 1992, a typical desktop computer was running at 0.03 Gflops. Thus, it would, at the time, take 106 years to calculate one phase diagram for ABC-stacked trilayer graphene.

CHAPTER 2

Preliminaries

A large part of the work done in the thesis is based on numerical calculations. Numerical integration is used to calculate the three dimensional exchange integral, while the Hamiltonian of ABC-trilayer graphene needs to be diagonalized numerically to find the dispersion. Once the energy difference between two systems have been calculated, numerical methods will be used to find the minimum of this difference, depending on the order parameter. Thus, the preferred value of the order parameter can be identified. In order to calculate the kinetic energy, it will become important to know the inverse of the dispersion. Since the dispersion ultimately will be calculated numerically, it will be necessary to search for the inverse numerically. Numerical algorithms will also be employed to find the root of a function. This chapter will explain the theoretical background of these algorithms as well as how to implement them. A large part of the thesis concerns ferromagnetism. Therefore, a section of this chapter has been devoted to this subject in general.

2.1 Numerical integration

2.1.1 Finite elements and Newton-Cotes

In order to evaluate an integral on a computer, a method of approximating the area or volume bounded by the integrand $f(\mathbf{x})$ is needed. One such method approximates the function of the integrand by a function that is easy to integrate analytically and subsequently performs the integration on this function. A three dimensional integrand can be approximated by first dividing the integration domain V into a mesh of tetrahedrons or rectangular cuboids K_i. Then, the integrand is approximatet on each K_i. Such an approximation on K_i constitutes a *finite element.* In order to project the integrand onto the finite element, the projection $\mathcal{I}_{K_i}(f)$ is defined. This is referred to as an *interpolant.*

In the following, finite elements will be used to derive the Newton-Cotes formulas for three dimensional numerical integration. An in-depth treatment of finite elements is given in reference [49].

Formally a finite element is defined by the triple $(K, \mathcal{P}, \mathcal{N})$, where

- $K \subseteq \mathbb{R}^n$ is a closed bounded set with non-empty interior and piecewise smooth boundary called the *element domain.*
- $\mathcal{P}$ is a finite dimensional space of functions on K called the *shape functions.*

- $\mathcal{N} = \{N_1, N_2, \ldots, N_k\}$ is a basis for the continuous dual space of $\mathcal{P}$ called the *nodal variables*. For the purpose of this discussion, this means that if $\{\phi_j\}$ is a basis of $\mathcal{P}$ then $\mathcal{N}$ is defined such that $N_i(\phi_j) = \delta_{ij}$.

Let $\mathcal{P}$ contain three dimensional third degree polynomials, which will eventually approximate the integrand. Once $\mathcal{N}$ has been defined, the relation $N_i(\phi_j) = \delta_{ij}$ can be used to calculate the coefficients of the basis polynomials ϕ_i. There are several ways of defining $\mathcal{N}$. Perhaps the simplest way is to say that $N_i(f) := f(z_i)$ where z_i are fixed points relative to K. Thus, $\phi_i(z_j) = \delta_{ij}$ where $\{\phi_i\}$ is a basis of $\mathcal{P}$. The interpolant is defined as

$$\mathcal{I}_K(f) = \sum_{i=1}^{k} N_i(f)\phi_i.$$

With this definition, it is clear that for any polynomial

$$P(\mathbf{x}) = \sum_{n=1}^{k} c_n\phi_n(x),$$

written in terms of the k basis elements $\{\phi_i\}$, the interpolant satisfies

$$\mathcal{I}_K(P) = \sum_{i=1}^{k} N_i(\sum_{n=1}^{k} c_n\phi_n)\phi_i = \sum_{n=1}^{k} c_n \sum_{i=1}^{k} N_i(\phi_n)\phi_i = \sum_{n=1}^{k} c_n \sum_{i=1}^{k} \delta_{in}\phi_i = \sum_{n=1}^{k} c_n\phi_n = P.$$

Note that since $N_i \in \mathcal{N}$ and $\mathcal{N}$ is a dual space, then N_i is by definition a linear functional and subsequently $N_i(f_1 + f_2) = N_i(f_1) + N_i(f_2)$.

Thus, the interpolant yields an exact answer for any polynomial of that degree, denoted by $\deg \mathcal{P}$. However, the integrand that is to be approximated will most likely not be a polynomial on K_i.Therefore, for $f \notin \mathcal{P}$ there is an associated error $||f - \mathcal{I}_{K_i}(f)|| \neq 0$. For a smooth function f, this error is bounded. To keep the error bounded, any singularities of the integrand should be moved to the boundaries of the integral (i.e. singularities that cancel / are suppressed by the integration).

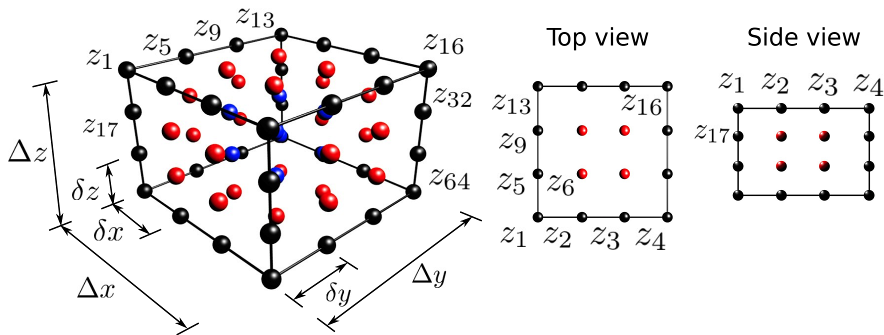


Figure 2.1: Finite element domain K.

Consider the finite element domain shown in figure 2.1. The evaluation points (called nodes) are indicated as solid spheres in the rectangular cuboid (i.e the points z_j at which $\phi_i(z_j) = \delta_{ij}$). Note that

$$\mathcal{I}_K(f)(z_j) = \sum_{i=1}^{k} N_i(f)\phi_i(z_j) = N_j(f) = f(z_j).$$

Thus, at each node z_i any smooth function f will coincide exactly with the polynomial approximation. In three dimensions a tensor product polynomial is defined as

$$Q(x, y, z) = \sum_{i,j,k} c_{i,j,k} p_i(x) p_j(y) p_k(z),$$

where p_i are basis polynomials of a given degree and i, j, k run over all basis elements. In the following, only polynomials of degree three will be considered. By using tensor product polynomials as elements of $\mathcal{P}$ a particularly simple basis $\{\phi_i\}$ of $\mathcal{P}$ can be constructed. Since $p_i(x)$ is independent of y and z there exists a complete basis for it that is independent of $p_j(y)$ and $p_k(z)$. An equivalent argument holds for $p_j(y)$ and $p_k(z)$. The space of third degree polynomials is completely spanned by four independent basis elements. Therefore, four evaluation points are required to solve for the unknown coefficients in the basis functions

$$p_i = a_i x^3 + b_i x^2 + c_i x + d_i.$$

These evaluation points are nodes in figure 2.2. To satisfy $\phi(z_i) = \delta_{ij}$ let $p_i(x_j) = \delta_{ij}$ with equivalent expressions for the two remaining directions y and z. Consider one of the directions, for example x. It helps to imagine that each node in figure 2.1, along this direction, is assigned one of the basis polynomials such that $p = 1$ at its assigned node. To this end, relabel the basis $\{\phi_i\}$ where i runs from 1 to 64, to $\{\phi_{i,j,k}\}$ where i, j, k all run from 1 to 4. Then it is clear that $\phi_{i,j,k} = p_i p_j p_k$ satisfies $\phi_{i,j,k}(x_l, y_m, z_n) = \delta_{il}\delta_{jm}\delta_{kn}$ as reqiured.

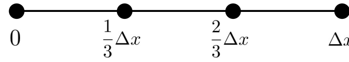


Figure 2.2: Evaluation points needed to find basis of polynomials p of one variable.

The next task is to find explicit expressions for the basis elements $p_i(x)$. Since $p_1(1/3\Delta x) = p_1(2/3\Delta x) = p_1(\Delta x) = 0$ this third degree polynomial of three distinct roots can be written as

$$p_1(x) = c_1 \left(x - \frac{1}{3}\Delta x\right)\left(x - \frac{2}{3}\Delta x\right)(x - \Delta x).$$

The constant c_1 is fixed by the requirement that $p_1(x_1) = p_1(0) = 1$ which yields

$$p_1(x) = -\frac{9}{2\Delta x^3}\left(x - \frac{1}{3}\Delta x\right)\left(x - \frac{2}{3}\Delta x\right)(x - \Delta x).$$

Furthermore, $p_2(0) = p_1(2/3\Delta x) = p_1(\Delta x) = 0$ and thus

$$p_2(x) = c_2 x\left(x - \frac{2}{3}\Delta x\right)(x - \Delta x).$$

Fixing c_2 by the requirement $p_2(1/3\Delta x) = 1$ one finds

$$p_2(x) = \frac{27}{2\Delta x^3} x\left(x - \frac{2}{3}\Delta x\right)(x - \Delta x).$$

Similarly,

$$p_3(x) = -\frac{27}{2\Delta x^3} x\left(x - \frac{1}{3}\Delta x\right)(x - \Delta x),$$

and

$$p_4(x) = \frac{9}{2\Delta x^3} x\left(x - \frac{1}{3}\Delta x\right)\left(x - \frac{2}{3}\Delta x\right).$$

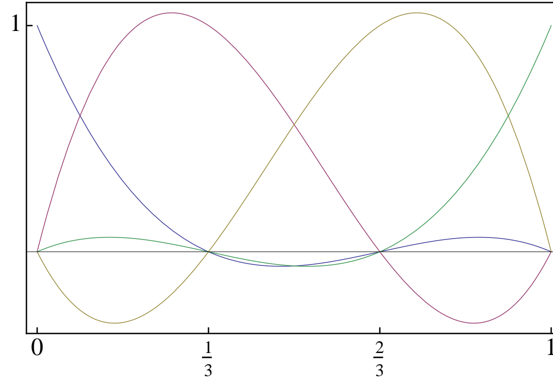


Figure 2.3: Basis functions p_1, p_2, p_3 and p_4.

The resulting basis functions for $\Delta x = 1$ are shown in figure 2.3. The basis functions for $p_j(y)$ and $p_k(z)$ are identical apart from Δx being substituted by Δy and Δz, respectively.

The interpolant corresponding to the above finite element is

$$\mathcal{I}_K(f)(x,y,z) = \sum_{i,j,k=1}^{4} N_{i,j,k}(f)\phi_{i,j,k}(x,y,z) = \sum_{i,j,k=1}^{4} f(x_i,y_j,z_k)p_i(x)p_j(y)p_k(z),$$

which can be used to approximate the integrand by a polynomial. The approximate volume can be found by integrating over the domain K:

$$\begin{aligned}\Delta V_K &= \int_K dV \mathcal{I}_K(f)(x,y,z) = \int_0^{\Delta x} dx \int_0^{\Delta y} dy \int_0^{\Delta z} dz \mathcal{I}_K(f)(x,y,z) \\ &= \sum_{i,j,k=1}^{4} f(x_i,y_j,z_k) \int_0^{\Delta x} dx p_i(x) \int_0^{\Delta y} dy p_j(y) \int_0^{\Delta z} dz p_k(z).\end{aligned}$$

The four polynomial integrals can now be calculated exactly and yields

$$U_i := \int_0^{\Delta x} dx p_i(x) = \frac{\Delta x}{8} \left[1,3,3,1\right]_i.$$

Thus, a simple set of additions, sums and evaluations of f are required to estimate the integral of f on one finite element:

$$\Delta V_K = \sum_{i,j,k=1}^{4} f(x_i,y_j,z_k) U_i U_j U_k,$$

where

$$\begin{aligned}x_i &= x_0 + \frac{i-1}{3}\Delta x, \\ y_j &= y_0 + \frac{j-1}{3}\Delta y, \\ z_k &= z_0 + \frac{k-1}{3}\Delta z,\end{aligned}$$

and (x_0, y_0, z_0) is the lower left corner of the element domain K. This is referred to as a third degree Newton-Cotes formula or equivalently Simpson's 3/8 rule. Algorithm 1 shows a complete

Algorithm 1 Third degree Newton-Cotes integration, equal space mesh

input : $f(x, y, z)$, $a_x, b_x, a_y, b_y, a_z, b_z$, Num. mesh points in one dir: N
output : $I \approx \int_{a_x}^{b_x} dx \int_{a_y}^{b_y} dy \int_{a_z}^{b_z} dz f(x, y, z)$

$I := 0;$
$\Delta x := [b_x - a_x]/N;$
$\Delta y := [b_y - a_y]/N;$
$\Delta z := [b_z - a_z]/N;$
for $l = 0$ **to** $N - 1$ **do**
  for $m = 0$ **to** $N - 1$ **do**
    for $n = 0$ **to** $N - 1$ **do**
      $x := a_x + \Delta x \cdot l;$
      $y := a_y + \Delta y \cdot m;$
      $z := a_z + \Delta z \cdot n;$
      $I := I + \sum_{i,j,k=1}^{4} f(x + \frac{i}{3}\Delta x, y + \frac{j}{3}\Delta y, z + \frac{k}{3}\Delta z) U_i U_j U_k;$

numerical integration of f.

This algorithm works well for smooth functions that are well behaved at the integration boundaries. However, if the boundaries are singular then N needs to be large which results in a slowly converging integration. This can be remedied by calculating an upper error bound for each finite element calculation and refine the mesh where the error is large. Thus, the mesh will be refined around the singular boundaries since the error will be larger there.

2.1.2 Double exponential integration

The Newton-Cotes integration technique attempts to approximate the integrand by polynomial interpolation. For smooth and well behaved integrands this will, in most cases, lead to a better and better approximation as the mesh is refined. There exist other approaches to approximating the integration. One such approach is the double exponential integration technique. The following treatment of this technique is based on references [50] and [51].

Let

$$I = \int_a^b dx f(x)$$

be a one dimensional integral. First, perform an affine transformation $v = c_1 x + c_2$ in order to move the integration boundaries to ± 1. This is achieved by solving

$$\begin{aligned} -1 &= c_1 a + c_2 \\ 1 &= c_1 b + c_2, \end{aligned}$$

which yields the transformation

$$v = \frac{1}{b-a}[2x - (b+a)] \Leftrightarrow x = \frac{b-a}{2}\left[v + \frac{b+a}{b-a}\right].$$

A change of variables according to this transformation leads to

$$I = \frac{b-a}{2}\int_{-1}^{1} dv f\left(\frac{b-a}{2}\left[v + \frac{b+a}{b-a}\right]\right) := \int_{-1}^{1} dv g(v),$$

where

$$g(v) := \frac{b-a}{2} f\left(\frac{b-a}{2}\left[v + \frac{b+a}{b-a}\right]\right).$$

Note that this transformation can also be done numerically as part of an integration algorithm.

Next, perform another change of variables $v = \alpha(t)$, such that

$$I = \int_{\alpha^{-1}(-1)}^{\alpha^{-1}(1)} dt g(\alpha(t))\alpha'(t).$$

The main idea of the double exponential algorithm is to require that $\alpha^{-1}(-1) \to -\infty$, $\alpha^{-1}(1) \to \infty$ and that

$$g(\alpha(t))\alpha'(t) \sim e^{-ce^{|t|}}$$

as $|t| \to \infty$ (i.e the integrand should behave as a double exponential in the limit of large t).

The transformed integral can be very well approximated using a first degree Newton-Cotes formula (also referred to as the *trapezoid rule*) with some fixed mesh distance Δt. This is due to the double exponential behavior of the integrand. To this end, let $\mathbf{U} = \frac{\Delta t}{2}[1, 1]$. Then, the resulting volume elements are approximated by

$$\Delta V_K = \sum_{i=0}^{1} f(t_i)U_i = \frac{\Delta t}{2}\sum_{i=0}^{1} f(t_i),$$

such that a complete integral is approximated by

$$I = \int_a^b dt g(t) \approx \frac{\Delta t}{2}\sum_{j=0}^{N-1}\sum_{i=0}^{1} f([j+i]\Delta t) = \Delta t\left[\frac{1}{2}f(0) + \sum_{j=1}^{N-1} f(j\Delta t) + \frac{1}{2}f(N\Delta t)\right].$$

In the case where the integral boundaries tend towards $\pm\infty$, the sum becomes an infinite sum for some fixed Δt. The sum can be chosen to range from $i = -\infty$ to $i = \infty$. Since the integrand in this case behaves as a double exponential, the two terms of $1/2f(0)$ and $1/2f(N\Delta t)$ become negligible and drop out. Furthermore, the double exponential behavior results in a very short range of terms in the sum that are non vanishing (to machine precision). Therefore, the infinite sum can be terminated at finite N. Thus,

$$I = \int_{-\infty}^{\infty} dt g(\alpha(t))\alpha'(t) \approx \sum_{i=-N}^{N} \Delta t g(\alpha(t_i))\alpha'(t_i).$$

There are two errors associated with this method. One stems from using the trapezoid rule. This error is however very small for the particular case where the integrand behaves as a double exponential. The second error comes from terminating the sum at a finite index N. To minimize the error, Δt must be optimized for the particular choice of $\alpha(t)$. A popular choice for $\alpha(t)$ when the integral boundaries run from -1 to 1 is

$$\alpha(t) = \tanh\left(\frac{\pi}{2}\sinh t\right) \Rightarrow \alpha'(t) = \frac{\pi}{2}\frac{\cosh t}{\cosh^2(\pi/2\sinh t)}.$$

For this choice it turns out that the smallest error is achieved by letting

$$\Delta t \sim \frac{\ln(2\pi N\omega/c)}{N},$$

where ω is the distance from the real axis to the nearest singularity of the integrand. In algorithm 2 the double exponential algorithm has been extended to three dimensions. Note that all necessary values of both $\alpha(t)$ and $\alpha'(t)$ can be calculated a priori for given value of N.

Algorithm 2 Double exponential algorithm

input : $g(u,v,w)$, Half num. mesh points in one dir: N
output : $I \approx \int_{-1}^{1} du \int_{-1}^{1} dv \int_{-1}^{1} dw g(u,v,w)$

$I := 0$;
$\Delta := \ln[4N]/N$;
$\alpha(t) := \tanh(\pi/2 \sinh t)$;
for $l = -N$ **to** $N-1$ **do**
  for $m = -N$ **to** $N-1$ **do**
    for $n = -N$ **to** $N-1$ **do**
      $I := I + g(\alpha(l\Delta), \alpha(m\Delta), \alpha(n\Delta))\alpha'(l\Delta)\alpha'(m\Delta)\alpha'(n\Delta)\Delta^3$;

2.1.3 Duffy transformations of Coulomb integrals

The singular behavior of a Coloumb integral is difficult to handle directly by numerical routines. This can be alleviated by rotating the singularities of the Coloumb potential, such that they align parallel to one of the axis. The Duffy coordinate transformation does exactly that, without deforming the integration domain.

A Coulomb integral has the form

$$I = \int_0^{2\pi} d\theta \int_0^1 dx \int_0^1 dy \frac{F(x,y,\theta)}{\sqrt{y^2 - 2xyQ\cos\theta + Q^2x^2}},$$

where the integral has first been transformed to spherical coordinates. The integration boundaries has been moved to the range $[0,1]$ by change of variables. Assuming F to be analytic, and thus without any singularities, the integral is singular when

$$y^2 - 2xyQ\cos\theta + Q^2x^2 = 0.$$

This equation is solved by

$$y = Qx[\cos\theta \pm \sqrt{\cos^2\theta - 1}].$$

Since y is a real solution only if $\theta \in \{0, \pi, 2\pi, \dots\}$ the singularities occur at $\theta = 0$ and $\theta = 2\pi$ along the line $y = Qx$ (note that, due to the integration boundaries $x, y \geq 0$). These are difficult singularities to handle numerically. In order to make numerical integration simpler, it is desirable to have all singularities placed at the integration boundaries such that the integrand is smooth in the interior of the integration domain.

In order to rotate the singularities along $y = Qx$ in such a way that the singularities instead lies along the x-axis, a Duffy coordinate transformation [52] is applied. Note that it is not simply a matter of an affine transformation since this would only lead to a more complicated integration domain. A Duffy coordinate transformation preserve the cuboid like integration domain while rotating the singularities. The first step in the Duffy transformation is to divide the y integral into two parts

$$I_1 := \int_0^{2\pi} d\theta \int_0^1 dx \int_0^x dy \frac{F(x,y,\theta)}{\sqrt{y^2 - 2xyQ\cos\theta + Q^2x^2}}$$
$$I_2 := \int_0^{2\pi} d\theta \int_0^1 dx \int_x^1 dy \frac{F(x,y,\theta)}{\sqrt{y^2 - 2xyQ\cos\theta + Q^2x^2}},$$

where $I = I_1 + I_2$. Apply the change of variables $y = xy'$ in I_1 to find

$$I_1 = \int_0^{2\pi} d\theta \int_0^1 dx \int_0^1 dy' \frac{F(x,xy',\theta)}{\sqrt{y'^2 - 2y'Q\cos\theta + Q^2}}.$$

Notice that the singularities of I_1 are independent of x and is located at $y' = Q$ for $\theta = 0$ or $\theta = 2\pi$. In integral I_2 one can change the integration order of the x and y integration. The effect of this is shown in figure 2.4.

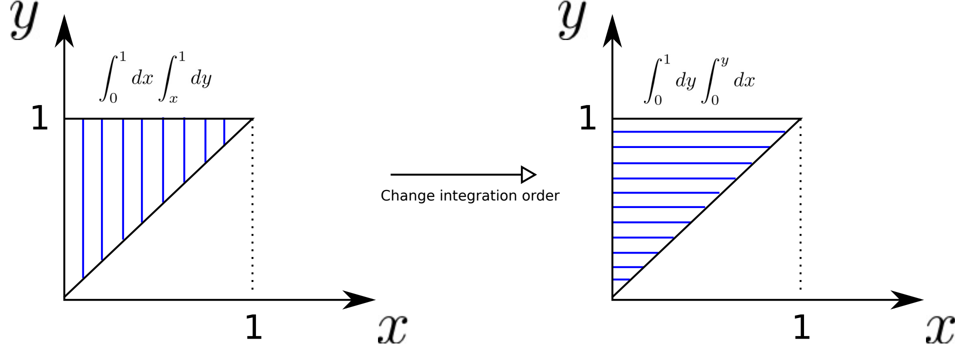


Figure 2.4: Effect of changing integration order.

After changing the integration order, rename $x \leftrightarrow y$. This yields

$$I_2 = \int_0^{2\pi} d\theta \int_0^1 dx \int_0^x dy \frac{F(y, x, \theta)}{\sqrt{x^2 - 2xyQ\cos\theta + Q^2y^2}}.$$

A change variables $y = xy'$ results in

$$I_2 = \int_0^{2\pi} d\theta \int_0^1 dx \int_0^1 dy' \frac{F(xy', x, \theta)}{\sqrt{1 - 2y'Q\cos\theta + Q^2y'^2}}.$$

Dropping the prime on y gives the complete Duffy coordinate transformation

$$\int_0^{2\pi} d\theta \int_0^1 dx \int_0^1 dy \frac{F(x, y, \theta)}{\sqrt{y^2 - 2xyQ\cos\theta + Q^2x^2}}$$
$$= \int_0^{2\pi} d\theta \int_0^1 dx \int_0^1 dy \left[\frac{F(x, xy, \theta)}{\sqrt{y^2 - 2yQ\cos\theta + Q^2}} + \frac{F(xy, x, \theta)}{\sqrt{1 - 2yQ\cos\theta + Q^2y^2}} \right].$$

Note that after a Duffy transformation all singularities have been rotated to $y = Q$ and $y = 1/Q$ and are independent of x, while the cuboid integration domain is in tact.

2.2 Numerical minimization

2.2.1 Minimization by second order interpolation

In this section a simple method of approximating a minimum numerically will be explored. The first step in finding a minimum involves bracketing the minimum. This refers to finding three samples a, b and c such that $a < b < c$ where $f(b) < f(a)$ and $f(b) < f(c)$ for all points b in the range $[a, c]$. The following minimization algorithm requires a priori knowledge of

- approximately where the requested minimum of the graph is located (i.e we know where to start searching)
- the samples of the function being close enough to bracket the minimum

- the original function being well approximated by a second order polynomial between the three points of the bracketing (this is true if the function is smooth and the sample points are sufficiently close)

Let $f(x_i)$ denote the discretization of $f(x)$. The first part of the algorithm searches for the coordinate x_2 where $f(x_i)$ has its lowest value, in a range where the minimum is known to be. The two coordinates $(x_1, f(x_1))$ and $(x_3, f(x_3))$ closest to x_2 (where $x_1 < x_2 < x_3$) is recorded. Then, a second order polynomial $g(x) = ax^2 + bx + c$ is fit through these three points. This is illustrated in figure 2.5.

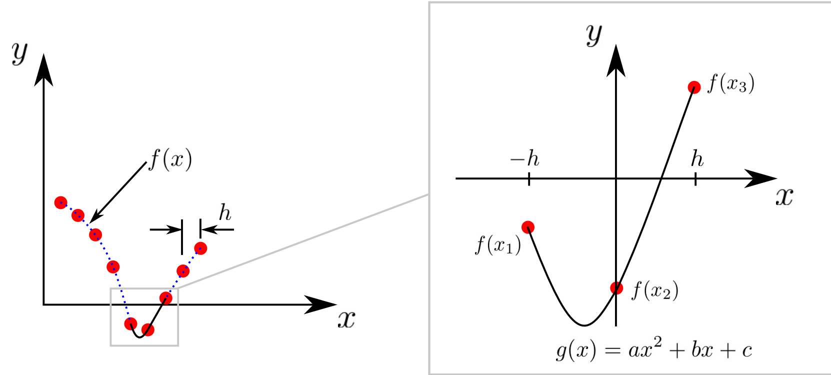


Figure 2.5: Numerical minimization.

The coefficients are found by solving

$$\begin{aligned} g(-h) &= ah^2 - bh + c = f(x_1), \\ g(0) &= c = f(x_2), \\ g(h) &= ah^2 + bh + c = f(x_3), \end{aligned}$$

which leads to

$$\begin{aligned} a &= \frac{1}{2h^2}[f(x_1) - 2f(x_2) + f(x_3)], \\ b &= \frac{1}{2h}[f(x_3) - f(x_1)], \\ c &= f(x_c). \end{aligned}$$

The minimum can now be approximated by solving $g'(x) = 0$. This yields

$$f_{min} \approx ax_m^2 + bx_m + c,$$

where

$$x_m = -\frac{b}{2a}.$$

Algorithm 3 shows an implementation of the method described above.

2.3 Numerical diagonalization

2.3.1 Jacobi diagonalization

There are several algorithms available for diagonalizing matrices. The Jacobi diagonalization is appropriate for small matrices and it is conceptually easy to understand. In the following, this

Algorithm 3 Algorithm for finding a minimum by second order interpolation

```
input :     f(x_i) = {f(x_1), f(x_2), ..., f(x_n)}, h := x_{i+1} - x_i
output :    f_min ≈ min{f(x)}

i_min := 0;
f_min := large value;
for i = 0 to n - 1 do
    if f(x_i) < f_min then
        f_min := f(i);
        i_min := i;

if i_min > 0 and i_min < n - 2 then
    a := 1/(2h^2)[f(i_min - 1) - 2f(i_min) + f(i_min + 1)];
    b := 1/(2h)[f(i_min + 1) - f(i_min - 1)];
    c := f(i_min);
    if a = 0 then
        f_min := f(i_min);
    else
        x_m := -b/(2a);
        f_min := a x_m^2 + b x_m + c;
else
    f_min = f(i_min);
```

diagonalization algorithm will be explained based on the derivation of reference [51]. The Jacobi algorithm is an iterative algorithm, which means that it iterates until reaching the desired accuracy.

Let A be a real symmetric matrix to be diagonalized. Then in one step (i.e. one iteration)

$$A' = P_{pq}^T A P_{pq},$$

where P_{pq} is a matrix designed to decrease the off diagonal elements A_{pq} and A_{qp} and store the results in A'. Subsequently, a complete sweep is performed for all off diagonal elements p, q of A. The Jacobi matrix is orthogonal and is defined as

$$P_{pq} := \begin{pmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{pmatrix},$$

where $s := \sin\phi$ and $c := \cos\phi$. Since only part of the Jacobi matrix is non-zero then only part of A' changes compared to A during a matrix multiplication. Performing the matrix multiplications leads to

$$a'_{rp} = ca_{rp} - sa_{rq} \text{ , for } r \neq q, r \neq p, \tag{2.1}$$

$$a'_{rq} = ca_{rq} + sa_{rp} \text{ , for } r \neq q, r \neq p, \tag{2.2}$$

$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq}, \tag{2.3}$$

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq}, \tag{2.4}$$

$$a'_{pq} = [c^2 - s^2]a_{pq} + sc[a_{pp} - a_{qq}], \tag{2.5}$$

while the remaining entries of A' are equal to the entries of A. The idea of the Jacobi diagonalization algorithm is to calculate an angle ϕ such that $a'_{pq} = 0$ for all p, q in the off diagonal of the matrix. However, one should not expect all off diagonal elements to vanish in one sweep. This is because the requirement that $a'_{pq} = 0$ for one entry can lead to an already zero off diagonal element becoming non-zero. It will however be shown that after several sweeps the off diagonal values will converge to zero. The algorithm can then be terminated once the off diagonal elements are zero, to machine precision.

Let $t := s/c$. Setting $a'_{pq} = 0$ leads to

$$[c^2 - s^2]a_{pq} + sc[a_{pp} - a_{qq}] = 0.$$

By dividing both sides with c^2 yields

$$t^2 + t\theta - 1 = 0,$$

where

$$\theta := \frac{a_{qq} - a_{pp}}{2a_{pq}}.$$

Solving the quadratic equation and choosing the root where $|t|$ is smallest gives

$$t = -\theta + \sqrt{\theta^2 + 1}$$

for $\theta > 0$ and

$$t = -\theta - \sqrt{\theta^2 + 1}$$

for $\theta \leq 0$. The angle θ can reach large enough values to cause an overflow of t when calculating it numerically since a_{pq} converges to zero. To alleviate this problem t can be rewritten. For $\theta > 0$

$$t = \frac{[-\theta + \sqrt{\theta^2 + 1}][-\theta - \sqrt{\theta^2 + 1}]}{-\theta - \sqrt{\theta^2 + 1}} = \frac{1}{\theta + \sqrt{\theta^2 + 1}},$$

and for $\theta \leq 0$

$$t = \frac{[|\theta| - \sqrt{\theta^2 + 1}][|\theta| + \sqrt{\theta^2 + 1}]}{|\theta| + \sqrt{\theta^2 + 1}} = \frac{-1}{|\theta| + \sqrt{\theta^2 + 1}}.$$

In a more compact form one finds that

$$t = \frac{\operatorname{sgn}\theta}{|\theta| + \sqrt{\theta^2 + 1}}.$$

For $|\theta| \gg 1$ this should be written as

$$t \approx \frac{\operatorname{sgn}\theta}{2|\theta|} = \frac{1}{2\theta}.$$

Thus, t is calculated in terms of the components of A such that $a'_{pq} = 0$. However, c and s have not yet been calculated. To calculate c one can use the trigonometric identity

$$t^2 + 1 = \frac{1}{c^2} \Rightarrow c = \frac{1}{\sqrt{t^2 + 1}}.$$

Now, $s = tc$ and all quantities s, c and t have been calculated in terms of the components of A which completely determines the Jacobi matrix rotation P_{pq}.

An implementation of the Jacobi algorithm is easier if every new matrix element is written in the form $a' = a + \delta$. Thus, equation (2.5) together with $a'_{pq} = 0$ yields

$$a_{qq} = \frac{c^2 - s^2}{sc}a_{pq} + a_{pp}.$$

Inserting this result into equation (2.3) gives

$$a'_{pp} = a_{pp} + \frac{s^2[c^2 - s^2] - 2s^2c^2}{sc} a_{pq} = a_{pp} - t a_{pq}.$$

Similarly, a_{pp} can be eliminated from equation (2.4) resulting in

$$a'_{qq} = a_{qq} + t a_{pq}.$$

Equation (2.1) is rewritten by adding and subtracting a_{rp}, which yields

$$a'_{rp} = a_{rp} - s\left[\frac{1-c}{s} a_{rp} + a_{rq}\right].$$

Note that

$$\frac{1-c}{s} = \frac{[1-c][1+c]}{s[1+c]} = \frac{s}{1+c} =: \tau.$$

Thus,

$$a'_{rp} = a_{rp} - s[a_{rq} + \tau a_{rp}].$$

Similarly, by adding and subtracting a_{rq} to equation (2.2) one finds that

$$a'_{rq} = a_{rq} + s[a_{rp} - \tau a_{rq}].$$

Let

$$S := \sum_{r \neq s} |a_{rs}|^2.$$

Then, by noting that equations (2.1) and (2.2) contain the only changes to off diagonal elements

$$\delta := S' - S = \sum_{r\neq p} |a'_{rp}|^2 + \sum_{r\neq q} |a'_{rq}|^2 - \sum_{r\neq p} |a_{rp}|^2 - \sum_{r\neq q} |a_{rq}|^2.$$

Using that $a'_{pq} = a'_{qp} = 0$ one finds

$$\begin{aligned}
\delta :&= \sum_{r\notin\{p,q\}} \left\{|a'_{rp}|^2 + |a'_{rq}|^2 - |a_{rp}|^2 - |a_{rq}|^2\right\} - 2|a_{pq}|^2 \\
&= \sum_{r\notin\{p,q\}} \left\{(ca_{rp} - sa_{rq})^2 + (ca_{rq} + sa_{rp})^2 - |a_{rp}|^2 - |a_{rq}|^2\right\} - 2|a_{pq}|^2 \\
&= -2|a_{pq}|^2.
\end{aligned}$$

Thus, since $S' < S$ for every Jacobi rotation, the off diagonal elements reduce for every rotation. Furthermore, both S and S' are bounded from above by zero. This proves that the Jacobi algorithm converges and that

$$P = \prod_{i=1}^{N} \prod_{p\neq q} P^i_{pq}$$

approximately diagonalizes A for sufficiently large N. By letting the algorithm terminate for $S = 0$ the accuracy will be in the order of machine precision.

Algorithm 5 together with algorithm 4 performs a single rotation on A and returns A' while algorithm 6 together with algorithm 4 performs a single rotation of P resulting in P'. Algorithm 7 shows a complete implementation of a Jacobi diagonalization. In this algorithm, after four sweeps, an off diagonal element D_{pq} is set to zero if it is small compared to the diagonal elements D_{pp} and

Algorithm 4 Partial Jacobi rotation at index r.

procedure :	$rot(Q, r, s, \tau, p, q, sym)$
input :	Unrotated matrix Q, index r, $s := \sin\phi$, τ, index p and q, sym : true $\Rightarrow$ symmetric entries are set equal
output :	$Q :=$ partial rotation of Q at index r

$Q_{rp} := Q_{rp} - s[Q_{rq} + \tau Q_{rp}]$;
$Q_{rq} := Q_{rp} + s[Q_{rp} - \tau Q_{rq}]$;
if $sym = true$ **then**
    $Q_{pr} := Q_{rp}$;
    $Q_{qr} := Q_{rq}$;

Algorithm 5 Single Jacobi rotation of a Jacobi matrix A.

procedure :	$rotA(A, t, s, \tau, p, q, n)$
input :	Unrotated matrix A, $t = s/c$, $s := \sin\phi$, $c := \cos\phi$, τ, index p and q, matrix size $n \times n$
output :	$A :=$ single Jacobi rotation of A

$A_{pp} := A_{pp} - tA_{pq}$;
$A_{qq} := A_{qq} + tA_{pq}$;
for $r := 0$ **to** $p - 1$ **do**
    $rot(A, r, s, \tau, p, q, true)$;
for $r := p + 1$ **to** $q - 1$ **do**
    $rot(A, r, s, \tau, p, q, true)$;
for $r := q + 1$ **to** $n - 1$ **do**
    $rot(A, r, s, \tau, p, q, true)$;
$A_{pq} := A_{qp} = 0$;

Algorithm 6 Single Jacobi rotation of a Jacobi matrix P.

procedure :	$rotP(P, s, \tau, p, q, n)$
input :	Unrotated matrix P, $s := \sin\phi$, τ, index p and q, matrix size $n \times n$
output :	$P :=$ single Jacobi rotation of P

for $r := 0$ **to** $n - 1$ **do**
    $rot(P, r, s, \tau, p, q, false)$;

Algorithm 7 Jacobi diagonalization of a real symmetric matrix.

input : Real symmetric matrix A, Max iterations N, Matrix size $n \times n$
output : Diagonalization matrix P, Diagonalized matrix D

```
D := A; P = 1;
for i = 0 to N − 1 do
    for p := 0 to n − 2 do
        for q := p + 1 to n − 1 do
            if i > 4 and D_pq < βD_pp and D_pq < βD_qq then
                D_pq := D_qp := 0;
            if D_pq ≠ 0 then
                θ := [D_qq − D_pp]/[2D_pq];
                if |θ| ≫ 1 then
                    t := 1/[2θ];
                else
                    t := sgn θ/[|θ| + sqrt(θ^2 + 1)];

                c := 1/sqrt(t^2 + 1);
                s := t · c;
                τ := s/[c + 1];

                rotA(D, t, s, τ, p, q, n);
                rotP(P, s, τ, p, q, n);
    S := Σ_{i>j} D_ij^2;
    if S = 0 then
        terminate with success;
terminate with failure;
```

Algorithm 8 Insertion sort of eigenvalues and eigenvectors.

input : eigenvalues D and eigenvectors P, matrix size $n \times n$
output : D sorted in descending order, P sorted in same order as D

```
for i := 0 to n − 2 do
    α := D_ii;
    i_T := i
    for j := i to j < n − 1 do
        if D_jj ≥ α then
            α := D_jj;
            i_T := j
    if i ≠ i_T then
        D_{i_T i_T} := D_ii;
        D_ii := α;
        for j := 0 to j < n − 1 do
            α := P_{j i_T};
            P_{j i_T} := P_ji;
            P_ji := α;
```

D_{qq}. This will make the algorithm converge faster. How small D_{pq} must be compared to the off diagonal elements is adjusted by the constant β. In what follows $\beta = 10^{-20}$.
The eigenvalues of D together with the eigenvectors in P can be sorted in descending order by using insertion sort. Algorithm 8 shows how to implement the sort operation.

The Jacobi diagonalization algorithm can be extended to diagonalization of Hermitian matrices. To accomplish this, write the complex Hermitian matrix as $C := A + iB$ where A and B are real but not necessarily symmetric matrices. Let $\mathbf{z} := \mathbf{x} + i\mathbf{y}$ where $x_i, y_i \in \mathbb{R}$. Then the eigenvalue problem is stated as $C\mathbf{z} = \lambda\mathbf{z}$, where $\lambda \in \mathbb{R}$ is an eigenvalue. Expanding the eigenvalue problem in terms of A, B and x, y yields

$$[A\mathbf{x} - B\mathbf{y}] + i[B\mathbf{x} + A\mathbf{y}] = \lambda\mathbf{x} + i\lambda\mathbf{y}.$$

Solving the above complex equation is equivalent to solving the set of equations

$$\begin{aligned} A\mathbf{x} - B\mathbf{y} &= \lambda\mathbf{x} \\ B\mathbf{x} + A\mathbf{y} &= \lambda\mathbf{y}, \end{aligned}$$

which is equivalent to solving the real eigenvalue problem

$$C_{2n}\mathbf{z}_{2n} := \begin{pmatrix} A & -B \\ B & A \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}.$$

Now note that since

$$C^\dagger = A^T - iB^T = C = A + iB,$$

then $B = -B^T$. Thus, $C_{2n}^T = C_{2n}$, and the equivalent $2n \times 2n$ eigenvalue problem is not only real but also symmetric. This means that the already stated Jacobi algorithm can be used directly to solve the eigenvalue problem of a Hermitian matrix by instead stating the initial $n \times n$ problem as an equivalent $2n \times 2n$ problem.

The $2n \times 2n$ problem will yield $2n$ eigenvalues and eigenvectors. Let $\mathbf{u} + i\mathbf{v}$ be an eigenstate and λ_q be an eigenvalue of C, then

$$[A + iB][\mathbf{u} + i\mathbf{v}] = \lambda_q[\mathbf{u} + i\mathbf{v}].$$

Furthermore, the eigenvalue of the vector $[-\mathbf{v}, \mathbf{u}]$ can be calculated by

$$[A + iB][-\mathbf{v} + i\mathbf{u}] = [A + iB]i[\mathbf{u} + i\mathbf{v}] = \lambda_q i[\mathbf{u} + i\mathbf{v}].$$

Thus, the eigenvalue corresponding to the eigenvector $[-\mathbf{v}, \mathbf{u}]$ is λ_q. This shows that there are only n distinct eigenvalues resulting from solving the $2n \times 2n$ problem. By using algorithm 8 on the results of the $2n \times 2n$ problem, one only needs to pick every second eigenvalue and choose one of the corresponding n eigenvectors.

Algorithm 9 shows how to choose one of the corresponding eigenvectors and how to build the complex diagonalization matrix $\mathcal{P}$ that would result from the $n \times n$ problem. In this algorithm, the eigenvector with the largest first real entry will be selected.

Consider a Hermitian $n \times n$ matrix function $C(x)$, where $x \in K \subset \mathbb{R}^m$. Then, for each x in the domain, a diagonalization of $C(x)$ will lead to a complex diagonalizing matrix $\mathcal{P}(x)$. However, since

$$R := \begin{pmatrix} e^{i\theta_1} & 0 & \cdots & 0 & 0 \\ 0 & \ddots & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & e^{i\theta_n} \end{pmatrix}$$

Algorithm 9 Assembling complex $n \times n$ matrices $\mathcal{D}, \mathcal{P}$ from $2n \times 2n$ real matrices D, P.

input : $2n \times 2n$ diagonal matrix D and real diagonalizing matrix P
output : $n \times n$ diagonal matrix $\mathcal{D}$ and complex diagonalizing matrix $\mathcal{P}$

for $i := 0$ **to** $n-1$ **do**
  if $P_{0,2i} > P_{0,2i+1}$ **then** $f_i := 2i$; **else** $f_i := 2i+1$;
  $\mathcal{D}_{ii} := D_{f_i,f_i}$;
for $j := 0$ **to** $n-1$ **do**
  for $i := 0$ **to** $n-1$ **do** $\mathcal{P}_{ij} := P_{i,f_i}$;
  for $i := n$ **to** $2n-1$ **do** $\mathcal{P}_{i-n,j} := P_{i,f_i}$;

satisfies $RR^\dagger = \mathbf{1}$ and R commutes with any diagonal matrix D, then

$$A = \mathcal{P}\mathcal{D}\mathcal{P}^\dagger = \mathcal{P}RR^\dagger\mathcal{D}\mathcal{P}^\dagger = (\mathcal{P}R)\mathcal{D}(\mathcal{P}R)^\dagger.$$

This means that also $\mathcal{P}R$ diagonalizes A. When using the Jacobi diagonalization, θ_i is very sensitive to the numerical values of the matrix to be diagonalized. Therefore, the real and complex parts of the diagonalizing matrices does not separately form continuous matrix functions, even if the matrix to be diagonalized has continuous real and complex elements. It is possible to alleviate the problem by using the degrees of freedom in R to rotate all entries in the first row of $\mathcal{P}$ until they become real. Once the entries are real, a second rotation can be done to make sure all elements are real and also positive. Note that the modulus of the elements of $\mathcal{P}$ does not suffer from the same discontinuities as for the corresponding real and complex parts.

2.4 Numerical function inverse

2.4.1 Function interpolation 1D

Consider the linear interpolation scheme in figure 2.6. The basis functions for the corresponding

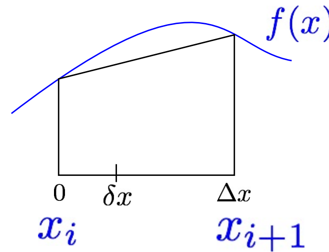


Figure 2.6: Linear interpolation of 1D function.

linear element are (see section 2.1.1 for derivation)

$$\phi_1 = -\frac{1}{\Delta x}[\delta x - \Delta x],$$
$$\phi_2 = \frac{1}{\Delta x}\delta x.$$

This results in the local interpolant

$$\mathcal{I}_{K_i} f := \sum_{j=1}^{2} \mathcal{N}_j(f)\phi_j = \sum_{j=1}^{2} f(x_{i+j-1})\phi_j.$$

Thus, for a function domain (a, b) divided into N parts

$$f(x) \approx \mathcal{I}_{K_i} f(x) = \frac{1}{\Delta x}[f(x_{i+1})\delta x - f(x_i)(\delta x - \Delta x)], \tag{2.6}$$

where $\delta x := x - [a + i\Delta x]$ and

$$\Delta x := \frac{b-a}{N} \ , \ i := \lfloor \frac{x-a}{\Delta x} \rfloor.$$

2.4.2 Function inverse by linear interpolation

A straight forward method of finding the inverse of a function $f(x)$, given a set of function values $\{f(x_i)\}$, could be to traverse the entire set searching for the value of $f(x_i)$ closest to some value f'. Then, the corresponding value of x_i would be an approximation of the inverse at f'. However, this would lead to an $\mathcal{O}(n)$ algorithm, where n is the number of elements in $\{f(x_i)\}$. It is possible to do better than that by using a binary search algorithm.

Consider a monotonic continuous function $f(x)$. The goal is to find the inverse $x' = f^{-1}(f')$. Instead of starting the search for f' from $i = 0$, start the search at the center $i = N/2$. Then, if $f' > f(x_i)$ the next point to test is centered between $N/2$ and N in the case that f is monotonically increasing, and centered between 0 and $N/2$ if f is monotonically decreasing. This strategy of continuously dividing the search area in half can be continued until the closest point has been found. Thus, all points in the set (of cardinality n) have been explored when $n = 2^m$ is satisfied, where m is the number of steps in the search. Thus, the binary search algorithm is of $\mathcal{O}(\ln_2 n)$. A typical cardinality of $\{f(x_i)\}$ could be 4096 or larger. Then the linear search algorithm is of order 4096 steps, while the binary search algorithm is of order 12 steps.

Once the point x_i closest to the inverse has been located, linear interpolation can be used to further approximate the inverse at f'. In order to achieve this, equation (2.6) is solved with respect to δx. This yields

$$\delta x \approx \frac{f' - f(x_i)}{f(x_{i+1}) - f(x_i)}\Delta x.$$

The chosen strategy will be to terminate the algorithm once x_i has been found such that $f' \in [f(x_{i-1}), f(x_i)]$. Thus, the above interpolation of δx must be rewritten to

$$\delta x \approx \frac{f' - f(x_{i-1})}{f(x_i) - f(x_{i-1})}\Delta x.$$

For a function f with domain (a, b) the interpolation of the inverse reads

$$x := a + [i-1]\Delta x + \delta x.$$

This can easily be seen from looking at figure 2.6. Algorithm 10 shows how to explicitly find i such that $f' \in [f(x_{i-1}), f(x_i)]$.

2.5 Numerical root search

2.5.1 Binary search for roots

Let f be a continuous function with domain $[0, b > E)$ such that $f < 0$ on $[0, E)$ and $f(E) = 0$ while $f(x) \geq 0$ for $x > E$. Then, it is possible to search for the root $x = E$ using a binary search algorithm. Examples of such functions are shown in figure 2.7. The principle of the algorithm is simple; start by the knowledge that the root lies between 0 and b. It is also known from the initial criteria of f, that the root in question is the first root starting from 0. Then, divide the range

Algorithm 10 Binary search for inverse, where inverse is interpolated.

input : $\{f(x_i)\}$ for $i = 1, 2, \cdots, N$, f', domain (a, b).
output : $\approx f^{-1}(f') =: x$.

$i := 0$; $j := [N-1]$ div 2; $k := N-1$;
if $f_1 > f_0$ **then**
  if $f' < f_0$ **or** $f' > f_k$ **then** exit with failure;
  while not $\{f_{j-1} \leq f'$ **and** $f' \leq f_j\}$ **do**
    if $f_j < f'$ **then** $i := j$; **else** $k := j$;
    $j := [i+k]$ div 2;
else if $f_1 < f_0$ **then**
  if $f' > f_0$ **or** $f' < f_k$ **then** exit with failure;
  while not $\{f_{j-1} \geq f'$ **and** $f' \geq f_j\}$ **do**
    if $f_j > f'$ **then** $i := j$; **else** $k := j$;
    $j := [i+k]$ div 2;
else
  exit with failure;

$x := a + \Delta x[j-1] + \Delta x[f' - f_{j-1}]/[f_j - f_{j-1}]$;

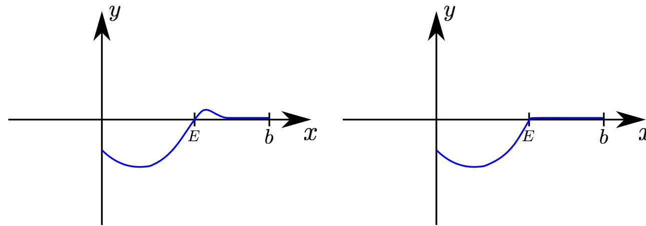


Figure 2.7: Examples of functions satisfying criteria for root searching algorithm.

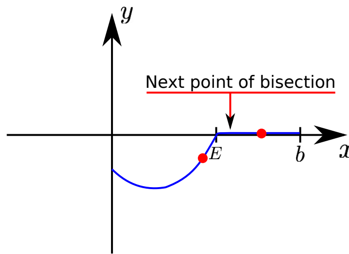


Figure 2.8: Problem with next point of bisection.

in half and subsequently find which half of the range contains the root. Continue in this manner until the required accuracy has been reached.

However, there are some subtleties. For example, consider the range where the root is located having been narrowed down to what is shown in figure 2.8. In that case, the next point of bisection of this range is positioned at a point where the function is zero. At the same time the right hand side of the range is also at a point where the function is zero. Thus, if the situation is not handled correctly, the next active range will be the right hand part of the currently active range. However, the root is located in the left hand part of the currently active range. Algorithm 11 shows how to implement the root search and avoid this problem. Note that $\Delta y_1 < 0$ for all steps in the

Algorithm 11 Binary root search.

input :	$f(x)$ at $[a,b]$. f cont., $f<0$ on $[0,E)$, $f(x)\geq 0$ for $x>E$, $f(E)=0$, number of iterations N.
output :	$\approx E$.

$x_1 := a$; $x_2 := b$;
$\Delta y_1 := f(a)$; $\Delta y_2 := f(b)$;
for $i := 0$ **to** $i < N-3$ **do**
  $x := [x_1 + x_2]/2$;
  $\Delta y := f(x)$;
  if $\Delta y < 0$ **then**
    $x_1 := x$; $\Delta y_1 := \Delta y$;
  else
    $x_2 := x$; $\Delta y_2 := \Delta y$;
$E := x$;

algorithm. This is easy to see, since by assumption $\Delta y_1 < 0$ in first step. In further steps, Δy_1 is only assigned Δy if $\Delta y < 0$. Thus, the root will always be located at some point to the right of x_1. Similarly, only $\Delta y \geq 0$ is assigned to Δy_2. Which shows that the root is always located in the range $(x_1, x_2]$ for all steps in the algorithm.

2.6 Ferromagnetic instability

2.6.1 The exchange integral

One of the major consequences of quantum mechanics is the fundamental indistinguishability of identical particles. In a two particle system transitioning from a state A to a state B, each particle will be in a superposition of all possible states between A and B. It is therefore impossible to distinguish identical particles in their initial and final states. Thus, a two particle state is described by

$$\psi(\mathbf{r}_1, \mathbf{r}_2) = \frac{1}{\sqrt{2}} \left\{ \psi_1(\mathbf{r}_1)\psi_2(\mathbf{r}_2) \pm \psi_2(\mathbf{r}_1)\psi_1(\mathbf{r}_2) \right\},$$

where ψ_1 and ψ_2 are two distinct states that the particles, labeled $\mathbf{r}_1$ and $\mathbf{r}_2$, can be in. An expectation value of an observable $\hat{\mathcal{O}}$ now takes the form

$$\begin{aligned} \langle \psi | \hat{\mathcal{O}} | \psi \rangle = \frac{1}{2} \int & d\mathbf{r}_1 d\mathbf{r}_2 \left\{ \psi_2^\dagger(\mathbf{r}_2)\psi_1^\dagger(\mathbf{r}_1)\hat{\mathcal{O}}\psi_1(\mathbf{r}_1)\psi_2(\mathbf{r}_2) + \psi_1^\dagger(\mathbf{r}_2)\psi_2^\dagger(\mathbf{r}_1)\hat{\mathcal{O}}\psi_2(\mathbf{r}_1)\psi_1(\mathbf{r}_2) \right\} \\ \pm \int & d\mathbf{r}_1 d\mathbf{r}_2 \psi_2^\dagger(\mathbf{r}_2)\psi_1^\dagger(\mathbf{r}_1)\hat{\mathcal{O}}\psi_2(\mathbf{r}_1)\psi_1(\mathbf{r}_2), \end{aligned} \tag{2.7}$$

where it is assumed that $\hat{\mathcal{O}}$ is Hermitian and therefore has a real expectation value. The last term in equation (2.7) is called the *exchange force* [53] if $\mathcal{O} = |\mathbf{r}_2 - \mathbf{r}_2|$ and the *exchange energy* [12] or *exchange integral* [54] if $\mathcal{O}$ is an interaction Hamiltonian. In general this will be refered to as an *exchange term.* The wave function for two distinguishable particles is described by

$$\psi_d(\mathbf{r}_1, \mathbf{r}_2) = \psi_1(\mathbf{r}_1)\psi_2(\mathbf{r}_2),$$

where it is a priori known that particle 1 is in state ψ_1 and particle 2 is in state ψ_2. It follows that

$$\langle \psi_d | \hat{\mathcal{O}} | \psi_d \rangle = \int d\mathbf{r}_1 d\mathbf{r}_2 \psi_2^\dagger(\mathbf{r}_2)\psi_1^\dagger(\mathbf{r}_1)\hat{\mathcal{O}}\psi_1(\mathbf{r}_1)\psi_2(\mathbf{r}_2)$$

for distinguishable particles. Comparing this with equation (2.7) shows that indistinguishable particles behave differently depending on the wave function symmetry under interchange of ψ_1

and ψ_2, while distinguishable particles do not. For an antisymmetric wave function the expectation value of $\hat{\mathcal{O}}$ is decreased by the exchange term. However, for a symmetric wave function the expectation value will be enhanced by the exchange term.

Following the treatment in references [54], [55] and [53]. A full wave function is described both by an orbital part and a spin part $\Psi = \chi\psi$. For fermions Ψ must be antisymmetric. In a two particle system the spins can align to form a spin-triplet state which is symmetric, or they can form a spin-singlet state which is antisymmetric. This means that if χ is symmetric then ψ must be antisymmetric, and if χ is antisymmetric then ψ must be symmetric. This means that in a spin-triplet state ψ is antisymmetric and the exchange term will lower the expectation value of $\hat{\mathcal{O}}$. Conversely, a spin-singlet state will enhance the expectation value of $\hat{\mathcal{O}}$.

For a system that can take a number of states $\{\psi_n\}$, all possible states must be summed over and the exchange integral takes the form [13]

$$E_{ex} = \pm \sum_{n,m} \int d\mathbf{r}_1 d\mathbf{r}_2 \psi_n^\dagger(\mathbf{r}_1)\psi_m^\dagger(\mathbf{r}_2)\hat{\mathcal{O}}\psi_m(\mathbf{r}_1)\psi_n(\mathbf{r}_2) := \pm \sum_{n,m} \langle n, m|\hat{\mathcal{O}}|m, n\rangle. \tag{2.8}$$

2.6.2 Ferromagnetic instability

Suppose $\hat{\mathcal{O}}$ is taken to be an interaction Hamiltonian, describing the interactions between the two particles. The expectation value then yields the corresponding interaction energy. It is then instructive to look at the difference in energy that emerges from a transition between a spin-triplet state and a spin-singlet state. Let ΔE_{ex} denote the difference in energy . Using equations (2.7) and (2.8) one finds that

$$\begin{aligned}\Delta E = \langle\psi|\hat{\mathcal{O}}|\psi\rangle_{trip} - \langle\psi|\hat{\mathcal{O}}|\psi\rangle_{sing} &= E_{ex,\text{trip}} - E_{ex,\text{sing}} = \Delta E_{ex} \\ &= -2\int d\mathbf{r}_1 d\mathbf{r}_2 \psi_2^\dagger(\mathbf{r}_2)\psi_1^\dagger(\mathbf{r}_1)\hat{\mathcal{H}}_I\psi_2(\mathbf{r}_1)\psi_1(\mathbf{r}_2),\end{aligned}$$

where for $E_{ex,trip}$ is calculated based on the antisymmetric spacial wave function and $E_{ex,sing}$ is calculated based on the symmetric spacial wave function (note that it is the spacial part of the wave function that enters the probability $|\psi|^2 d^3x_1 d^3x_2$). If $\Delta E_{ex} > 0$ then the spin-singlet state is favored and if $\Delta E_{ex} < 0$ the spin-triplet state is favored. Thus, the exchange energy plays an important role in determining ferromagnetic instabilities in a system due to the change in sign between a symmetric and an anti-symmetric state.

Although the purely quantum mechanical exchange energy plays a crucial role in determining ferromagnetic instabilities, it is of course not the only contributing factor. In fact, all phenomenon that yields a different energy in a polarized (i.e. majority of spins aligned) compared to an unpolarized state will play a role in determining if a system is ferromagnetic. Until now only a system of two particles have been explored. Things become slightly more complicated when looking at a many body system in an external potential V_{ext}. In that case, some electron pairs might be in a triplet combination and some in a singlet combinations while the complete system on average might be polarized or unpolarized. It will no longer be possible to separate out the single exchange term as in equation (2.8). Luckily, by using field theoretical formalisms such as second quantization or path integral formalism, the interaction terms of the many body Hamiltonian will automatically include the effects of exchange interactions as well other effects. The effects included will effectively depend on the order of perturbation.

From the above discussion it should now be clear that for a many body system, one can predict the ferromagnetic instabilities by comparing the total energy of a system in a hypothetical ferromagnetic state with the total energy of the system in a hypothetical paramagnetic state. The state of lowest energy will be the preferred state of the system. A natural place to start is the

single particle effects. This amounts to calculating the dispersions $\epsilon(\mathbf{k})$ of the systems and then integrating the energy times the density of states $\mathcal{D}$ up to the Fermi surface ϵ_F to find the total kinetic energy

$$\langle E_k \rangle = \int_0^{\epsilon_F} d\epsilon \epsilon \mathcal{D}(\epsilon)$$

of each system. The next step is to look at interaction effects. This is done by calculating $\langle \hat{H}_I \rangle$ where $\hat{H}_I$ is the interaction hamiltonian of the paramagnetic of ferromagnetic system. To first order in the interaction the propagator G is represented by [56]

$$G = \overset{\mathbf{q}=0 \quad \mathbf{p}}{\sim\!\sim\!\bigcirc} + \overset{\mathbf{p}}{\underset{\mathbf{q}}{\frown}} \; .$$

These terms can be recognized as the Hartree and Fock terms (also referred to as the direct and exchange contributions). In the Jellium model a constant positive background is assumed in order to approximate the effects of a more complicated lattice potential. This has as a consequence that $V(\mathbf{q} = 0) = 0$. Thus, in the Jellium model it can be seen that the Hartree contribution vanishes. The details of these types of calculations will be shown later. In the ABC-trilayer case, effects of higher order terms will also be considered.

It turns out that the effects of the exchange contribution and the effects of the kinetic energy are competing. This is illustrated in figure 2.9, which shows some dispersion $\epsilon(\mathbf{k})$ where the energy levels are filled with spin-up and spin-down electrons. Because of the Pauli exclusion principle the Fermi surface is lifted for the entire system and it will cost kinetic energy to transition from an unpolarized state to a polarized state.

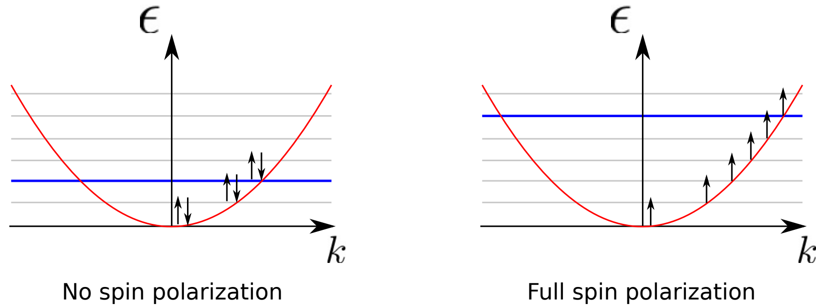


Figure 2.9: Illustration of the cost in kinetic energy by transitioning to a polarized state.

Because of the two fold degeneracy caused by spin in a fermionic system, it is appropriate to separate the band structure (and hence the dispersion) into a *spin up channel* and a *spin down channel*. When a system transitions from a paramagnetic phase to a ferromagnetic "spin-up" phase more spin-up electrons must fill the "spin-up" band. This is demonstrated in figure 2.10 for a system at some fermi energy ϵ_F in the paramagnetic phase (equal filling of spin-up and spin-down) undergoing a transition to a ferromagnetic phase, with more spins in the spin-up channel. This separation of the bands, into spin-up channels and spin-down channels will be used extensively in the coming chapters.

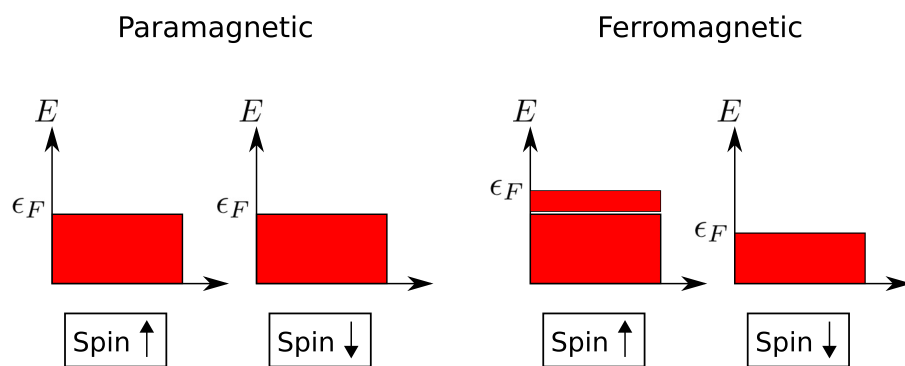


Figure 2.10: The dividing of bands into spin up channels and spin down channels.

CHAPTER 3

Monolayer graphene

This chapter is devoted to monolayer graphene. In the calculations that follow, it will be necessary to know the single particle dispersion of the electrons moving on the lattice potential, i.e. the band structure of graphene. From the dispersion it will be possible to derive the density of states for the system, which in turn is used to find the kinetic energy. The chapter therefore start by deriving the band structure. This will be done using two equivalent techniques, in which the second technique will be used in subsequent chapters.

The ferromagnetic properties of monolayer graphene will be calculated in the second part of this chapter. This is achieved by calculating the kinetic energy and the Coloumb interaction energy and comparing these energies in ferromagnetic and paramagnetic states of the system.

3.1 Band structure

In this section the band structure of single-layer Graphene is calculated following the work done in reference [1], reference [7]. For further information on methods related to second quantization one can consult reference [56].

3.1.1 The structure of Graphene

The *honeycomb lattice structure* of Graphene is shown in figure 3.1. Each lattice site consists of a Carbon atom with one free electron. The structure is divided into two sub-lattices A and B, where each sub-lattice by itself yields a triangular lattice. Let $\mathbf{a}_1$ and $\mathbf{a}_2$ denote the lattice translation vectors of the B sub-lattice and describe all vectors on the lattice in terms of the lattice parameter a as shown in figure 3.1. Since all edges of the honeycomb lattice has the length a, then the points connecting the B sites form an equilateral triangle; hence $\angle(\mathbf{a}_1, \mathbf{a}_2) = 60°$ and $\angle(\mathbf{a}_1, \hat{\mathbf{x}}) = 30°$. Therefore, the angle between $\boldsymbol{\delta}_3$ and $-\mathbf{a}_1$ must be 30°. Furthermore, since all sides in the honeycomb lattice are of equal length the angle between $\mathbf{a}_1$ and the segment BA (i.e. the segment making the smallest angle with $\mathbf{a}_1$) must also be 30°. Thus,

$$\frac{1}{2}|\mathbf{a}_1| = a\cos 30° = \frac{a\sqrt{3}}{2} \Leftrightarrow |\mathbf{a}_1| = |\mathbf{a}_2| = \sqrt{3}a,$$

$$\mathbf{a}_1 = |\mathbf{a}_1|\,(\cos 30°, \sin 30°) = \sqrt{3}a\left(\frac{\sqrt{3}}{2}, \frac{1}{2}\right) = \frac{a}{2}(3, \sqrt{3})$$

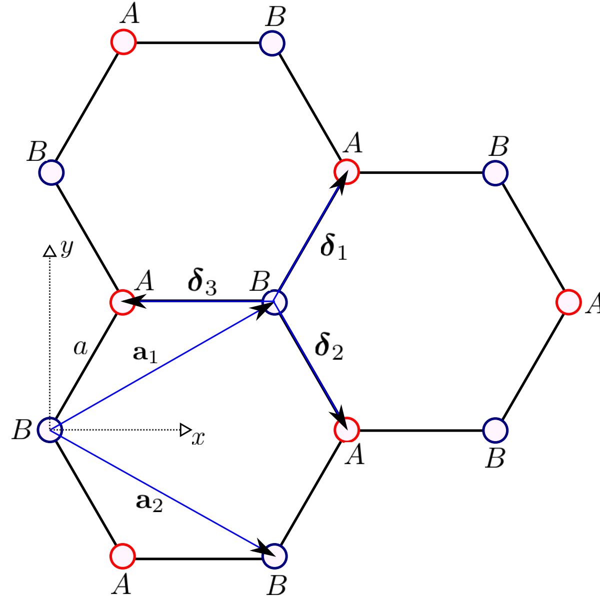


Figure 3.1: The honeycomb structure of Graphene.

and

$$\mathbf{a}_2 = \frac{a}{2}(3, -\sqrt{3}).$$

The reciprocal lattice vectors $\mathbf{b}_1$ and $\mathbf{b}_2$, by definition, satisfy

$$\mathbf{b}_i \cdot \mathbf{a}_j = 2\pi\delta_{ij}, \tag{3.1}$$

where the factor of 2π is due to choice of normalization. Equation (3.1) forms the set of linear equations:

$$\frac{a}{2}\begin{pmatrix} 3 & \sqrt{3} & 0 & 0 \\ 3 & -\sqrt{3} & 0 & 0 \\ 0 & 0 & 3 & \sqrt{3} \\ 0 & 0 & 3 & -\sqrt{3} \end{pmatrix} \begin{pmatrix} b_{1,x} \\ b_{1,y} \\ b_{2,x} \\ b_{2,y} \end{pmatrix} = \begin{pmatrix} 2\pi \\ 0 \\ 0 \\ 2\pi \end{pmatrix},$$

where $\mathbf{b}_i \equiv (b_{i,x}, b_{i,y})$. Solving the set of equations leads to

$$\mathbf{b}_1 = \frac{2\pi}{3a}(1, \sqrt{3}) \tag{3.2}$$

and

$$\mathbf{b}_2 = \frac{2\pi}{3a}(1, -\sqrt{3}). \tag{3.3}$$

The three nearest neighbor lattice translation vectors of the honeycomb lattice are:

$$\boldsymbol{\delta}_1 = a(\cos 60^\circ, \sin 60^\circ) = \frac{a}{2}(1, \sqrt{3}),$$

$$\boldsymbol{\delta}_2 = \frac{a}{2}(1, -\sqrt{3})$$

and

$$\boldsymbol{\delta}_3 = a(-1, 0).$$

It is easy to see that the distance between closest B sites is the next nearest neighbor distance. Thus,

$$\boldsymbol{\delta}_1^{\pm} \equiv \pm \mathbf{a}_1,$$
$$\boldsymbol{\delta}_2^{\pm} \equiv \pm \mathbf{a}_2,$$
$$\boldsymbol{\delta}_3^{\pm} \equiv \pm (\mathbf{a}_1 - \mathbf{a}_2),$$

are the six next nearest neighbor lattice translation vectors of the honeycomb lattice. The Brillouin zone of Graphene is described in appendix B.

3.1.2 Tight binding Hamiltonian

Consider using the tight binding approximation. Then, an electron on the lattice is assumed to be localized close to the Carbon atoms. Thus, the wave function ψ of an electron on the lattice can be approximated by a linear combination of single carbon atom wave functions ϕ. Mathematically this is written as

$$\psi_{\mathbf{k},\sigma}(\mathbf{r}) = \sum_{i=1}^{N} C_{i,\mathbf{k}} \phi_\sigma(\mathbf{r} - \mathbf{r}_i) = \frac{1}{\sqrt{N}} \sum_{i=1}^{N} \phi_\sigma(\mathbf{r} - \mathbf{r}_i) e^{i\mathbf{r}_i \cdot \mathbf{k}},$$

where σ denotes the spin and N denotes the total number of lattice sites. Since the Graphene honeycomb lattice is divided into two sub-lattices A and B, the corresponding single electron wave functions are

$$\psi^a_{\mathbf{k},\sigma}(\mathbf{r}) = \frac{1}{\sqrt{N^a}} \sum_{i=1}^{N^a} \phi^a_\sigma(\mathbf{r} - \mathbf{r}_i) e^{i\mathbf{r}_i \cdot \mathbf{k}} \text{ and } \psi^b_{\mathbf{k},\sigma}(\mathbf{r}) = \frac{1}{\sqrt{N^b}} \sum_{i=1}^{N^b} \phi^b_\sigma(\mathbf{r} - \mathbf{r}_i) e^{i\mathbf{r}_i \cdot \mathbf{k}}, \tag{3.4}$$

where N^a and N^b are the number of electrons on the A sub-lattice and the B sub-lattice respectively.

The Hamiltonian of the many body system, consisting of electrons on the honeycomb lattice, can be written as (with spin indices suppressed)

$$H = \sum_{i=1}^{N} \frac{\mathbf{k}_i^2}{2m} + \sum_{i,j=1}^{N} V(\mathbf{r}_i - \mathbf{r}_j) + \sum_{i=1}^{N} V_{ext}(\mathbf{r}_i), \tag{3.5}$$

where $\mathbf{k}_i$ is the momentum of each electron on the lattice. Furthermore, V is the potential between the electrons and V_{ext} is the potential of the Carbon atoms as a function of the electron positions $\mathbf{r}_i$.

In the tight binding approximation, the electrons are localized to the Carbon atoms and the average momentum of an electron is therefore small. Thus, the first term in equation (3.5) can be neglected in this approximation. The second term in equation (3.5) is neglected for now. The effects of this term will be treated separately as a perturbation to the system. Thus, to a good approximation

$$H = \sum_{i=1}^{N} V_{ext}(\mathbf{r}_i). \tag{3.6}$$

Going to second quantization, a Fock space is constructed which consist of states with spin σ at lattice site i on sub lattice $a \in \{A, B\}$ denoted by $|\mathbf{N}\rangle := |\{n^a_{i,\sigma}\}\rangle$. Since electrons are fermions $n^a_{i,\sigma} \in \{0, 1\}$. The ground state $|0\rangle$ represents a lattice with no free electrons. Two types of creation and annihilation operators are needed; one type for the A sub-lattice sites and another for the B sub-lattice sites. To this end, let $\hat{a}^\dagger_{i,\sigma}$ and $\hat{b}^\dagger_{j,\sigma}$ create an electron of spin σ on the A sub-lattice

site i and the B sub-lattice site j, respectively. The corresponding annihilation operators are $\hat{a}_{i,\sigma}$ and $\hat{b}_{j,\sigma}$. By construction, these operators satisfy

$$\{\hat{a}_{i,\sigma}, \hat{a}^{\dagger}_{j,\sigma'}\} = \delta_{ij}\delta_{\sigma\sigma'} \ , \ \{\hat{b}_{i,\sigma}, \hat{b}^{\dagger}_{j,\sigma'}\} = \delta_{ij}\delta_{\sigma\sigma'},$$

where all other anti-commutators vanish.

The potential V_{ext} can be thought of as an interaction between the electrons and the potential resulting from the positively charged Carbon atoms of the lattice. An electron on the lattice can move from site to site, which requires an amount of energy depending on how far apart the sites are. Thus, the Hamiltonian in equation (3.6) can be modeled as

$$\hat{H} = -\sum_{ij,\sigma}[\hat{a}^{\dagger}_{i,\sigma}t_{ij}\hat{b}_{j,\sigma} + \hat{a}^{\dagger}_{i,\sigma}t'_{ij}\hat{a}_{j,\sigma}] + \text{h.c.}$$

using second quantization, where i and j run over all possible lattice sites. An electron annihilated from the B sub-lattice and created on the A sub-lattice requires an energy t_{ij}, while an electron annihilated from the A sub-lattice site and created on another A sub-lattice site requires an energy t'_{ij}. To simplify, consider that tight binding will prevent electron hopping further than one honeycomb cell. This amounts to vanishing couplings t_{ij} and t'_{ij} beyond next nearest neighbors on the lattice. Furthermore, assume that zero energy amounts to an electron remaining localized to a lattice site. Thus, the Hamiltonian simplifies to

$$\hat{H} = -t\sum_{\langle i,j\rangle,\sigma}[\hat{a}^{\dagger}_{i,\sigma}\hat{b}_{j,\sigma} + \text{h.c.}] - t'\sum_{[i,j],\sigma}[\hat{a}^{\dagger}_{i,\sigma}\hat{a}_{j,\sigma} + \hat{b}^{\dagger}_{i,\sigma}\hat{b}_{i,\sigma} + \text{h.c.}], \tag{3.7}$$

where $\langle i,j\rangle$ and $[i,j]$ denote summation over nearest and next nearest neighbor sites, respectively. This is the tight binding Hamiltonian of Graphene, in second quantization.

3.1.3 Graphene and the Schroedinger equation

The next step is to find the single particle energy eigenstates $\epsilon_{\mathbf{k}}$ (i.e. dispersion relation) of the tight binding Hamiltonian of Graphene. Thus,

$$\hat{H}\Psi(\mathbf{r},t) = i\frac{\partial}{\partial t}\Psi(\mathbf{r},t) \tag{3.8}$$

needs to be solved, where $\Psi(\mathbf{r},t)$ is the single particle electron wave function in the lattice potential described by $\hat{H}$. Note that the Hamiltonian in equation (3.8) is not the second quantized Hamiltonian of equation (3.7). It is not necessary to consider time evolution in what follows and it is therefore going to be sufficient to solve the eigenvalue problem

$$\hat{H}\psi(\mathbf{r}) = \epsilon\psi(\mathbf{r}),$$

where ψ is the time independent part of $\Psi(\mathbf{r},t)$. In the tight binding approximation the wave functions $\{\psi^r(\mathbf{r})\}$, where r denotes the sub lattice, can be expanded as in equation (3.4). In the more basis independent bra-ket notation, this can be written as

$$|\mathbf{k}_r\rangle := |\psi^r_{\mathbf{k}}(\mathbf{r})\rangle = \frac{1}{\sqrt{N_r}}\sum_{i=1}^{N_r} e^{i\mathbf{r}_i\cdot\mathbf{k}}|\phi^r(\mathbf{r}-\mathbf{r}_i)\rangle,$$

where the spin indices have now been suppressed. It will be assumed that the Carbon atom wave functions satisfy

$$\langle\phi^q(\mathbf{r}-\mathbf{r}_i)|\phi^r(\mathbf{r}-\mathbf{r}_j)\rangle = \delta_{qr}\delta_{ij}$$

such that $\langle\mathbf{k}_q|\mathbf{k}'_r\rangle = \delta_{qr}\delta(\mathbf{k}-\mathbf{k}')$.

A single electron momentum state of the A-sub lattice is described by $|\mathbf{k}_a\rangle$ (at some position $\mathbf{r}$) and similarly a single electron momentum state on the B-sub lattice is described by $|\mathbf{k}_b\rangle$. Thus, a single electron momentum state $|\mathbf{K}\rangle$ on the honeycomb lattice must be described by the superposition

$$|\mathbf{K}\rangle := \lambda^a|\mathbf{k}_a\rangle + \lambda^b|\mathbf{k}_b\rangle,$$

where $\lambda^r \in \mathbb{R}$. Then, the Schroedinger equation to be solved is

$$\hat{H}|\mathbf{K}\rangle = \epsilon_{\mathbf{k}}|\mathbf{K}\rangle.$$

Proceeding as in reference [7] by multiplying from the left with $\langle\mathbf{k}_a|$ and $\langle\mathbf{k}_b|$ yields

$$\lambda^a\langle\mathbf{k}_a|\hat{H}|\mathbf{k}_a\rangle + \lambda^b\langle\mathbf{k}_a|\hat{H}|\mathbf{k}_b\rangle = \epsilon\lambda^a,$$

$$\lambda^a\langle\mathbf{k}_b|\hat{H}|\mathbf{k}_a\rangle + \lambda^b\langle\mathbf{k}_b|\hat{H}|\mathbf{k}_b\rangle = \epsilon\lambda^b.$$

Next, define $H_{ij} := \langle\mathbf{k}_i|\hat{H}|\mathbf{k}_j\rangle$ and rewrite the equations in the simplified form

$$H_{aa}\lambda^a + H_{ab}\lambda^b = \epsilon\lambda^a \Leftrightarrow [H_{aa} - \epsilon]\lambda^a + H_{ab}\lambda^b = 0,$$

$$H_{ba}\lambda^a + H_{bb}\lambda^b = \epsilon\lambda^b \Leftrightarrow H_{ba}\lambda^a + [H_{bb} - \epsilon]\lambda^b = 0.$$

Note that since $\hat{H}$ is Hermitian then $H^*_{ij} = H_{ji}$. In matrix form this becomes

$$U\boldsymbol{\lambda} \equiv \begin{pmatrix} H_{aa} - \epsilon & H_{ab} \\ H^*_{ab} & H_{bb} - \epsilon \end{pmatrix} \begin{pmatrix} \lambda^a \\ \lambda^b \end{pmatrix} = 0.$$

To calculate the eigenvalues ϵ one must compute $|U| = 0$. Thus,

$$[H_{aa} - \epsilon][H_{bb} - \epsilon] - H_{ab}H^*_{ab} = 0 \Leftrightarrow \epsilon^2 - [H_{aa} + H_{bb}]\epsilon + [H_{aa}H_{bb} - H_{ab}H^*_{ab}] = 0.$$

By looking at equation (3.7) and letting the electron states be Fock states and the Hamiltonian be in second quantization it is clear that

$$H_{aa} = \langle\mathbf{k}_a|\hat{H}|\mathbf{k}_a\rangle = \langle\mathbf{k}_b|\hat{H}|\mathbf{k}_b\rangle = H_{bb},$$

since an interchange of $\hat{a}$ and $\hat{b}$ leaves the Hamiltonian invariant. Thus,

$$\epsilon^2 - 2H_{aa}\epsilon + H^2_{aa} = H_{ab}H^*_{ab} \Leftrightarrow [\epsilon - H_{aa}]^2 = H_{ab}H^*_{ab},$$

which finally leads to

$$\epsilon = H_{aa} \pm \sqrt{H_{ab}H^*_{ab}}.$$

The matrix elements that were calculated above are written in the language of first quantization. However, these matrix elements can be calculated equivalently in the language of second quantization by letting $\langle\mathbf{k}_b| := \hat{a}_{\mathbf{k},\sigma}|0\rangle$ be a Fock state and letting $\hat{H}$ be in second quantization and in the momentum representation. In order to write equation (3.7) in the momentum representation the creation and annihilation operators must be Fourier transformed according to

$$\hat{a}_{i,\sigma} = \frac{1}{\sqrt{N}}\sum_{\mathbf{k}}\hat{a}_{\mathbf{k},\sigma}e^{i\mathbf{k}\cdot\mathbf{r}_i},\ \hat{a}^\dagger_{i,\sigma} = \frac{1}{\sqrt{N}}\sum_{\mathbf{k}}\hat{a}^\dagger_{\mathbf{k},\sigma}e^{-i\mathbf{k}\cdot\mathbf{r}_i}, \tag{3.9}$$

$$\hat{b}_{i,\sigma} = \frac{1}{\sqrt{N}}\sum_{\mathbf{k}}\hat{b}_{\mathbf{k},\sigma}e^{i\mathbf{k}\cdot\mathbf{r}_i},\ \hat{b}^\dagger_{i,\sigma} = \frac{1}{\sqrt{N}}\sum_{\mathbf{k}}\hat{b}^\dagger_{\mathbf{k},\sigma}e^{-i\mathbf{k}\cdot\mathbf{r}_i}. \tag{3.10}$$

It is also instructive to note that an equivalent procedure of finding ϵ is to multiply the Schroedinger equation from the left with $\langle\mathbf{K}|$, which yields

$$\begin{aligned}\langle\epsilon\rangle = \langle\mathbf{K}|\hat{H}|\mathbf{K}\rangle = (\lambda^a)^2\langle\mathbf{k}_a|\hat{H}|\mathbf{k}_a\rangle + (\lambda^b)^2\langle\mathbf{k}_b|\hat{H}|\mathbf{k}_b\rangle \\ + \lambda^a\lambda^b[\langle\mathbf{k}_a|\hat{H}|\mathbf{k}_b\rangle + \langle\mathbf{k}_b|\hat{H}|\mathbf{k}_a\rangle].\end{aligned}$$

Then, proceed to find the desired eigenvalues ϵ by forming the vector

$$\boldsymbol{\lambda} = (\lambda^a, \lambda^b)$$

and writing the equation in matrix form

$$\langle\epsilon\rangle = \boldsymbol{\lambda}^\dagger M\boldsymbol{\lambda}.$$

Diagonalizing M gives $\langle\epsilon\rangle = \tilde{\boldsymbol{\lambda}}^\dagger D\tilde{\boldsymbol{\lambda}}$ where D is diagonal. Then, the diagonal elements of D are the energy eigenvalues corresponding to the two possible energy bands.

3.1.4 Single particle dispersion relation

To calculate the dispersion, the Hamiltonian in equation (3.7) must be Fourier transformed. Using equations (3.9) and (3.10) on the first term of the Hamiltonian yields

$$\hat{H}_1 = \frac{1}{N}\sum_{\langle i,j\rangle,\sigma}\sum_{\mathbf{k},\mathbf{k}'}\hat{a}^\dagger_{\mathbf{k},\sigma}\hat{b}_{\mathbf{k}',\sigma}e^{-i\mathbf{r}_i\cdot\mathbf{k}+\mathbf{r}_j\cdot\mathbf{k}'} + \text{h.c.}.$$

Since the sum over i, j is over nearest neighbors only, the distance $\mathbf{r}_j - \mathbf{r}_i$ must be one of $\{-\boldsymbol{\delta}_m\}$ shown in figure 3.1, where $m \in \{1, 2, 3\}$. Using that $\mathbf{r}_j - \mathbf{r}_i = -\boldsymbol{\delta}_m$ gives

$$\hat{H}_1 = \frac{1}{N}\sum_{m=1}^{3}\sum_{i=1}^{N}\sum_{\sigma}\sum_{\mathbf{k},\mathbf{k}'}\hat{a}^\dagger_{\mathbf{k},\sigma}\hat{b}_{\mathbf{k}',\sigma}e^{-i\mathbf{r}_i\cdot(\mathbf{k}-\mathbf{k}')-i\boldsymbol{\delta}_m\cdot\mathbf{k}'} + \text{h.c.}.$$

In the discrete limit

$$\sum_{i=1}^{N}e^{-i\mathbf{r}_i\cdot(\mathbf{k}-\mathbf{k}')} = N\delta_{\mathbf{k}\mathbf{k}'},$$

which can be used to find that

$$\hat{H}_1 = \sum_{\mathbf{k},\sigma}\hat{a}^\dagger_{\mathbf{k},\sigma}\hat{b}_{\mathbf{k},\sigma}\sum_{m=1}^{3}e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}} + \text{h.c.}.$$

The Fourier transform of the second term of $\hat{H}$ is nearly identical except for the sum being over next nearest neighbors. This has as a consequence that $\mathbf{r}_j - \mathbf{r}_i$ instead must be equal to one of the vectors $\boldsymbol{\rho}_m \in \{\boldsymbol{\delta}_i^\pm\}$ where $m \in \{1, 2, 3, 4, 5, 6\}$ as described in subsection 3.1.1. Furthermore,

$$\sum_{i=1}^{N}e^{-i\mathbf{r}_i\cdot(\mathbf{k}-\mathbf{k}')} = \frac{N}{2}\delta_{\mathbf{k}\mathbf{k}'},$$

since the sum over next nearest neighbors only runs over half of the N lattice sites. Thus,

$$\hat{H}_2 = \frac{1}{2}\sum_{\mathbf{k},\sigma}\left[\hat{a}^\dagger_{\mathbf{k},\sigma}\hat{a}_{\mathbf{k},\sigma} + \hat{b}^\dagger_{\mathbf{k},\sigma}\hat{b}_{\mathbf{k},\sigma}\right]\sum_{m=1}^{6}e^{-i\boldsymbol{\rho}_m\cdot\mathbf{k}} + \text{h.c.}.$$

The complete Hamiltonian in momentum space becomes

$$\hat{H} = -t \sum_{\mathbf{k},\sigma} \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} \sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m \cdot \mathbf{k}} - \frac{t'}{2} \sum_{\mathbf{k},\sigma} \left[\hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} + \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} \right] \sum_{m=1}^{6} e^{-i\boldsymbol{\rho}_m \cdot \mathbf{k}} + \text{h.c.} \,. \tag{3.11}$$

In subsection 3.1.3 the matrix elements H_{ij} were defined. These need to be calculated in order to find the energy eigenvalues. It is easy to see from the Hamiltonian in equation (3.11) that the matrix elements

$$\langle \mathbf{k}_i | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_j \rangle \ , \ \langle \mathbf{k}_i | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_j \rangle \ , \ \langle \mathbf{k}_i | \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_j \rangle$$

and the hermitian conjugate of the first of these matrix elements

$$\langle \mathbf{k}_i | \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_j \rangle$$

must be computed for $(i, j) = \{(a, a), (a, b)\}$. Furthermore, it is immediately clear that the matrix elements $\langle \mathbf{k}_a | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_a \rangle$, $\langle \mathbf{k}_a | \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_a \rangle$ and $\langle \mathbf{k}_a | \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_a \rangle$ will vanish, since all of these contain either a B-sub lattice annihilation operator or a B-sub lattice creation operator that will annihilate the ket or bra vacuum state. Similarly, the matrix elements $\langle \mathbf{k}_a | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_b \rangle$, $\langle \mathbf{k}_a | \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_b \rangle$ and $\langle \mathbf{k}_a | \hat{b}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_b \rangle$ will vanish. Two non-vanishing matrix elements

$$\langle \mathbf{k}_a | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_a \rangle$$

and

$$\langle \mathbf{k}_a | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_b \rangle$$

are left which must be computed explicitly.

Expanding the first of the non-vanishing matrix element leads to

$$\begin{aligned} \langle \mathbf{k}_a | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} | \mathbf{k}_a \rangle &= \langle 0 | \hat{a}_{\mathbf{k}_a,\sigma_r} \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k},\sigma} \hat{a}^{\dagger}_{\mathbf{k}_a,\sigma_r} | 0 \rangle \\ &= \langle 0 | \left[\delta_{\mathbf{k}\mathbf{k}_a} \delta_{\sigma\sigma_r} - \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k}_a,\sigma_r} \right] \left[\delta_{\mathbf{k}\mathbf{k}_a} \delta_{\sigma\sigma_r} - \hat{a}^{\dagger}_{\mathbf{k}_a,\sigma_r} \hat{a}_{\mathbf{k},\sigma} \right] | 0 \rangle \\ &= \delta_{\mathbf{k}\mathbf{k}_a} \delta_{\sigma\sigma_r}, \end{aligned} \tag{3.12}$$

while expanding the second non-vanishing matrix element yields

$$\begin{aligned} \langle \mathbf{k}_a | \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} | \mathbf{k}_b \rangle &= \langle 0 | \hat{a}_{\mathbf{k}_a,\sigma_r} \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{b}_{\mathbf{k},\sigma} \hat{b}^{\dagger}_{\mathbf{k}_b,\sigma_{r'}} | 0 \rangle \\ &= \langle 0 | \left[\delta_{\mathbf{k}\mathbf{k}_a} \delta_{\sigma\sigma_r} - \hat{a}^{\dagger}_{\mathbf{k},\sigma} \hat{a}_{\mathbf{k}_a,\sigma_r} \right] \left[\delta_{\mathbf{k}\mathbf{k}_b} \delta_{\sigma\sigma_{r'}} - \hat{b}^{\dagger}_{\mathbf{k}_b,\sigma_{r'}} \hat{b}_{\mathbf{k},\sigma} \right] | 0 \rangle \\ &= \delta_{\mathbf{k}\mathbf{k}_a} \delta_{\mathbf{k}\mathbf{k}_b} \delta_{\sigma\sigma_r} \delta_{\sigma\sigma_{r'}} \,. \end{aligned} \tag{3.13}$$

Using the above results leads to

$$\langle \mathbf{k}_a | \hat{H} | \mathbf{k}_a \rangle = -\frac{t'}{2} \sum_{m=1}^{6} [e^{-i\boldsymbol{\rho}_m \cdot \mathbf{k}_a} + e^{i\boldsymbol{\rho}_m \cdot \mathbf{k}_a}],$$

where the second term originates from the hermitian conjugate term of $\hat{H}$. Similarly,

$$\langle \mathbf{k}_a | \hat{H} | \mathbf{k}_b \rangle = -t \delta_{\mathbf{k}_a \mathbf{k}_b} \delta_{\sigma_r \sigma_{r'}} \sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m \cdot \mathbf{k}_a}.$$

By assuming that the initial and final momentum state have the same momentum and same spin, then

$$\langle \mathbf{k}_a|\hat{H}|\mathbf{k}_b\rangle = -t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}_a}.$$

Finally,

$$\epsilon_{\mathbf{k}} = H_{aa} \pm \sqrt{H_{ab}H_{ab}^*} = -\frac{t'}{2}\sum_{m=1}^{6}[e^{-i\boldsymbol{\rho}_m\cdot\mathbf{k}} + e^{i\boldsymbol{\rho}_m\cdot\mathbf{k}}] \pm t\sqrt{\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}}\sum_{m'=1}^{3} e^{i\boldsymbol{\delta}_{m'}\cdot\mathbf{k}}}. \tag{3.14}$$

Note that, by inserting the values for $\{\boldsymbol{\rho}_m\}$

$$\begin{aligned}
\sum_{m=1}^{6} e^{-i\boldsymbol{\rho}_m\cdot\mathbf{k}} &= e^{i\frac{a}{2}(3k_x+\sqrt{3}k_y)} + e^{-i\frac{a}{2}(3k_x+\sqrt{3}k_y)} \\
&+ e^{i\frac{a}{2}(3k_x-\sqrt{3}k_y)} + e^{-i\frac{a}{2}(3k_x-\sqrt{3}k_y)} \\
&+ e^{ia\sqrt{3}k_y} + e^{-ia\sqrt{3}k_y} \\
&= (e^{i\frac{3a}{2}k_x} + e^{-i\frac{3a}{2}k_x})(e^{i\frac{a}{2}\sqrt{3}ky} + e^{-i\frac{a}{2}\sqrt{3}ky}) + e^{ia\sqrt{3}k_y} + e^{-ia\sqrt{3}k_y} \\
&= 4\cos\left(\frac{3a}{2}k_x\right)\cos\left(\frac{\sqrt{3}a}{2}k_y\right) + 2\cos\left(\sqrt{3}ak_y\right),
\end{aligned}$$

and

$$\begin{aligned}
\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}} &= e^{-i\frac{a}{2}(k_x+\sqrt{3}k_y)} + e^{-i\frac{a}{2}(k_x-\sqrt{3}k_y)} + e^{iak_x} \\
&= e^{-i\frac{a}{2}k_x}(e^{i\frac{\sqrt{3}a}{2}k_y} + e^{-i\frac{\sqrt{3}a}{2}k_y}) + e^{iak_x},
\end{aligned}$$

such that

$$\begin{aligned}
\sum_{m,m'=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}}e^{i\boldsymbol{\delta}'_m\cdot\mathbf{k}} &= \left[e^{-i\frac{a}{2}k_x}(e^{i\frac{\sqrt{3}a}{2}k_y} + e^{-i\frac{\sqrt{3}a}{2}k_y}) + e^{iak_x}\right]\left[e^{i\frac{a}{2}k_x}(e^{i\frac{\sqrt{3}a}{2}k_y} + e^{-i\frac{\sqrt{3}a}{2}k_y}) + e^{-iak_x}\right] \\
&= \left[e^{i\frac{\sqrt{3}a}{2}k_y} + e^{-i\frac{\sqrt{3}a}{2}k_y}\right]^2 + \left[e^{i\frac{3a}{2}k_x} + e^{-i\frac{3a}{2}k_x}\right]\left[e^{i\frac{\sqrt{3}a}{2}k_y} + e^{-i\frac{\sqrt{3}a}{2}k_y}\right] + 1 \\
&= 4\cos^2\left(\frac{\sqrt{3}a}{2}k_y\right) + 4\cos\left(\frac{3a}{2}k_x\right)\cos\left(\frac{\sqrt{3}a}{2}k_y\right) + 1 \\
&= 4\left[\frac{1}{2}\cos^2\left(\frac{\sqrt{3}a}{2}k_y\right) - \frac{1}{2}\sin^2\left(\frac{\sqrt{3}a}{2}k_y\right) + \frac{1}{2}\right] \\
&+ 4\cos\left(\frac{3a}{2}k_x\right)\cos\left(\frac{\sqrt{3}a}{2}k_y\right) + 1 \\
&= 2\cos\left(\sqrt{3}ak_y\right) + 4\cos\left(\frac{3a}{2}k_x\right)\cos\left(\frac{\sqrt{3}a}{2}k_y\right) + 3.
\end{aligned}$$

This gives the dispersion

$$\epsilon_{\mathbf{k}} = -t'f(\mathbf{k}) \pm t\sqrt{f(\mathbf{k})+3},$$

where

$$f(\mathbf{k}) := 2\cos\left(\sqrt{3}ak_y\right) + 4\cos\left(\frac{3a}{2}k_x\right)\cos\left(\frac{\sqrt{3}a}{2}k_y\right).$$

The dispersion $\epsilon_{\mathbf{k}}$ is plotted in figure 3.2 with the values $t = 1$ and $t' = 0.2t$. The units are such that $v_F = 3ta/2 := 1$ where v_F is the Fermi velocity. Thus, in these units the lattice constant is

$a = 2/3$. Notice that near the Dirac points in the low momentum regime the dispersion is nearly linear. This is shown in the magnified region of the plot. Furthermore, in the plot of figure 3.2 there is a broken symmetry between electrons and holes. The electron band is the positive part of the dispersion while the hole band is the negative part of the dispersion and they are asymmetric. The electron-hole symmetry is restored for $t' = 0$.

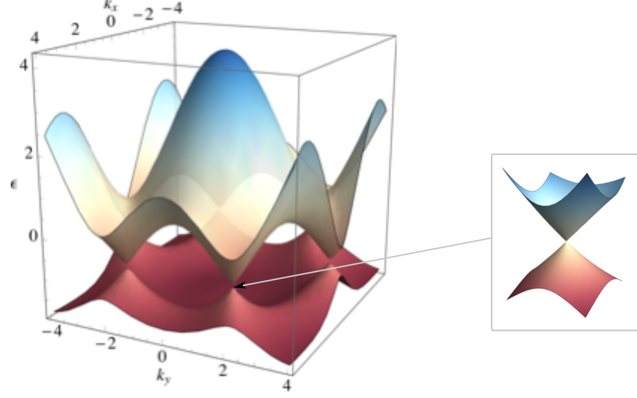


Figure 3.2: Dispersion of single layer Graphene with the values $t = 1$, $t' = 0.2t$ and $a = 2/3$.

3.1.5 Alternative method for calculating the dispersion

There is an alternative, more compact, method for calculating the dispersion by using second quantization. Starting from the Fourier transformed Hamiltonian in equation (3.11) and rewriting it in matrix form yields

$$\begin{aligned}\hat{H} &= \sum_{\mathbf{k},\sigma} \begin{pmatrix} \hat{a}^\dagger_{\mathbf{k},\sigma} & \hat{b}^\dagger_{\mathbf{k},\sigma} \end{pmatrix} \begin{pmatrix} -\frac{t'}{2}\sum_{m=1}^{6}[e^{-i\boldsymbol{\rho}_m\cdot\mathbf{k}} + e^{i\boldsymbol{\rho}_m\cdot\mathbf{k}}] & -t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}} \\ -t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{k}} & -\frac{t'}{2}\sum_{m=1}^{6}[e^{-i\boldsymbol{\rho}_m\cdot\mathbf{k}} + e^{i\boldsymbol{\rho}_m\cdot\mathbf{k}}] \end{pmatrix} \begin{pmatrix} \hat{a}_{\mathbf{k},\sigma} \\ \hat{b}_{\mathbf{k},\sigma} \end{pmatrix} \\ &\equiv \sum_{\mathbf{k},\sigma} \begin{pmatrix} \hat{a}^\dagger_{\mathbf{k},\sigma} & \hat{b}^\dagger_{\mathbf{k},\sigma} \end{pmatrix} \begin{pmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{pmatrix} \begin{pmatrix} \hat{a}_{\mathbf{k},\sigma} \\ \hat{b}_{\mathbf{k},\sigma} \end{pmatrix}. \end{aligned} \tag{3.15}$$

Continue to diagonalize the matrix of the Hamiltonian. The calculations are simplified by noting that $H_{11} = H_{22}$ and $H_{21} = H^*_{12}$. The eigenvalues λ_i are found by solving

$$\begin{vmatrix} H_{11} - \lambda & H_{12} \\ H^*_{12} & H_{11} - \lambda \end{vmatrix} = (H_{11} - \lambda)^2 - H_{12}H^*_{12} = 0,$$

which leads to

$$\lambda = H_{11} \pm \sqrt{H_{12}H^*_{12}}.$$

It turns out that the two eigenvalues λ are the two energy bands in the dispersion. However, to see this, it is necessary to find the eigenvector corresponding to $\lambda_1 = H_{11} + \sqrt{H_{12}H^*_{12}}$. Solving

$$\begin{pmatrix} H_{11} - \lambda_1 & H_{12} \\ H^*_{12} & H_{11} - \lambda_1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} -\sqrt{H_{12}H^*_{12}} & H_{12} \\ H^*_{12} & -\sqrt{H_{12}H^*_{12}} \end{pmatrix} \mathbf{x} = 0,$$

leads to the two equations

$$-\sqrt{H_{12}H^*_{12}}x_1 + H_{12}x_2 = 0, \tag{3.16}$$

$$H^*_{12}x_1 - \sqrt{H_{12}H^*_{12}}x_2 = 0. \tag{3.17}$$

Inserting equation (3.17) into equation (3.16) shows that one can choose $x_1 = a$ for all $a \in \mathbb{R}$. Thus, using equation (3.17) yields

$$x_2 = \frac{aH_{12}^*}{\sqrt{H_{12}H_{12}^*}},$$

and the eigenvector $\mathbf{x}_1$ belonging to the eigenvalue λ_1 is

$$\mathbf{x}_1 = a \begin{pmatrix} 1 \\ \frac{H_{12}^*}{\sqrt{H_{12}H_{12}^*}} \end{pmatrix}.$$

Similarly, it is found that the eigenvector $\mathbf{x}_2$ belonging to the eigenvalue λ_2 is

$$\mathbf{x}_2 = b \begin{pmatrix} 1 \\ \frac{-H_{12}^*}{\sqrt{H_{12}H_{12}^*}} \end{pmatrix}.$$

Note that $\mathbf{x}_1 \cdot \mathbf{x}_2 = \mathbf{x}_1^\dagger \mathbf{x}_2 = 0$ and that

$$\mathbf{x}_1 \cdot \mathbf{x}_1 = \mathbf{x}_1^\dagger \mathbf{x}_1 = 2a,$$

$$\mathbf{x}_2 \cdot \mathbf{x}_2 = \mathbf{x}_2^\dagger \mathbf{x}_2 = 2b.$$

Thus, there exists a freedom to choose an orthonormal set of eigenvectors by setting $a = b = 1/\sqrt{2}$. This allows the unitary matrix

$$U = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ \frac{H_{12}^*}{\sqrt{H_{12}H_{12}^*}} & \frac{-H_{12}^*}{\sqrt{H_{12}H_{12}^*}} \end{pmatrix}$$

and the diagonal matrix

$$D = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} = \begin{pmatrix} H_{11} + \sqrt{H_{12}H_{12}^*} & 0 \\ 0 & H_{11} - \sqrt{H_{12}H_{12}^*} \end{pmatrix}$$

to be constructed. It is easy to check that $U^\dagger U = 1$ and that

$$UDU^\dagger = \begin{pmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{pmatrix}.$$

The matrix U diagonalizes the Hamiltonian and

$$\hat{H} = \sum_{\mathbf{k},\sigma} \begin{pmatrix} \hat{a}_{\mathbf{k},\sigma}^\dagger & \hat{b}_{\mathbf{k},\sigma}^\dagger \end{pmatrix} UU^\dagger \begin{pmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{pmatrix} UU^\dagger \begin{pmatrix} \hat{a}_{\mathbf{k},\sigma} \\ \hat{b}_{\mathbf{k},\sigma} \end{pmatrix} U,$$

leading to

$$\hat{H} = \sum_{\mathbf{k},\sigma} \begin{pmatrix} \tilde{a}_{\mathbf{k},\sigma}^\dagger & \tilde{b}_{\mathbf{k},\sigma}^\dagger \end{pmatrix} \begin{pmatrix} H_{11} + \sqrt{H_{12}H_{12}^*} & 0 \\ 0 & H_{11} - \sqrt{H_{12}H_{12}^*} \end{pmatrix} \begin{pmatrix} \tilde{a}_{\mathbf{k},\sigma} \\ \tilde{b}_{\mathbf{k},\sigma} \end{pmatrix},$$

The anti-commutator relations for $\hat{a}$ and $\hat{b}$ stay invariant under unitary transformations U such that $\tilde{a}$ and $\tilde{b}$ conform to the same anti-commutation relations. Thus, $\tilde{a}$ and $\tilde{b}$ are also annihilation operators. However, while for example $\hat{a}$ annihilates an electron of a certain momentum from the A-sub lattice, $\tilde{a}$ will annihilate an electron with the same momentum from one of the two energy bands of the system. This can be seen more clearly by writing the Hamiltonian in the form

$$\hat{H} = \sum_{\mathbf{k},\sigma} \left[H_{11} + \sqrt{H_{12}H_{12}^*} \right] \tilde{a}_{\mathbf{k},\sigma}^\dagger \tilde{a}_{\mathbf{k},\sigma} + \sum_{\mathbf{k},\sigma} \left[H_{11} - \sqrt{H_{12}H_{12}^*} \right] \tilde{b}_{\mathbf{k},\sigma}^\dagger \tilde{b}_{\mathbf{k},\sigma}.$$

It is already known from second quantization that the dispersion of the system can be defined as

$$\hat{H} = \sum_{\mathbf{k},\sigma} \hbar\omega^a_{\mathbf{k}} \hat{a}^\dagger_{\mathbf{k},\sigma}\hat{a}_{\mathbf{k},\sigma} + \sum_{\mathbf{k},\sigma} \hbar\omega^b_{\mathbf{k}} \hat{b}^\dagger_{\mathbf{k},\sigma}\hat{b}_{\mathbf{k},\sigma},$$

where in that case $\hat{a}$ and $\hat{b}$ annihilate electrons from the two energy bands. Thus, the dispersion relation is found to be

$$\epsilon_{\mathbf{k}} = H_{11} \pm \sqrt{H_{12}H_{12}^*} = -\frac{t'}{2}\sum_{m=1}^{6}[e^{-i\boldsymbol{\rho}_m\cdot\mathbf{k}} + e^{i\boldsymbol{\rho}_m\cdot\mathbf{k}}] \pm t\sqrt{\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}} \sum_{m'=1}^{3} e^{i\boldsymbol{\delta}_{m'}\cdot\mathbf{k}}},$$

which is exactly what was found in equation (3.14).

3.2 Ferromagnetism in monolayer graphene

3.2.1 Low energy dispersion relation

In the following, only nearest neighbor hopping terms of the graphene Hamiltonian is considered and $t' = 0$. Thus, equation (3.15) shows that the matrix representation of the momentum space Hamiltonian is

$$\mathcal{H}_K = \begin{pmatrix} 0 & -t\sum_{m=1}^3 e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}} \\ -t\sum_{m=1}^3 e^{i\boldsymbol{\delta}_m\cdot\mathbf{k}} & 0 \end{pmatrix},$$

where the values of the vectors $\boldsymbol{\delta}_1$, $\boldsymbol{\delta}_2$ and $\boldsymbol{\delta}_3$ were calculated in section 3.1.1. In order to find the low energy approximation of $\mathcal{H}_K$ it will be linearized around the Dirac point $\mathbf{K} = \left(0, \frac{4\pi}{3\sqrt{3}a}\right)$. Evaluating the entries of $\mathcal{H}_K$ at $\mathbf{K}$ then yields

$$-t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{K}} = -t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{K}} = 0, \tag{3.18}$$

while evaluating the partial derivatives of the entries at $\mathbf{K}$ results in

$$-\frac{\partial}{\partial k_x} t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = \frac{\partial}{\partial k_x} t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = -i\frac{3at}{2}, \tag{3.19}$$

$$-\frac{\partial}{\partial k_y} t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = -\frac{\partial}{\partial k_y} t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = \frac{3at}{2}. \tag{3.20}$$

Thus, to first order in k [12]

$$\mathcal{H}_K = \frac{3at}{2}\begin{pmatrix} 0 & k_y - ik_x \\ k_y + ik_x & 0 \end{pmatrix}.$$

The eigenvalues of $\mathcal{H}_K$ are $\lambda = \pm|\mathbf{k}|$, which leads to the diagonalized Hamiltonian

$$\tilde{\mathcal{H}}_K = \frac{3at}{2}\begin{pmatrix} |\mathbf{k}| & 0 \\ 0 & -|\mathbf{k}| \end{pmatrix}.$$

Thus, the low energy dispersion relation, expanded around $\mathbf{K}$, becomes [1]

$$\epsilon^{\pm}_{\mathbf{k}} = \pm v_F|\mathbf{k}|, \tag{3.21}$$

where $v_F = \frac{3at}{2}$ is the fermi velocity.

3.2.2 The monolayer field operator

By solving

$$\begin{pmatrix} \pm v_F|\mathbf{k}| & v_F[k_y - ik_x] \\ v_F[k_y + ik_x] & \pm v_F|\mathbf{k}| \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = 0$$

with respect to ψ_1 and ψ_2 one finds the eigenstates

$$\psi_\alpha = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix}_\alpha = C \begin{pmatrix} 1 \\ \pm\frac{|\mathbf{k}|}{k_x - ik_y} \end{pmatrix} = C \begin{pmatrix} 1 \\ \pm[k_x + ik_y] \end{pmatrix} = C \begin{pmatrix} 1 \\ \alpha e^{-i\phi_\mathbf{k}} \end{pmatrix},$$

where $\phi_\mathbf{k} = \tan^{-1}(k_y/k_x)$ is the angle of the momentum vector $\mathbf{k}$, C is a constant and $\alpha \in \{\pm\}$ denotes the two possible band states (i.e. hole $(-)$ and electron $(+)$ bands). The wave function is normalized by setting $C = 1/\sqrt{2}$. Furthermore, spin is added by multiplying with the spin wave function χ_σ. Then, a Fourier component of the wave function is described by

$$\Psi_{\mathbf{k},\sigma,\alpha}(\mathbf{r}) = \frac{e^{i\mathbf{k}\cdot\mathbf{r}}}{\sqrt{2}} \begin{pmatrix} 1 \\ \alpha e^{-i\phi_\mathbf{k}} \end{pmatrix} \chi_\sigma, \tag{3.22}$$

where the corresponding second quantized field operator is

$$\hat{\Psi}(\mathbf{r}) = \frac{1}{\sqrt{A_c}} \sum_{\mathbf{k},\sigma,\alpha} \Psi_{\mathbf{k},\sigma,\alpha}(\mathbf{r}) \hat{a}_{\mathbf{k},\sigma,\alpha}. \tag{3.23}$$

As expected, with the chosen normalizations, the field operator satisfies the anti-commutator relation:

$$\begin{aligned}
\left\{\hat{\Psi}_i(\mathbf{r}), \hat{\Psi}_j^\dagger(\mathbf{r}')\right\} &= \frac{1}{A_c} \sum_{\mathbf{k},\sigma,\alpha} \sum_{\mathbf{k}',\sigma',\beta} \Psi_{\mathbf{k},\sigma,\alpha,i}(\mathbf{r}) \Psi^\dagger_{\mathbf{k}',\sigma',\beta,j}(\mathbf{r}') \left\{\hat{a}_{\mathbf{k},\sigma,\alpha}, \hat{a}^\dagger_{\mathbf{k}',\sigma',\beta}\right\} \\
&= \delta_{ij} \frac{1}{A_c} \sum_{\mathbf{k},\sigma,\alpha} \Psi_{\mathbf{k},\sigma,\alpha,i}(\mathbf{r}) \Psi^\dagger_{\mathbf{k},\sigma,\alpha,j}(\mathbf{r}') \\
&= \delta_{ij} \frac{1}{2A_c} \sum_\sigma \chi^*_\sigma \chi_\sigma \sum_{\mathbf{k},\alpha} e^{i\mathbf{k}\cdot(\mathbf{r}-\mathbf{r}')} \left[\begin{pmatrix} 1 & \alpha e^{i\phi_\mathbf{k}} \end{pmatrix} \begin{pmatrix} 1 \\ \alpha e^{-i\phi_\mathbf{k}} \end{pmatrix} \right] \\
&= \delta_{ij} \frac{1}{2A_c} \sum_\sigma \chi^*_\sigma \chi_\sigma \sum_{\mathbf{k},\alpha} e^{i\mathbf{k}\cdot(\mathbf{r}-\mathbf{r}')} \left[1 + \alpha^2\right] \\
&= \delta_{ij} \sum_\sigma \chi^*_\sigma \chi_\sigma \frac{1}{A_c} \sum_\mathbf{k} e^{i\mathbf{k}\cdot(\mathbf{r}-\mathbf{r}')} = \delta_{ij} \delta^{(2)}(\mathbf{r} - \mathbf{r}').
\end{aligned}$$

3.2.3 Average kinetic energy

The density of states is needed in order to arrive at an expression for the average kinetic energy of the electrons. The density of states is calculated by first counting the number of states $N^\pm$ below a Fermi energy ϵ for each band. This yields

$$N^\pm = \sum_\mathbf{k} \Theta(\epsilon - \epsilon^\pm_\mathbf{k}),$$

where the sum is over all $\mathbf{k}$ and Θ is the Heaviside step function. By assuming periodic boundary conditions such that $k_i = 2\pi n/L$ it is easy to find the corresponding integral in the continuum limit. In two dimensions the density of momentum states is $A_c/(2\pi)^2$ which leads to

$$N^\pm = \frac{A_c}{(2\pi)^2} \int d\mathbf{k} \Theta(\epsilon - \epsilon^\pm_\mathbf{k}) = \frac{A_c}{2\pi} \int_0^\infty dk k \Theta(\epsilon - \epsilon^\pm_\mathbf{k}) = \frac{A_c}{2\pi} \int_0^{k^\pm(\epsilon)} dk k,$$

where in the last step polar coordinates were introduced. Equation (3.21) implies that $k^{\pm}(\epsilon) = \pm\epsilon/v_F$ and therefore

$$N^{\pm} = \frac{A_c}{4\pi}\left(\frac{\epsilon}{v_F}\right)^2.$$

Combining this with the existence of a two fold degeneracy due to the two possible spin states of the system leads to the density of states

$$\mathcal{D}^{\pm}(\epsilon) = 2\frac{\partial N}{\partial \epsilon} = \frac{A_c}{\pi}\frac{\epsilon}{v_F^2}$$

near the Dirac point $\mathbf{K}$, where $A_c = 3a^2\sqrt{3}/2$ is the area of the honeycomb unit cell in position space [1]. The density of states can now be utilized to calculate the expectation value of the

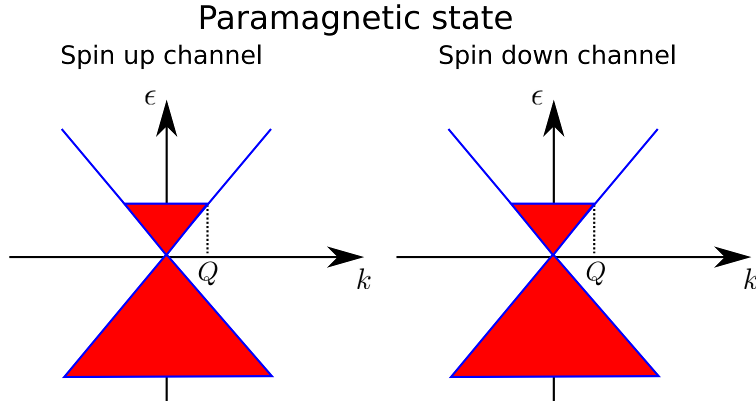


Figure 3.3: Unpolarized state.

kinetic energy by integrating $\epsilon\mathcal{D}^{\pm}(\epsilon)$ over all energies up to the Fermi energy which is limited by the momentum cutoff. The low energy dispersion is only valid close to the Dirac points. At the very least it is required that all momenta are within the first Brillouin zone. Thus, a cutoff k_c is introduced such that $0 \leq |\mathbf{k}| \leq k_c$. By considering k_c as a radius in momentum space it is found that $\pi k_c^2 = A_B$, where A_B is the area of the first Brillouin zone in momentum space. Note that $A_B = (2\pi)^2/A_c$. This has as a consequence that the number of states in the Brillouin zone is fixed. Labeling the Fermi momentum as Q and considering an unpolarized state,as shown in figure 3.3, leads to

$$\langle E_K\rangle = \int_0^{\epsilon(Q)} d\epsilon\epsilon\mathcal{D}^{+}(\epsilon) - \int_0^{\epsilon(k_c)} d\epsilon\epsilon\mathcal{D}^{-}(\epsilon) = \frac{A_c}{\pi v_F^2}\int_{\epsilon(k_c)}^{\epsilon(Q)} d\epsilon\epsilon^2 = \frac{A_c}{3\pi v_F^2}[\epsilon^3(Q) - \epsilon^3(k_c)].$$

Note that both the spin channels of figure 3.3 have been taken into account through the spin degeneracy factor of 2 that was introduced into $\mathcal{D}(\epsilon)$. Using the linearized dispersion relation of equation (3.21) leads to

$$\langle E_K\rangle = -\frac{A_c v_F}{3\pi}[k_c^3 - Q^3].$$

In the case that the two spin channels differ (i.e. in a polarized state), each spin channel can be calculated separately and then summed. Thus,

$$\langle E_K\rangle = -\frac{1}{2}\sum_{\sigma\in\{\uparrow,\downarrow\}}\frac{A_c v_F}{3\pi}[k_c^3 - Q_\sigma^3], \tag{3.24}$$

where Q_σ denotes the Fermi momentum in spin channel σ, and the factor of 1/2 originates from removing the spin degeneracy of each term in the summation.

3.2.4 Exchange energy due to Coulomb interactions

The exchange energy will be calculated based on the derivations done in reference [12]. In second quantization the interaction Hamiltonian takes the form

$$\mathcal{H}_I = \frac{1}{2}\int d\mathbf{r}_1 d\mathbf{r}_2 \hat{\Psi}^\dagger(\mathbf{r}_1)\hat{\Psi}^\dagger(\mathbf{r}_2)V(\mathbf{r}_2-\mathbf{r}_1)\hat{\Psi}(\mathbf{r}_2)\hat{\Psi}^(\mathbf{r}_1),$$

where

$$V(\mathbf{r}_2-\mathbf{r}_1) = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}_2-\mathbf{r}_1|}$$

is the Coulomb potential. Inserting the expressions for the field operators of equation (3.23) into the expression for $\mathcal{H}_I$ leads to

$$\begin{aligned}\mathcal{H}_I =& \frac{e^2}{2\epsilon_0 A_c^2}\sum_{\mathbf{k}_1,\ldots,\mathbf{k}_4}\sum_{\alpha_1,\ldots,\alpha_4}\sum_{\sigma_1,\ldots,\sigma_4}\hat{a}^\dagger_{\mathbf{k}_1,\sigma_1,\alpha_1}\hat{a}^\dagger_{\mathbf{k}_2,\sigma_2,\alpha_2}\hat{a}_{\mathbf{k}_3,\sigma_3,\alpha_3}\hat{a}_{\mathbf{k}_4,\sigma_4,\alpha_4}\\ &\times\int d\mathbf{r}_1 d\mathbf{r}_2 \Psi^\dagger_{\mathbf{k}_1,\sigma_1,\alpha_1}\Psi^\dagger_{\mathbf{k}_2,\sigma_2,\alpha_2}\Psi_{\mathbf{k}_3,\sigma_3,\alpha_3}\Psi_{\mathbf{k}_4,\sigma_4,\alpha_4}\frac{1}{|\mathbf{r}_2-\mathbf{r}_1|}.\end{aligned}$$

Furthermore, after inserting the wave function of equation (3.22) into this expression the Hamiltonian becomes

$$\begin{aligned}\mathcal{H}_I =& \frac{e^2}{8\epsilon_0 A_c^2}\sum_{\mathbf{k}_1,\ldots,\mathbf{k}_4}\sum_{\alpha_1,\ldots,\alpha_4}\sum_{\sigma_1,\ldots,\sigma_4}\hat{a}^\dagger_{\mathbf{k}_1,\sigma_1,\alpha_1}\hat{a}^\dagger_{\mathbf{k}_2,\sigma_2,\alpha_2}\hat{a}_{\mathbf{k}_3,\sigma_3,\alpha_3}\hat{a}_{\mathbf{k}_4,\sigma_4,\alpha_4}\\ &\times\chi^*_{\sigma_1}\chi^*_{\sigma_2}\chi_{\sigma_3}\chi_{\sigma_4}\left\{1+\alpha_2\alpha_3 e^{i[\phi_{\mathbf{k}_2}-\phi_{\mathbf{k}_3}]}\right\}\left\{1+\alpha_1\alpha_4 e^{i[\phi_{\mathbf{k}_1}-\phi_{\mathbf{k}_4}]}\right\}\\ &\times\int d\mathbf{r}_1 d\mathbf{r}_2\frac{1}{|\mathbf{r}_2-\mathbf{r}_1|}e^{i\mathbf{r}_1\cdot(\mathbf{k}_4-\mathbf{k}_1)}e^{i\mathbf{r}_2\cdot(\mathbf{k}_3-\mathbf{k}_4)}.\end{aligned} \tag{3.25}$$

Making the substitution $\mathbf{r}=\mathbf{r}_2-\mathbf{r}_1$, noting that the Jacobian is unity and thus $d\mathbf{r}=d\mathbf{r}_2$, leads to

$$\int d\mathbf{r}_1 d\mathbf{r}_2\frac{1}{|\mathbf{r}_2-\mathbf{r}_1|}e^{i\mathbf{r}_1\cdot(\mathbf{k}_4-\mathbf{k}_1)}e^{i\mathbf{r}_2\cdot(\mathbf{k}_3-\mathbf{k}_4)} = \int d\mathbf{r}_1 e^{i\mathbf{r}_1\cdot(\mathbf{k}_4-\mathbf{k}_1+\mathbf{k}_3-\mathbf{k}_2)}\int d\mathbf{r}\frac{1}{|\mathbf{r}|}e^{i\mathbf{r}\cdot(\mathbf{k}_3-\mathbf{k}_2)},$$

where the integral representation

$$\int d\mathbf{r}_1 e^{i\mathbf{r}_1\cdot(\mathbf{k}_4-\mathbf{k}_1+\mathbf{k}_3-\mathbf{k}_2)} = A_c\delta_{\mathbf{k}_4-\mathbf{k}_1+\mathbf{k}_3-\mathbf{k}_2,0}$$

of the delta function is recognized. By inserting this into equation (3.25) at the same time as setting $\mathbf{p}=\mathbf{k}_3-\mathbf{k}_2$ and summing over $\mathbf{p}$ instead of $\mathbf{k}_3$ it is found that

$$\begin{aligned}\mathcal{H}_I =& \frac{e^2}{8\epsilon_0 A_c}\sum_{\mathbf{k}_1,\mathbf{k}_2,\mathbf{p}}\sum_{\alpha_1,\ldots,\alpha_4}\sum_{\sigma_1,\ldots,\sigma_4}\hat{a}^\dagger_{\mathbf{k}_1,\sigma_1,\alpha_1}\hat{a}^\dagger_{\mathbf{k}_2,\sigma_2,\alpha_2}\hat{a}_{\mathbf{k}_2+\mathbf{p},\sigma_3,\alpha_3}\hat{a}_{\mathbf{k}_1-\mathbf{p},\sigma_4,\alpha_4}\\ &\times\chi^*_{\sigma_1}\chi^*_{\sigma_2}\chi_{\sigma_3}\chi_{\sigma_4}\left\{1+\alpha_2\alpha_3 e^{i[\phi_{\mathbf{k}_2}-\phi_{\mathbf{k}_2+\mathbf{p}}]}\right\}\left\{1+\alpha_1\alpha_4 e^{i[\phi_{\mathbf{k}_1}-\phi_{\mathbf{k}_1-\mathbf{p}}]}\right\}\int d\mathbf{r}\frac{1}{|\mathbf{r}|}e^{i\mathbf{r}\cdot\mathbf{p}}.\end{aligned}$$

It is possible to write

$$\int d\mathbf{r}\frac{1}{|\mathbf{r}|}e^{i\mathbf{r}\cdot\mathbf{p}} = \int_0^{2\pi}d\theta\int_0^\infty dr e^{irp\cos\theta} = \frac{1}{p}\int_0^\infty dr\int_0^{2\pi}d\theta e^{ir\cos\theta},$$

where the substitution $r'=rp$ was performed followed by renaming the integration variable $r'\to r$. Recognizing the Bessel function $J_0(r)$ the expression reads

$$\int d\mathbf{r}\frac{1}{|\mathbf{r}|}e^{i\mathbf{r}\cdot\mathbf{p}} = \frac{1}{p}\int_0^\infty dr 2\pi J_0(r) = \frac{2\pi}{p} = \frac{2\pi}{|\mathbf{p}|}.$$

Mutual orthonormality of the spin wave functions χ^* and χ leads to

$$\mathcal{H}_I = \frac{2\pi e^2}{8\epsilon_0 A_c} \sum_{\mathbf{k_1},\mathbf{k_2},\mathbf{p}} \sum_{\alpha_1,\ldots,\alpha_4} \sum_{\sigma_1,\sigma_2} \hat{a}^\dagger_{\mathbf{k_1},\sigma_1,\alpha_1} \hat{a}^\dagger_{\mathbf{k_2},\sigma_2,\alpha_2} \hat{a}_{\mathbf{k_2+p},\sigma_2,\alpha_3} \hat{a}_{\mathbf{k_1-p},\sigma_1,\alpha_4}$$
$$\times \left\{1 + \alpha_2\alpha_3 e^{i[\phi_{\mathbf{k_2}} - \phi_{\mathbf{k_2+p}}]}\right\} \left\{1 + \alpha_1\alpha_4 e^{i[\phi_{\mathbf{k_1}} - \phi_{\mathbf{k_1-p}}]}\right\} \frac{1}{|\mathbf{p}|}.$$

To calculate the total energy of a configuration $|\mathbf{N}\rangle$ the expectation value of the Hamiltonian must be calculated with respect to this state. This yields

$$E_f = \langle \mathbf{N}| : H_I : |\mathbf{N}\rangle,$$

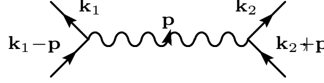
where

$$|\mathbf{N}\rangle = \prod_{\mathbf{k},\sigma,\alpha} \left[\hat{a}^\dagger_{\mathbf{k},\sigma,\alpha}\right]^{N_{\mathbf{k},\sigma,\alpha}},$$

and : : denotes normal ordering. Thus, one needs to evaluate the matrix element

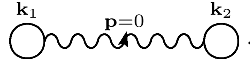
$$\mathcal{V} := \langle \mathbf{N}|\hat{a}^\dagger_{\mathbf{k_1},\sigma_1,\alpha_1} \hat{a}^\dagger_{\mathbf{k_2},\sigma_2,\alpha_2} \hat{a}_{\mathbf{k_2+p},\sigma_2,\alpha_3} \hat{a}_{\mathbf{k_1-p},\sigma_1,\alpha_4}|\mathbf{N}\rangle V(\mathbf{p}).$$

This can be represented by the diagram

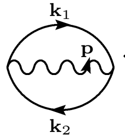


where time runs from the bottom of the diagram to the top of the diagram. In the above situation, the interaction $1/p$ represents the bare potential. Therefore, the interaction energy is an approximation that does not take into account higher order corrections above first order in the interaction. Thus, screening effects are neglected in the above expression.

Note that $\mathcal{V}$ will vanish if $\{\mathbf{k}_1, \sigma_1, \alpha_1\}$ differs from $\{\mathbf{k_1} - \mathbf{p}, \sigma_1, \alpha_4\}$ and $\{\mathbf{k}_2 + \mathbf{p}, \sigma_2, \alpha_3\}$. A similar argument holds for $\{\mathbf{k}_2, \sigma_2, \alpha_2\}$. Thus, there are only two possible terms that survive after expanding the Fock states. One of these occurs when $\{\mathbf{k}_1, \sigma_1, \alpha_1\} = \{\mathbf{k_1} - \mathbf{p}, \sigma_1, \alpha_4\}$ and $\{\mathbf{k}_2, \sigma_2, \alpha_2\} = \{\mathbf{k}_2 + \mathbf{p}, \sigma_2, \alpha_3\}$, which can be represented by



This is recognized as the Hartree contribution and will vanish since $V(p = 0) = 0$ due to a positive Jellium background. The other contribution arises from terms where $\{\mathbf{k}_1, \sigma_1, \alpha_1\} = \{\mathbf{k}_2{+}\mathbf{p}, \sigma_2, \alpha_3\}$ and $\{\mathbf{k}_2, \sigma_2, \alpha_2\} = \{\mathbf{k_1} - \mathbf{p}, \sigma_1, \alpha_4\}$, which is represented by



This is recognized as the Fock contribution, also referred to as the exchange contribution. Inserting the delta functions and the Fermi occupation functions (i.e. densities) resulting from this term into the energy E_f leads to the exchange energy

$$E_{ex} = -\frac{2\pi e^2}{8\epsilon_0 A_c} \sum_{\mathbf{k},\mathbf{k'}} \sum_{\sigma} \sum_{\alpha,\alpha'} \frac{1}{|\mathbf{k} - \mathbf{k'}|} \left\{1 + \alpha\alpha' e^{i[\phi_{\mathbf{k'}} - \phi_{\mathbf{k}}]}\right\} \left\{1 + \alpha\alpha' e^{i[\phi_{\mathbf{k}} - \phi_{\mathbf{k'}}]}\right\} n_F^{\sigma,\alpha}(\mathbf{k}) n_F^{\sigma,\alpha'}(\mathbf{k'}).$$

Denoting the angle between $\mathbf{k}$ and $\mathbf{k}'$ as $\theta \equiv \phi_{\mathbf{k}} - \phi_{\mathbf{k}'}$ and multiplying the two factors under the sum, the exchange energy becomes

$$E_{ex} = -\frac{2\pi e^2}{8\epsilon_0 A_c} \sum_{\mathbf{k},\mathbf{k}'} \sum_{\sigma} \sum_{\alpha,\alpha'} \frac{1}{|\mathbf{k} - \mathbf{k}'|} \left\{1 + \alpha\alpha' e^{i\theta} + \alpha\alpha' e^{-i\theta} + (\alpha\alpha')^2\right\} n_F^{\sigma,\alpha}(\mathbf{k}) n_F^{\sigma,\alpha'}(\mathbf{k}').$$

Since the band index α is either $+1$ or -1 then $(\alpha\alpha')^2 = 1$. Furthermore, it is possible to write the momentum sums as integrals by letting

$$\sum_{\mathbf{k}} \to \frac{A_c}{(2\pi)^2} \int d\mathbf{k}.$$

This leads to

$$E_{ex} = -\frac{A_c}{(2\pi)^3} \frac{e^2}{8\epsilon_0} \sum_{\sigma} \sum_{\alpha,\alpha'} \int d\mathbf{k} \int d\mathbf{p} \frac{1}{|\mathbf{k} - \mathbf{p}|} \left\{2 + 2\alpha\alpha' \cos\theta\right\} n_F^{\sigma,\alpha}(\mathbf{k}) n_F^{\sigma,\alpha'}(\mathbf{p}).$$

Introducing polar coordinates one finally arrives at the exchange energy

$$E_{ex} = -\frac{A_c}{(2\pi)^2} \frac{e^2}{4\epsilon_0} \sum_{\sigma} \sum_{\alpha,\alpha'} \int_0^{2\pi} d\theta \int kdk \int pdp \frac{1 + \alpha\alpha' \cos\theta}{|\mathbf{k} - \mathbf{p}|} n_F^{\sigma,\alpha}(\mathbf{k}) n_F^{\sigma,\alpha'}(\mathbf{p}). \quad (3.26)$$

3.2.5 Phase transitions

It is desirable to explore the possibility of monolayer graphene being in a ferromagnetic state. To active this, the analysis performed in reference [12] will be followed. Suppose a system is prepared in a paramagnetic state and another system in a ferromagnetic state. Then the total energy $E = \langle E_K \rangle + E_{ex}$ can be calculated for each of the systems by using equations (3.24) and (3.26). Finally one can subtract the total energy of the paramagnetic system from the ferromagnetic system. If this energy difference is positive it is known that a paramagnetic phase is preferred, but if it is negative then a ferromagnetic phase is preferred.

Imagine a paramagnetic phase with the energy bands half filled. The ferromagnetic phase can be constructed to have an increased filling in the spin-up channel and a corresponding decrease in the spin-down channel. This is referred to as an electron / hole pocket which is illustrated in figure 3.4.

From equation (3.24) it can be seen that the ferromagnetic phase results in a total kinetic energy

$$E_K^{FM} = -\frac{1}{2}\left\{\frac{A_c v_F}{3\pi}[k_c^3 - 0] + \frac{A_c v_F}{3\pi}[0 - k_\uparrow^3]\right\} - \frac{1}{2}\left\{\frac{A_c v_F}{3\pi}[k_c^3 - k_\downarrow^3]\right\} = -\frac{A_c v_F}{3\pi}[k_c^3 - k_\uparrow^3],$$

where $k_\uparrow = k_\downarrow$. In the paramagnetic phase $k_{F,\sigma} = 0$, which yields

$$E_K^{PM} = -\frac{A_c v_F}{3\pi} k_c^3.$$

Putting the last two results together leads to a total kinetic energy difference of

$$\Delta E_K = E_K^{FM} - E_K^{PM} = \frac{A_c v_F}{3\pi} k_\uparrow^3.$$

In equation (3.26) it is necessary to sum over all possible combinations of bands $\{\alpha, \alpha'\} \in \{\pm 1, \pm 1\}$ and integrate over the occupied momentum states belonging to the respective bands. Guided by

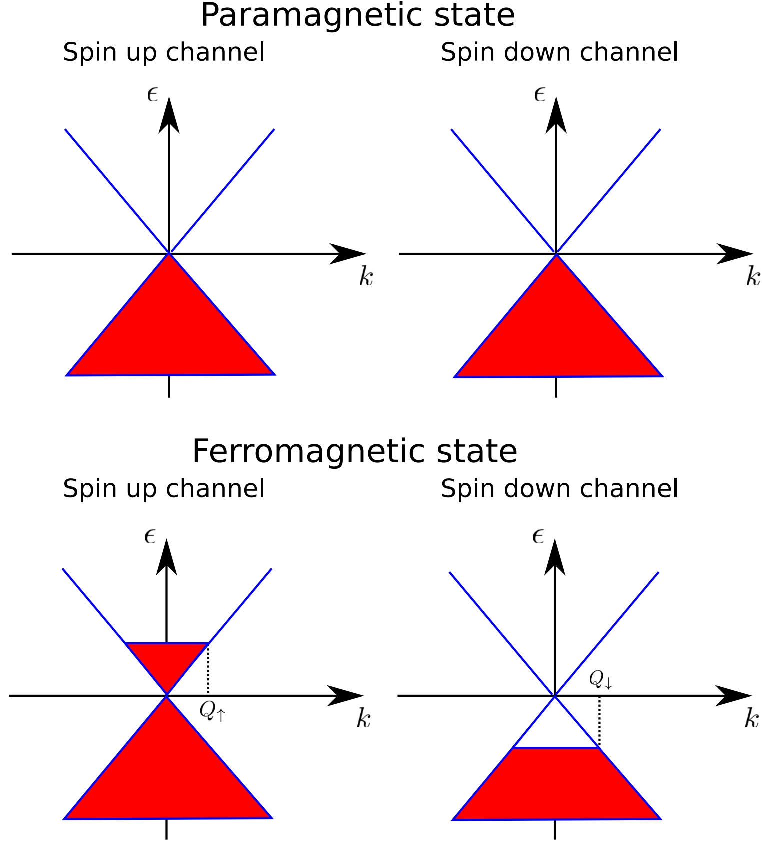


Figure 3.4: Un-doped Paramagnetic state (i.e. with half filled bands) versus a ferromagnetic state.

figure 3.4 and using equation (3.26) one can write down the exchange energy contribution coming from the ferromagnetic spin-up channel:

$$-\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\int_0^{2\pi} d\theta\Big\{\int_{k_c}^{0} kdk\int_{k_c}^{0} pdp\frac{1+\cos\theta}{|\mathbf{k}-\mathbf{p}|}+\int_0^{k_\uparrow} kdk\int_0^{k_\uparrow} pdp\frac{1+\cos\theta}{|\mathbf{k}-\mathbf{p}|}$$
$$+2\int_{k_c}^{0} kdk\int_0^{k_\uparrow} pdp\frac{1-\cos\theta}{|\mathbf{k}-\mathbf{p}|}\Big\},$$

where the three terms come from the hole-hole band ($\alpha=\alpha'=-1$), electron-electron band ($\alpha=\alpha'=1$) and electron-hole band ($\alpha=1,\alpha'=-1$ or $\alpha=-1,\alpha'=1$), respectively. The corresponding up channel yields

$$-\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\int_0^{2\pi} d\theta\Big\{\int_{k_c}^{k_\uparrow} kdk\int_{k_c}^{k_\uparrow} pdp\frac{1+\cos\theta}{|\mathbf{k}-\mathbf{p}|}\Big\}$$
$$=-\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\int_0^{2\pi} d\theta\Big\{\Big(\int_{k_c}^{0}\int_{k_c}^{k_0}+\int_0^{k_\uparrow}\int_0^{k_\uparrow}+2\int_{k_c}^{0}\int_0^{k_\uparrow}\Big) kdkpdp\frac{1+\cos\theta}{|\mathbf{k}-\mathbf{p}|}\Big\}.$$

The paramagnetic spin-up channel and spin-down channel contributions are equal leading to the total paramagnetic contribution

$$-\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\int_0^{2\pi} d\theta\Big\{2\int_{k_c}^{0} kdk\int_{k_c}^{0} pdp\frac{1+\cos\theta}{|\mathbf{k}-\mathbf{p}|}\Big\}.$$

When subtracting the paramagnetic contribution from the sum of the ferromagnetic contributions one finds

$$\Delta E_{ex} = -\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\Big\{ -4\int_0^{k_c} kdk \int_0^{k^\uparrow} pdp \int_0^{2\pi} d\theta \frac{1}{\sqrt{k^2+p^2-2kp\cos\theta}} + 2\int_0^{k_\uparrow} kdk \int_0^{k_\uparrow} pdp \int_0^{2\pi} d\theta \frac{1+\cos\theta}{\sqrt{k^2+p^2-2kp\cos\theta}}\Big\}.$$

Making the substitutions $x \to k_c x$ and $x \to k_\uparrow x$, where $x \in \{k, p\}$, in such a way that the integration boundaries run from 0 to 1 leads to

$$\Delta E_{ex} = -\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\Big\{ -4k_c k_\uparrow^2 \int_0^{1} kdk \int_0^{1} pdp \int_0^{2\pi} d\theta \frac{1}{\sqrt{k^2+\left(\frac{k_\uparrow}{k_c}\right)^2 p^2-2\left(\frac{k_\uparrow}{k_c}\right)kp\cos\theta}} + 2k_\uparrow^3\int_0^{1} kdk \int_0^{1} pdp \int_0^{2\pi} d\theta \frac{1+\cos\theta}{\sqrt{k^2+p^2-2kp\cos\theta}}\Big\}.$$

It is possible to define the triple integral in the first term as $R_0(k_\uparrow/k_c)$ and the triple integral in the second term as $R_1(1)$. Then

$$\Delta E_{ex} = -\frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\left\{-4k_c k_\uparrow^2 R_0\left(\frac{k_\uparrow}{k_c}\right) + 2k_\uparrow^3 R_1(1)\right\},$$

giving the total energy difference

$$\Delta E_a = \frac{A_c v_F}{3\pi}k_\uparrow^3 - \frac{A_c}{(2\pi)^2}\frac{e^2}{4\epsilon_0}\left\{-4k_c k_\uparrow^2 R_0\left(\frac{k_\uparrow}{k_c}\right) + 2k_\uparrow^3 R_1(1)\right\}. \tag{3.27}$$

The integrals $R_n(a)$ can be estimated or calculated numerically.

Introducing doping into the graphene structure, with one type of carrier in the ferromagnetic phase leads to the situation is shown in figure 3.5. A similar calculation as shown above yields the energy difference [12]

$$\Delta E_b = \frac{A_c v_F}{6\pi}(k_\uparrow^3 + k_\downarrow^3 - 2k_F^3) - \frac{A_c}{(2\pi)^2}\frac{e^2}{\epsilon_0}\Big[k_\uparrow^3 R_1(1) + k_\downarrow^3 R_1(1) - 2k_F^3 R_1(1) + 2k_c k_\downarrow^2 R_2\left(\frac{k_\downarrow}{k_c}\right) + 2k_c k_\uparrow^2 R_2\left(\frac{k_\uparrow}{k_c}\right) - 4k_c k_F^2 R_2\left(\frac{k_F}{k_c}\right)\Big].$$

Figure 3.6 shows a scenario where the graphene is still doped, but now with two types of charge carriers in the ferromagnetic phase. In this case it is found that [12]

$$\Delta E_c = \frac{A_c v_F}{6\pi}(k_\uparrow^3 - k_\downarrow^3 - 2k_F^3) - \frac{A_c}{(2\pi)^2}\frac{e^2}{\epsilon_0}\Big[k_\uparrow^3 R_1(1) + k_\downarrow^3 R_1(1) - 2k_F^3 R_1(1) - 2k_c k_\downarrow^2 R_2\left(\frac{k_\downarrow}{k_c}\right) + 2k_c k_\uparrow^2 R_2\left(\frac{k_\uparrow}{k_c}\right) - 4k_c k_F^2 R_2\left(\frac{k_F}{k_c}\right)\Big].$$

The spin polarization is defined as $s = k_\uparrow^2/(2\pi)$. For $k_\uparrow \ll k_c$ it turns out that the leading contribution in ΔE_a is $R_0(x) \approx -x\ln x$. Therefore, $\Delta E_a > 0$ for $x \ll 1$ and ferromagnetism is not favored for small magnetization s. For larger magnetizations where $k_c^2/s \sim 1$ one finds

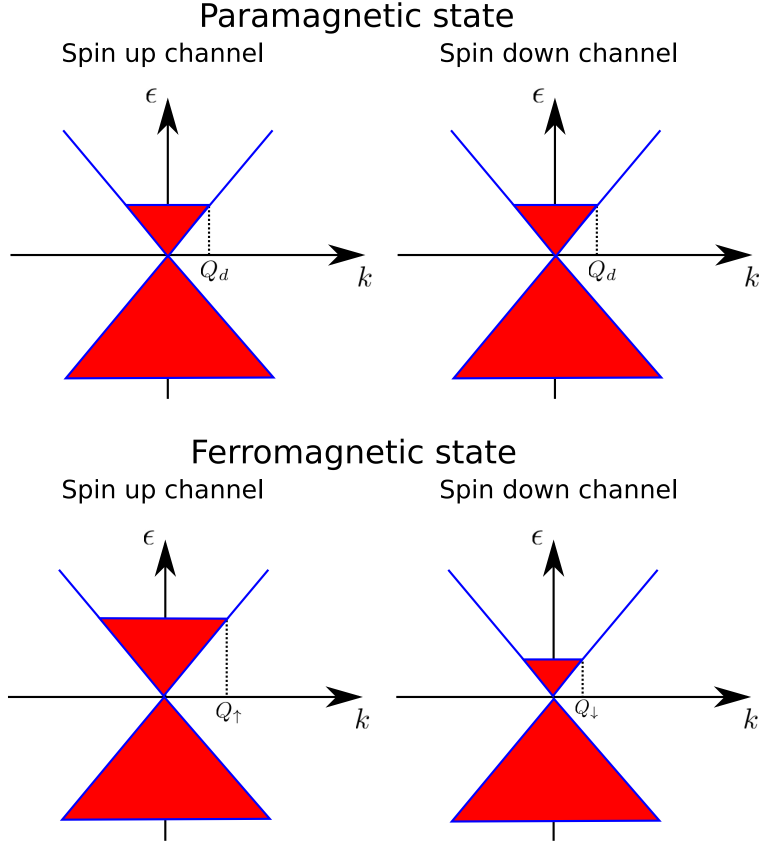


Figure 3.5: Doped paramagnetic state versus ferromagnetic state with one type of charge carrier.

that $k_c k_\uparrow^2 \sim (k_c/k_\uparrow)k_\uparrow^3 \sim k_\uparrow^3$. Using equation (3.27) it is possible to construct the condition for ferromagnetism

$$\frac{v_F}{3} - \frac{e^2}{\epsilon_0}\frac{1}{16\pi}\left[2R_1(1) - 4R_0(1)\right] \leq 0,$$

which can be rewritten as

$$g_c = \frac{e^2}{v_F \epsilon_0} \geq \frac{16\pi}{6R_1(1) - 12R_0(1)} \approx 5.3,$$

where g_c is the critical electron-electron coupling. The electron-electron coupling has been estimated to be $g \sim 2.8$ and is therefore far away from the critical coupling g_c that would result in ferromagnetism.

Using that $g = e^2/(v_F \epsilon_0)$, it is possible to determine v_F for a given coupling g. Furthermore, one can define $m \equiv sA_c$ to determine s given m and use that $n = \delta A_c$ to determine δ given n, where δ is the doping per unit area and n is the number of electrons away from half filling. Now it is possible to determine $k_\uparrow$ and $k_\downarrow$ by $k_\uparrow^2 = 2\pi(s+\delta)$ and $k_\downarrow^2 = 2\pi(s-\delta)$. In figure 3.7, ΔE_b as a function of m has been plotted for $n \approx 0.2$ with $g \approx 6$ (graph 1) and $g \approx 7.75$ (graph 2). It is evident that for $g \approx 6$ the graph never becomes negative within its range and ferromagnetism is not favored. However, for $g \approx 7.75$ it can be seen that ΔE_b does attain a negative value for large enough polarization m which leads to ferromagnetic behavior. In figure 3.7 it is clear that the transition from one graph to the other leads to a discontinuous drop to zero for the order parameter m. Thus, this is a first order phase transition.

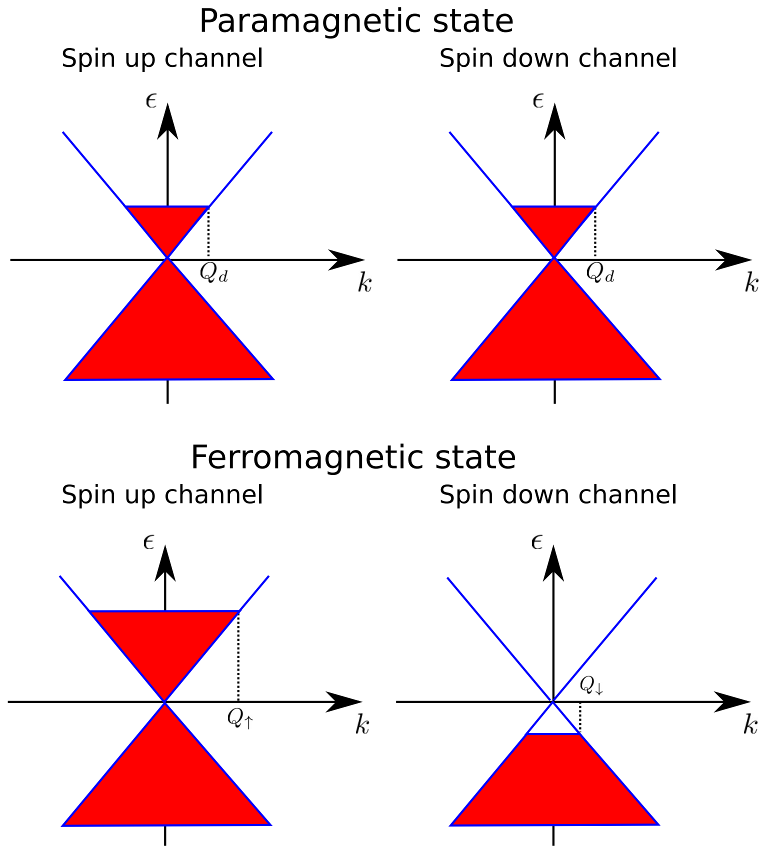


Figure 3.6: Doped paramagnetic state versus ferromagnetic state with two types of charge carriers.

A similar plot can be made for ΔE_c. This is shown in figure 3.8 for $n \approx 0.3$ with $g \approx 5.5$ (graph 3) and $g \approx 6.75$ (graph 4). In this case, for $g \approx 5.5$ a ferromagnetic phase cannot be obtained. However, for $g \approx 6.75$ the graph attains a minimum for a negative value of ΔE_c for a given polarization m. This signifies a possible phase transition from a paramagnetic to a ferromagnetic phase. Note that in transitioning from graph 3 to 4, the order parameter m will continuously tend to zero, which signifies a second order phase transition. This is in contrast to what was seen in figure 3.7. It is then evident that monolayer graphene can have both first and continuous phase transitions.

By finding at which doping level n the order parameter m reaches zero for a given value of the electron-electron coupling g one has found a single point in a phase diagram. The complete phase diagram is shown in figure 3.9.

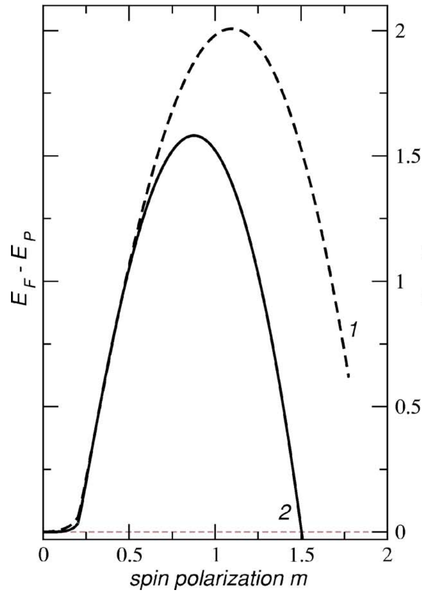


Figure 3.7: Behavior of $\Delta E_b(m) = [E_F - E_P](m)$ (plot taken from reference [12]).

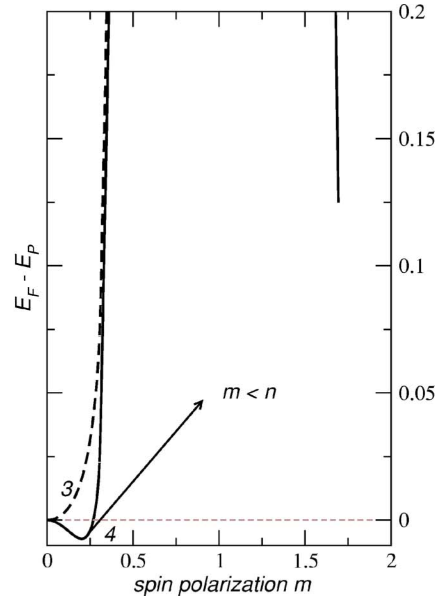


Figure 3.8: Behavior of $\Delta E_c(m) = [E_F - E_P](m)$ (plot taken from reference [12]).

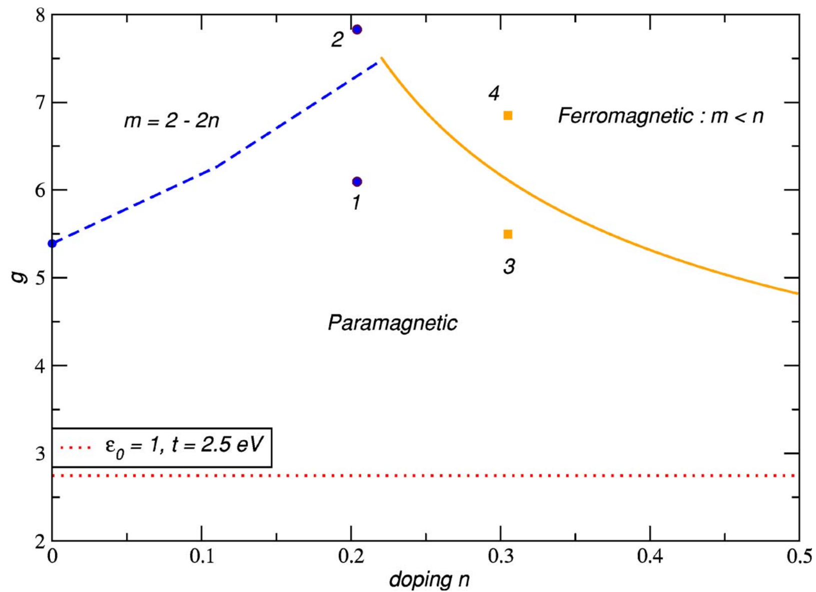


Figure 3.9: Complete phase diagram for monolayer graphene (figure taken from reference [12]).

CHAPTER 4

Bilayer graphene

Now that the ferromagnetic properties of monolayer graphene has been understood, it is time to add one more layer. This chapter will begin with a description of the bilayer Hamiltonian. With this as a basis, the single particle dispersion for bilayer graphene can be calculated. The dispersion will be used to derive the density of states, leading to an expression for the kinetic energy. Next, the expression for the exchange energy will be derived. This chapter will finish by showing the results of a numerical calculation of the phase diagram in the electron-electron coupling versus doping plane for bilayer graphene.

4.1 Band structure

4.1.1 Bilayer Hamiltonian

Figure 4.1 shows the atomic layout of AB stacked bilayer graphene. Only the nearest neighbor in-plane hopping parameter t and nearest neighbor inter-plane hopping parameter $t_\perp$ are considered in the calculations. The inter-plane hopping parameter describes the hopping energy associated to hopping between a_1 and a_2 sub lattice sites on the two separate layers. The Hamiltonian for such a system is

$$H = -t \sum_{\langle i,j \rangle, m, \sigma} [\hat{a}^\dagger_{i,\sigma,m} \hat{b}_{j,\sigma,m} + \text{h.c.}] - t_\perp \sum_{i,\sigma} [\hat{a}^\dagger_{i,\sigma,1} \hat{a}_{i,\sigma,2} + \text{h.c.}],$$

where $\hat{a}_{i,\sigma,m}$ annihilates an electron with spin $\sigma \in \{\uparrow, \downarrow\}$ at lattice site i on sub lattice a of layer $m \in \{1, 2\}$. Similarly $\hat{b}_{i,\sigma,m}$ annihilates the corresponding electron at sub lattice b. The Hamiltonian will be Fourier transformed by Fourier transforming the creation and annihilation operators. After an expansion around the K-points of the Brillouin zone, the Fourier transform of the creation operator is

$$\hat{a}_{i,\sigma,m} = \frac{1}{\sqrt{N}} \sum_{\mathbf{k},a} \hat{a}_{\mathbf{k},\sigma,m,a} e^{i\mathbf{k}\cdot\mathbf{r}_i},$$

where $a \in 1, 2$ refers to the two K-points of the Brillouin zone (K and K'). The expression for the annihilation operator $\hat{b}$ is similar. Inserting these expressions into the first term of the Hamiltonian yields

$$\sum_{\langle i,j \rangle, m, \sigma} \hat{a}^\dagger_{i,\sigma,m} \hat{b}_{j,\sigma,m} = \frac{1}{N} \sum_{\langle i,j \rangle, m, \sigma} \sum_{\mathbf{k}_1, \mathbf{k}_2} \sum_{a_1, a_2} \hat{a}^\dagger_{\mathbf{k_1},\sigma,m,a_1} \hat{b}_{\mathbf{k_2},\sigma,m,a_2} e^{-i\mathbf{r}_i\cdot\mathbf{k}_1 + i\mathbf{r}_j\cdot\mathbf{k}_2}.$$

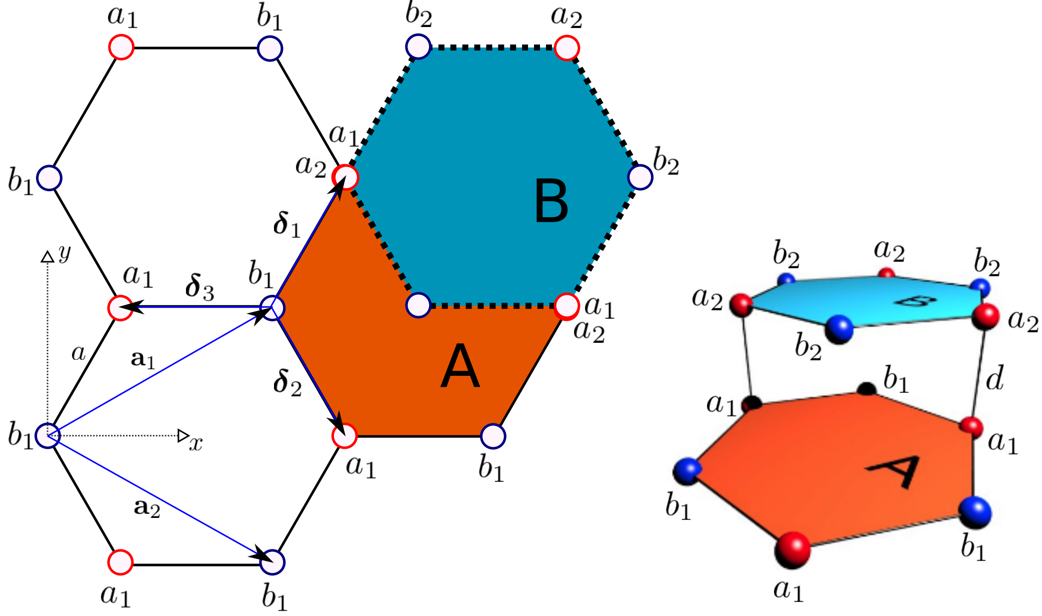


Figure 4.1: Structure of bilayer graphene.

Furthermore, since the sum over i and j only involve nearest neighbors it is allowed to use that $-\boldsymbol{\delta}_n = \mathbf{r}_i - \mathbf{r_j}$ where $n \in \{1, 2, 3\}$ depending on i, j. However, this is only true for the lower graphene layer. The a and b sub-lattice of the upper graphene layer has been interchanged compared to the lower graphene layer. Therefore, as can be seen from figure 4.1, $\mathbf{r}_i - \mathbf{r_j} = +\boldsymbol{\delta}_n$ for the upper graphene layer. This leads to

$$\sum_{\langle i,j\rangle,m,\sigma} \hat{a}^{\dagger}_{i,\sigma,m}\hat{b}_{j,\sigma,m} = \frac{1}{N}\sum_{\sigma}\sum_{\mathbf{k}_1,\mathbf{k}_2}\sum_{a_1,a_2}\hat{a}^{\dagger}_{\mathbf{k_1},\sigma,1,a_1}\hat{b}_{\mathbf{k_2},\sigma,1,a_2}\sum_{i=1}^{N}e^{-i\mathbf{r}_i\cdot(\mathbf{k}_1-\mathbf{k}_2)}\sum_{n=1}^{3}e^{-i\mathbf{k}_2\cdot\boldsymbol{\delta}_n}$$
$$+\frac{1}{N}\sum_{\sigma}\sum_{\mathbf{k}_1,\mathbf{k}_2}\sum_{a_1,a_2}\hat{a}^{\dagger}_{\mathbf{k_1},\sigma,2,a_1}\hat{b}_{\mathbf{k_2},\sigma,2,a_2}\sum_{i=1}^{N}e^{-i\mathbf{r}_i\cdot(\mathbf{k}_1-\mathbf{k}_2)}\sum_{n=1}^{3}e^{+i\mathbf{k}_2\cdot\boldsymbol{\delta}_n}.$$

The delta function is defined by

$$\sum_{i=1}^{N}e^{-i\mathbf{r}_i\cdot(\mathbf{k}_1-\mathbf{k}_2)} = N\delta_{\mathbf{k}_1,\mathbf{k}_2}\delta_{a_1,a_2},$$

where the factor δ_{a_1,a_2} arrises from each $\mathbf{k}_i$ being defined with respect to one of the K-points. Thus, if $\mathbf{k}_1$ is located at $a = 1$ and $\mathbf{k}_2 = \mathbf{k}_1$ then $\mathbf{k}_2$ must also be located at $a = 1$. This yields

$$\sum_{\langle i,j\rangle,m,\sigma} \hat{a}^{\dagger}_{i,\sigma,m}\hat{b}_{j,\sigma,m} = \sum_{\mathbf{k},\sigma,a}\hat{a}^{\dagger}_{\mathbf{k},\sigma,1,a}\hat{b}_{\mathbf{k},\sigma,1,a}\sum_{n=1}^{3}e^{-i\mathbf{k}\cdot\boldsymbol{\delta}_n} + \sum_{\mathbf{k},\sigma,a}\hat{a}^{\dagger}_{\mathbf{k},\sigma,2,a}\hat{b}_{\mathbf{k},\sigma,2,a}\sum_{n=1}^{3}e^{+i\mathbf{k}\cdot\boldsymbol{\delta}_n}. \quad (4.1)$$

The second term of the Hamiltonian is Fourier transformed in the same manner by substituting the Fourier transforms of $\hat{a}$ and $\hat{a}^{\dagger}$ leading to

$$\sum_{i,\sigma}\hat{a}^{\dagger}_{i,\sigma,1}\hat{a}_{i,\sigma,2} = \frac{1}{N}\sum_{i,\sigma}\sum_{\mathbf{k}_1,\mathbf{k}_2}\sum_{a_1,a_2}\hat{a}^{\dagger}_{\mathbf{k_1},\sigma,1,a_1}\hat{a}_{\mathbf{k_2},\sigma,2,a_2}e^{-i\mathbf{r}^1_i\cdot\mathbf{k}_1+i\mathbf{r}^2_i\cdot\mathbf{k}_2},$$

where $\mathbf{r}_i^1$ is the site vector for the lower graphene layer and $\mathbf{r}_i^2$ is the site vector for the upper graphene layer. Since $\mathbf{r}_i^1, \mathbf{r}_i^2 \in \mathbb{R}^2$ and are located at the same (x, y) coordinate for given i then $\mathbf{r}_i := \mathbf{r}_i^1 = \mathbf{r}_i^2$. Thus, by recognizing the delta function, the expression reduces to

$$\sum_{i,\sigma} \hat{a}^\dagger_{i,\sigma,1}\hat{a}_{i,\sigma,2} = \sum_{\mathbf{k},\sigma,a} \hat{a}^\dagger_{\mathbf{k},\sigma,1,a}\hat{a}_{\mathbf{k},\sigma,2,a}. \tag{4.2}$$

By using equations (4.1) and (4.2) the Hamiltonian can be written as

$$H = \sum_{\mathbf{k},\sigma,a} \hat{\Psi}^\dagger_{\mathbf{k},\sigma,a}\mathcal{H}\hat{\Psi}_{\mathbf{k},\sigma,a}$$

where

$$\hat{\Psi}^\dagger_{\mathbf{k},\sigma,a} = \begin{pmatrix} \hat{a}^\dagger_{\mathbf{k},\sigma,1,a} & \hat{b}^\dagger_{\mathbf{k},\sigma,1,a} & \hat{a}^\dagger_{\mathbf{k},\sigma,2,a} & \hat{b}^\dagger_{\mathbf{k},\sigma,2,a} \end{pmatrix}$$

and

$$\mathcal{H} = \begin{pmatrix} 0 & -t\sum_{n=1}^3 e^{-i\mathbf{k}\cdot\boldsymbol{\delta}_n} & -t_\perp & 0 \\ -t\sum_{n=1}^3 e^{i\mathbf{k}\cdot\boldsymbol{\delta}_n} & 0 & 0 & 0 \\ -t_\perp & 0 & 0 & -t\sum_{n=1}^3 e^{i\mathbf{k}\cdot\boldsymbol{\delta}_n} \\ 0 & 0 & -t\sum_{n=1}^3 e^{-i\mathbf{k}\cdot\boldsymbol{\delta}_n} & 0 \end{pmatrix}. \tag{4.3}$$

4.1.2 Bilayer dispersion

The dispersion relation can be calculated through diagonalizing the Hamiltonian. The resulting eigenvalues will be the band energies while the corresponding eigenstates will be the band eigenstates. The operators $\hat{\Psi}^\dagger_{\mathbf{k},\sigma,a,i}$ create electrons of momentum $\mathbf{k}$ and spin σ at K-point a on layer A or B of sub lattice a or b (labeled by $i \in \{0, 4\}$). After diagonalizing, the resulting rotated operator $\hat{\Phi}^\dagger_{\mathbf{k},\sigma,a,i} = [\mathcal{M}\hat{\Psi}_{\mathbf{k},\sigma,a,i}]^\dagger$ (where $\mathcal{M}$ diagonalizes $\mathcal{H}$) creates an electron on the band indicated by i. Thus, diagonalizing has effectively rotated the system from a layer oriented to a band oriented one. Equation (4.3) can be written in the form

$$\mathcal{H} := \begin{pmatrix} 0 & z & -t_\perp & 0 \\ z^* & 0 & 0 & 0 \\ -t_\perp & 0 & 0 & z^* \\ 0 & 0 & z & 0 \end{pmatrix}.$$

The eigenvalues λ_i are then determined by

$$\det \mathcal{H} = \lambda^4 - \lambda^2[2zz^* + t_\perp^2] + (zz^*)^2 = 0$$

Solving for λ^2 yields

$$\lambda^2 = \frac{2zz^* + t_\perp^2}{2} \pm \sqrt{\left(\frac{2zz^* + t_\perp^2}{2}\right)^2 - (zz^*)^2},$$

where

$$zz^* = t^2 \sum_{n,n'=1}^{3} e^{i\mathbf{k}\cdot(\boldsymbol{\delta}_n - \boldsymbol{\delta}'_n)}.$$

This gives four bands in the band structure:

$$\epsilon_1(\mathbf{k}) = +\sqrt{\frac{2zz^*(\mathbf{k}) + t_\perp^2}{2} - f(\mathbf{k})},\ \epsilon_2(\mathbf{k}) = -\sqrt{\frac{2zz^*(\mathbf{k}) + t_\perp^2}{2} - f(\mathbf{k})},$$

$$\epsilon_3(\mathbf{k}) = +\sqrt{\frac{2zz^*(\mathbf{k}) + t_\perp^2}{2} + f(\mathbf{k})},\ \epsilon_4(\mathbf{k}) = -\sqrt{\frac{2zz^*(\mathbf{k}) + t_\perp^2}{2} + f(\mathbf{k})},$$

where

$$f(\mathbf{k}) = \sqrt{\left(\frac{2zz^*(\mathbf{k}) + t_\perp^2}{2}\right)^2 - (zz^*(\mathbf{k}))^2}.$$

Figures 4.2 and 4.3 shows the resulting energy bands. These bands are plotted for $t_\perp = 0.05$ and $t = 0.43$ with the lattice constant set to $a = 1.56$ (see section 4.2.6 for explanation of units).

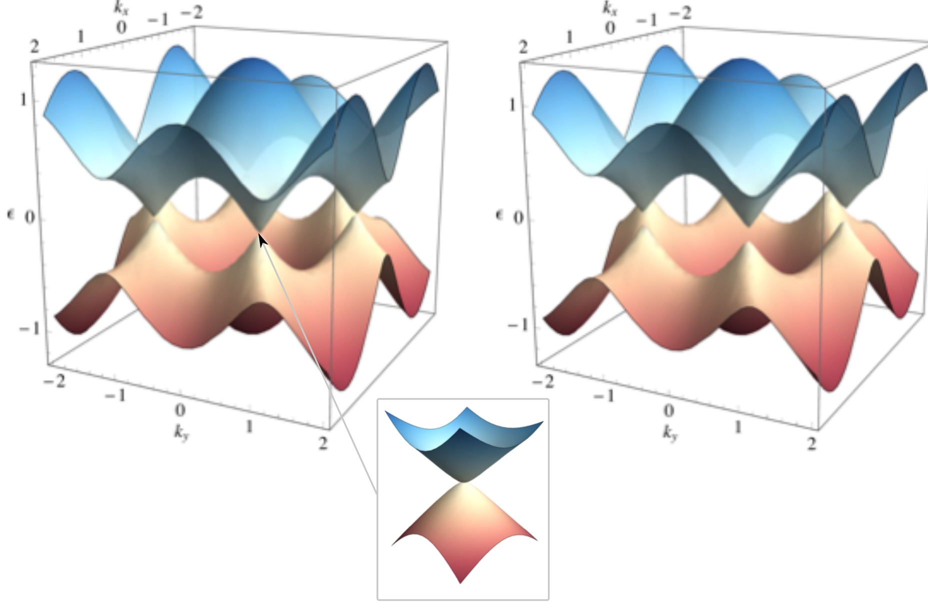


Figure 4.2: Energy bands $\epsilon_1(\mathbf{k})$ and $\epsilon_2(\mathbf{k})$ to the left and energy bands $\epsilon_3(\mathbf{k})$ and $\epsilon_4(\mathbf{k})$ to the right.

4.2 Ferromagnetism in bilayer graphene

In this section the doping versus electron-electron coupling phase diagram for bilayer graphene will be derived. The derivations of the kinetic and exchange energies are based on the work of reference [13].

4.2.1 Dispersion, low energy approximation

Eventually all calculated results will be in the low energy regime. It is therefore sufficient to Taylor expand the Hamiltonian to first order at one of the K-points. Note that

$$-t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{K}} = -t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{K}} = 0,$$

$$-\frac{\partial}{\partial k_x} t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = \frac{\partial}{\partial k_x} t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = -i\frac{3at}{2},$$

$$-\frac{\partial}{\partial k_y} t\sum_{m=1}^{3} e^{-i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = -\frac{\partial}{\partial k_y} t\sum_{m=1}^{3} e^{i\boldsymbol{\delta}_m\cdot\mathbf{k}}\bigg|_{\mathbf{k}=\mathbf{K}} = \frac{3at}{2}.$$

where the K-point $\mathbf{K}$ is

$$\mathbf{K} = \left(0, \frac{4\pi}{3\sqrt{3}a}\right).$$

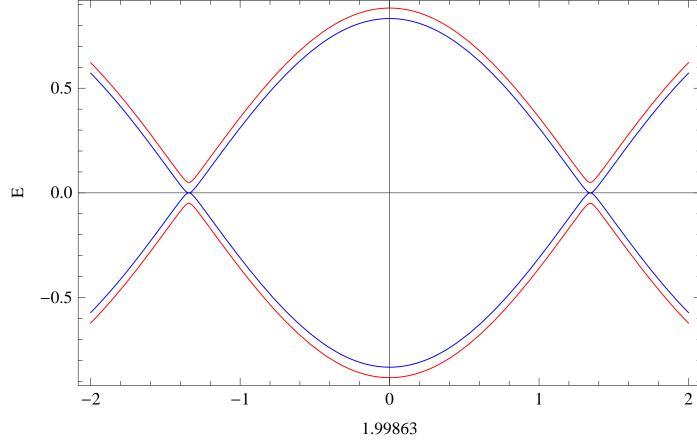


Figure 4.3: Cross section of energy bands $\epsilon_i(\mathbf{k})$ for $i = 1, 2, 3, 4$ along $k_y = 2\pi/(3\sqrt{3}a)$ (y-value of K-point).

Using these results, the entries of $\mathcal{H}$ from equation (4.3) can be expanded, yielding

$$\mathcal{H} = \begin{pmatrix} 0 & \frac{3at}{2}[k_y - ik_x] & -t_\perp & 0 \\ \frac{3at}{2}[k_y + ik_x] & 0 & 0 & 0 \\ -t_\perp & 0 & 0 & \frac{3at}{2}[k_y + ik_x] \\ 0 & 0 & \frac{3at}{2}[k_y - ik_x] & 0 \end{pmatrix}.$$

By writing this in Euler notation and using units such that $v_F = 3at/2 = 1$ and defining $\phi(\mathbf{k}) := \tan^{-1}(k_y/k_x)$, the Hamiltonian matrix becomes

$$\mathcal{H} = \begin{pmatrix} 0 & ke^{-i\phi(\mathbf{k})} & -t_\perp & 0 \\ ke^{i\phi(\mathbf{k})} & 0 & 0 & 0 \\ -t_\perp & 0 & 0 & ke^{i\phi(\mathbf{k})} \\ 0 & 0 & ke^{-i\phi(\mathbf{k})} & 0 \end{pmatrix}. \tag{4.4}$$

The matrix $\mathcal{H}$ is diagonalizable by the matrix $\mathcal{M}(\mathbf{k}) = \mathcal{M}_1(\mathbf{k})\mathcal{M}_2\mathcal{M}_3(\mathbf{k})$ [13], where

$$\mathcal{M}_1(\mathbf{k}) := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\phi(\mathbf{k})} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-i\phi(\mathbf{k})} \end{pmatrix},$$

$$\mathcal{M}_2 := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix},$$

$$\mathcal{M}_3(\mathbf{k}) := \begin{pmatrix} \cos[\theta(\mathbf{k})] & \sin[\theta(\mathbf{k})] & 0 & 0 \\ -\sin[\theta(\mathbf{k})] & \cos[\theta(\mathbf{k})] & 0 & 0 \\ 0 & 0 & \cos[\theta(\mathbf{k})] & -\sin[\theta(\mathbf{k})] \\ 0 & 0 & \sin[\theta(\mathbf{k})] & \cos[\theta(\mathbf{k})] \end{pmatrix}.$$

Thus, $\hat{\mathcal{H}}(\mathbf{k}) = \mathcal{M}^\dagger(\mathbf{k})\mathcal{H}(\mathbf{k})\mathcal{M}(\mathbf{k})$ is diagonal, provided the free parameter θ is chosen such that $\tan[2\theta] = 2k/t_\perp$. This choice ensures that all non-diagonal entries vanish. Since $\tan[2\theta] = 2k/t_\perp$ then

$$\cos[2\theta] = \frac{t_\perp}{\sqrt{4k^2 + t_\perp^2}},$$

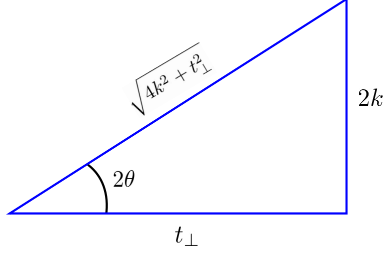


Figure 4.4: Visual representation of trigonometric identity.

as can be seen clearly from figure 4.4. Combining this result with the trigonometric half-angle identities yields

$$\cos^2\theta = \frac{1}{2}\left[1+\cos 2\theta\right] = \frac{1}{2}\left[1+\frac{t_\perp}{\sqrt{4k^2+t_\perp^2}}\right],$$
$$\sin^2\theta = \frac{1}{2}\left[1-\cos 2\theta\right] = \frac{1}{2}\left[1-\frac{t_\perp}{\sqrt{4k^2+t_\perp^2}}\right].$$

Next, the matrix multiplications can be performed which finally leads to

$$\hat{\mathcal{H}} = \mathrm{diag}[-t_\perp/2-g(k), -t_\perp/2+g(k), t_\perp/2+g(k), t_\perp/2-g(k)],$$

where $g(k)=\sqrt{t_\perp^2/4+k^2}$, giving the low energy dispersion at the chosen K-point:

$$\epsilon_1(k) = -t_\perp/2+g(k), \tag{4.5}$$
$$\epsilon_2(k) = t_\perp/2-g(k), \tag{4.6}$$
$$\epsilon_3(k) = t_\perp/2+g(k), \tag{4.7}$$
$$\epsilon_4(k) = -t_\perp/2-g(k). \tag{4.8}$$

Note that the numbering of the bands does not correspond to indices of the diagonal entries of $\hat{\mathcal{H}}$. This will become important later when the matrix $\mathcal{M}$ appears in the integrand of the exchange energy. It is then crucial that the correct bands are referenced with regards to filling of the electronic bands. The numbering of the bands is shown in figure 4.5 which also indicates the corresponding matrix entries along the diagonal of the diagonalized Hamiltonian.

4.2.2 Average kinetic energy

The density of states will be needed in order to calculate the average kinetic energy of the free electrons. To this end the total number of states N_α up to some energy ϵ will be calculated. Let $\alpha \in \{1,2,3,4\}$ denote the four bands produced by ϵ_α. Then,

$$N_\alpha = \sum_{\mathbf{k}} \Theta(\epsilon-\epsilon_\alpha(\mathbf{k})) = \frac{A_c}{(2\pi)^2}\int d\mathbf{k}\Theta(\epsilon-\epsilon_\alpha(\mathbf{k})) = \frac{A_c}{2\pi}\int_0^\infty dkk\Theta(\epsilon-\epsilon_\alpha(\mathbf{k}))$$
$$= \frac{A_c}{2\pi}\int_0^{k_\alpha(\epsilon)} dkk = \frac{A_c}{4\pi}(k_\alpha(\epsilon))^2.$$

Solving equations (4.5), (4.6), (4.7) and (4.8) with respect to k^2 and inserting the resulting expressions into the corresponding expressions for N_α yields

$$N_1(\epsilon) = \frac{A_c}{4\pi}\left[\left(\epsilon+\frac{t_\perp}{2}\right)^2-\frac{t_\perp^2}{4}\right],\ N_2(\epsilon) = \frac{A_c}{4\pi}\left[\left(\epsilon-\frac{t_\perp}{2}\right)^2-\frac{t_\perp^2}{4}\right],$$

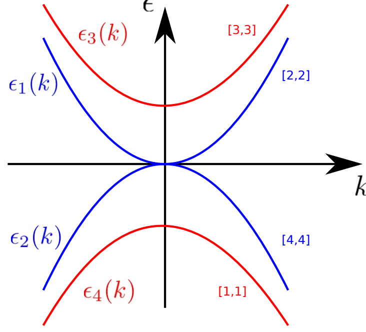


Figure 4.5: Numbering of the bands and the corresponding matrix entries $[i, i]$ along the diagonal of the diagonalized Hamiltonian.

$$N_3(\epsilon) = \frac{A_c}{4\pi}\left[\left(\epsilon - \frac{t_\perp}{2}\right)^2 - \frac{t_\perp^2}{4}\right],\ N_4(\epsilon) = \frac{A_c}{4\pi}\left[\left(\epsilon + \frac{t_\perp}{2}\right)^2 - \frac{t_\perp^2}{4}\right],$$

leading to the density of states

$$\mathcal{D}_1(\epsilon) = \frac{\partial N_1}{\partial \epsilon} = \frac{A_c}{2\pi}\left[\epsilon + \frac{t_\perp}{2}\right],\ \mathcal{D}_2(\epsilon) = \frac{\partial N_1}{\partial \epsilon} = \frac{A_c}{2\pi}\left[\epsilon - \frac{t_\perp}{2}\right],$$

$$\mathcal{D}_3(\epsilon) = \frac{\partial N_1}{\partial \epsilon} = \frac{A_c}{2\pi}\left[\epsilon - \frac{t_\perp}{2}\right],\ \mathcal{D}_4(\epsilon) = \frac{\partial N_1}{\partial \epsilon} = \frac{A_c}{2\pi}\left[\epsilon + \frac{t_\perp}{2}\right].$$

Note that $\mathcal{D}_3(\epsilon)$ is only valid for $\epsilon \geq \epsilon_3(0) = t_\perp$ and $\mathcal{D}_4(\epsilon)$ is only valid for $\epsilon \leq \epsilon_4(0) = -t_\perp$.

Consider the paramagnetic and ferromagnetic states shown in figure 4.6, where only bands 1 and 2 are shown. The average kinetic energy contribution $\Delta E_k(Q_\uparrow)$ resulting from filling band 1 in the ferromagnetic spin-up channel is given by

$$\begin{aligned}
\Delta E_k &= \int_0^{\epsilon_1(Q_\uparrow)} d\epsilon \epsilon \mathcal{D}_1(\epsilon) = \frac{A_c}{2\pi}\int_0^{\epsilon_1(Q_\uparrow)} d\epsilon \left[\epsilon + \frac{t_\perp}{2}\right]\epsilon = \frac{A_c}{2\pi}\left[\frac{1}{3}\epsilon^3 + \frac{1}{4}\epsilon^2 t_\perp\right]_0^{\epsilon_1(Q_\uparrow)} \\
&= \frac{A_c}{2\pi}\left[\frac{1}{3}\left(-\frac{t_\perp}{2} + \sqrt{\frac{t_\perp^2}{4} + Q_\uparrow^2}\right)^3 + \frac{1}{4}\left(-\frac{t_\perp}{2} + \sqrt{\frac{t_\perp^2}{4} + Q_\uparrow^2}\right)^2 t_\perp\right] \\
&= \frac{A_c}{2\pi}\left[\frac{(t_\perp^2/4 + Q_\uparrow^2)^{3/2}}{3} - \frac{t_\perp^3}{24} - \frac{Q_\uparrow^2 t_\perp^2}{4}\right]. \qquad (4.9)
\end{aligned}$$

Similarly, the expression for the average kinetic energy contribution resulting from the hole pocket of the ferromagnetic spin-down channel of figure 4.6 is given by $\Delta E_k(Q_\downarrow)$.

4.2.3 Exchange energy due to Coloumb interactions

Similarly to calculations done for monolayer graphene, it is possible to define field operators by expanding in the complete set of eigenstates formed by the Hamiltonian in equation (4.4). Let this complete set be denoted by $\{\Phi_{\mathbf{k},\sigma,i,a}\}$ where σ denotes the two possible spin states, i denotes the four possible energy bands and a denotes the two K-points of the Brillouin zone. Note that each vector in this set has four components. Then,

$$\hat{\Psi}(\mathbf{r}) = \frac{1}{\sqrt{A_c}}\sum_{\mathbf{k},\sigma,n,a} \Phi_{\mathbf{k},\sigma,n,a}(\mathbf{r})\hat{a}_{\mathbf{k},\sigma,n,a}, \qquad (4.10)$$

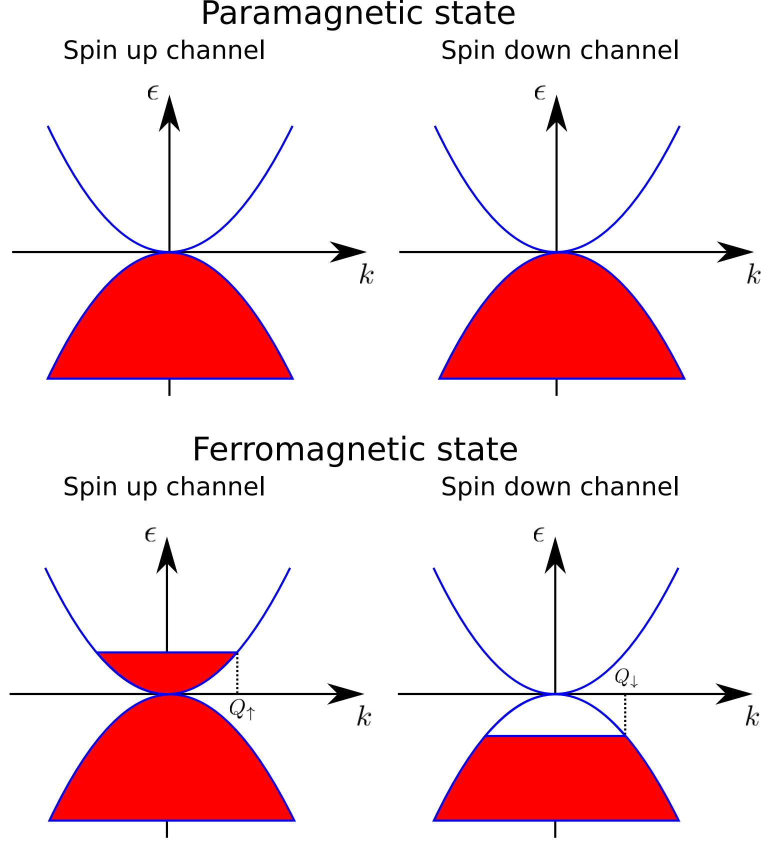


Figure 4.6: Half filled paramagnetic state and equal particle hole pocket ferromagnetic state.

satisfying

$$\left\{\hat{\Psi}_i(\mathbf{r}), \hat{\Psi}_j^\dagger(\mathbf{r}')\right\} = \delta_{ij}\delta^{(2)}(\mathbf{r}-\mathbf{r}'),$$

where $i, j \in \{1, 2, 3, 4\}$. The first two components of the field operator annihilates electrons on sub lattice a and b on layer 1 of the bilayer system, respectively. Similarly, the third and fourth components annihilate electrons on sub lattice a and b on layer 2. Thus, $\hat{\Psi}(\mathbf{r})$ can be decomposed into two-component operators $\{\hat{\Psi}_\alpha(\mathbf{r})\}$, where $\alpha \in \{1, 2\}$ denote the two layers of the graphene system. The two-component operators then satisfy

$$\left\{\hat{\Psi}_{\alpha,i}(\mathbf{r}), \hat{\Psi}_{\beta,j}^\dagger(\mathbf{r}')\right\} = \delta_{\alpha\beta}\delta_{ij}\delta^{(2)}(\mathbf{r}-\mathbf{r}'),$$

where now $i, j \in \{1, 2\}$.

The coulomb interactions in bilayer graphene consist of four separate interaction terms. The first two terms represent electron-electron interactions within the two graphene layers and the last two terms represent electron-electron interactions between the two layers. In second quantization the corresponding interaction Hamiltonian therefore has the form

$$\begin{aligned} H_I = \frac{1}{2}\int d\mathbf{r}_1 \int d\mathbf{r}_2 \big[& \hat{\Psi}_1^\dagger(\mathbf{r}_1)\hat{\Psi}_1^\dagger(\mathbf{r}_2)V_{11}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_1) \\ & +\hat{\Psi}_2^\dagger(\mathbf{r}_1)\hat{\Psi}_2^\dagger(\mathbf{r}_2)V_{22}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_1) \\ & +\hat{\Psi}_1^\dagger(\mathbf{r}_1)\hat{\Psi}_2^\dagger(\mathbf{r}_2)V_{12}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_1) \\ & +\hat{\Psi}_2^\dagger(\mathbf{r}_1)\hat{\Psi}_1^\dagger(\mathbf{r}_2)V_{21}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_1)\big]. \end{aligned} \tag{4.11}$$

The intralayer Coulomb potential is the same within the two layers. Therefore, $V_{11} = V_{22} = V$. The same reasoning holds for the interlayer Coulomb potential. Thus, $V_{12} = V_{21} = V_I$. It is more convenient to write the terms of H_I in the form $\hat{\Psi}^\dagger\hat{\Psi}\hat{\Psi}^\dagger\hat{\Psi}$ which effectively consist of two densities multiplied. To this end the anti-commutation relation can be used to write

$$\begin{aligned}
\hat{\Psi}^\dagger_\alpha(\mathbf{r}_1)\hat{\Psi}^\dagger_\beta(\mathbf{r}_2)\hat{\Psi}_\beta(\mathbf{r}_2)\hat{\Psi}_\alpha(\mathbf{r}_1) &= \sum_{ij} \hat{\Psi}^\dagger_{\alpha,i}(\mathbf{r}_1)\hat{\Psi}^\dagger_{\beta,j}(\mathbf{r}_2)\hat{\Psi}_{\beta,j}(\mathbf{r}_2)\hat{\Psi}_{\alpha,i}(\mathbf{r}_1) \\
&= \sum_{ij} -\hat{\Psi}^\dagger_{\alpha,i}(\mathbf{r}_1)\hat{\Psi}^\dagger_{\beta,j}(\mathbf{r}_2)\hat{\Psi}_{\alpha,i}(\mathbf{r}_1)\hat{\Psi}_{\beta,j}(\mathbf{r}_2) \\
&= \sum_{ij} -\hat{\Psi}^\dagger_{\alpha,i}(\mathbf{r}_1)\left[\delta_{\alpha\beta}\delta_{ij}\delta^{(2)}(\mathbf{r}_1-\mathbf{r}_2) - \hat{\Psi}_{\alpha,i}(\mathbf{r}_1)\hat{\Psi}^\dagger_{\beta,j}(\mathbf{r}_2)\right]\hat{\Psi}_{\beta,j}(\mathbf{r}_2) \\
&= \hat{\rho}_\alpha(\mathbf{r}_1)\hat{\rho}_\beta(\mathbf{r}_2) - \delta_{\alpha\beta}\delta_{ij}\delta^{(2)}(\mathbf{r}_1-\mathbf{r}_2)\sum_{ij}\hat{\Psi}^\dagger_{\alpha,i}(\mathbf{r}_1)\hat{\Psi}_{\beta,j}(\mathbf{r}_2),
\end{aligned}$$

where

$$\hat{\rho}_\alpha(\mathbf{r}) := \hat{\Psi}^\dagger_\alpha(\mathbf{r})\hat{\Psi}_\alpha(\mathbf{r}).$$

Inserting into H_I and integrating over $\mathbf{r}_1$ and $\mathbf{r}_2$ will result in four terms containing $V(0)$. These terms will all vanish due to a positive Jellium background resulting in

$$\begin{aligned}
H_I = \frac{1}{2}\int d\mathbf{r}_1 \int d\mathbf{r}_2 \big[& V(\mathbf{r}_1-\mathbf{r}_2)\left\{\hat{\rho}_1(\mathbf{r}_1)\hat{\rho}_1(\mathbf{r}_2) + \hat{\rho}_2(\mathbf{r}_1)\hat{\rho}_2(\mathbf{r}_2)\right\} \\
& + V_I(\mathbf{r}_1-\mathbf{r}_2)\left\{\hat{\rho}_1(\mathbf{r}_1)\hat{\rho}_2(\mathbf{r}_2) + \hat{\rho}_2(\mathbf{r}_1)\hat{\rho}_1(\mathbf{r}_2)\right\}\big],
\end{aligned}$$

where

$$V(\mathbf{r}_1-\mathbf{r}_2) = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|} = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}|} = \frac{e^2}{\epsilon_0}\frac{1}{r}, \tag{4.12}$$

and

$$V_I(\mathbf{r}_1-\mathbf{r}_2) = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|} = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}+\mathbf{d}|}\frac{e^2}{\epsilon_0}\frac{1}{\sqrt{r^2+d^2}}. \tag{4.13}$$

The vector $\mathbf{d}$ is defined to be a 3D vector because of its extension in the z-direction, thus it is defined as $\mathbf{d} := d[0,0,1]$.

In order to Fourier transform H_I, the potentials must be Fourier transformed. Using the normalization convention

$$V(\mathbf{k}) := \int d\mathbf{r} V(\mathbf{r}) e^{-i\mathbf{k}\cdot\mathbf{r}} \Rightarrow V(\mathbf{r}) = \frac{1}{(2\pi)^2}\int d\mathbf{k} V(\mathbf{k}) e^{i\mathbf{k}\cdot\mathbf{r}} \tag{4.14}$$

leads to

$$V(\mathbf{k}) = \frac{e^2}{\epsilon_0}\int d\mathbf{r}\frac{1}{r}e^{-i\mathbf{k}\cdot\mathbf{r}} = \frac{e^2}{\epsilon_0}\int_0^{2\pi} d\theta \int_0^\infty dr e^{-ikr\cos\theta}.$$

Substituting $r' = kr$ and subsequently renaming the integration variable $r' \to r$ yields

$$V(\mathbf{k}) = \frac{e^2}{\epsilon_0 k}\int_0^\infty dr \int_0^{2\pi} d\theta e^{-ir\cos\theta} = \frac{e^2}{\epsilon_0 k}\int_0^\infty dr 2\pi J_0(r) = \frac{2\pi e^2}{\epsilon_0}\frac{1}{k},$$

where $J_0(r)$ is the zeroth order Bessel function of the first kind. Similarly,

$$V_I(\mathbf{k}) = \frac{e^2}{\epsilon_0}\int_0^{2\pi} d\theta \int_0^\infty \frac{dr r e^{-ikr\cos\theta}}{\sqrt{r^2+d^2}} = \frac{e^2}{\epsilon_0 k}\int_0^{2\pi} d\theta \int_0^\infty \frac{dr r e^{-ir\cos\theta}}{\sqrt{r^2+(kd)^2}},$$

where in the last step the substitution $r' = kr$ was performed followed by $r' \to r$. This expression can be written in terms of the zeroth order Bessel function and the Bessel integration can subsequently be computed:

$$V_I(\mathbf{k}) = \frac{e^2}{\epsilon_0 k}\int_0^\infty \frac{drr}{\sqrt{r^2+(kd)^2}}\int_0^{2\pi} d\theta e^{-ir\cos\theta} = \frac{2\pi e^2}{\epsilon_0 k}\int_0^\infty \frac{drrJ_0(r)}{\sqrt{r^2+(kd)^2}} = \frac{2\pi e^2}{\epsilon_0}\frac{1}{k}e^{-kd}.$$

Furthermore, the density operators can be Fourier transformed according to

$$\hat{\rho}_\alpha(\mathbf{k}) := \int d\mathbf{r}\hat{\rho}(\mathbf{r})e^{-i\mathbf{k}\cdot\mathbf{r}} \Rightarrow \hat{\rho}_\alpha(\mathbf{r}) = \frac{1}{(2\pi)^2}\int d\mathbf{k}\hat{\rho}(\mathbf{k})e^{i\mathbf{k}\cdot\mathbf{r}}. \tag{4.15}$$

Inserting equations (4.14) and (4.15) into the first term of H_I gives

$$\begin{aligned} H_I^{(1)} &= \frac{1}{2(2\pi)^6}\int d\mathbf{k}_1 d\mathbf{k}_2 \int d\mathbf{q}\hat{\rho}_1(\mathbf{k_1})\hat{\rho}_1(\mathbf{k}_2)V(\mathbf{q})\int d\mathbf{r}_1 e^{i\mathbf{r}_1\cdot(\mathbf{k}_1+\mathbf{q})}\int d\mathbf{r_2}e^{i\mathbf{r}_2\cdot(\mathbf{k}_2-\mathbf{q})} \\ &= \frac{1}{2(2\pi)^2}\int d\mathbf{q}\hat{\rho}_1(-\mathbf{q})\hat{\rho}_1(\mathbf{q})V(\mathbf{q}) \to \frac{1}{2A_c}\sum_{\mathbf{q}}\hat{\rho}_1(-\mathbf{q})\hat{\rho}_1(\mathbf{q})V(\mathbf{q}) \\ &= \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\hat{\rho}_1(-\mathbf{q})\hat{\rho}_1(\mathbf{q})V(\mathbf{q}), \end{aligned}$$

where $\mathbf{q} = 0$ terms vanish against the positively charged Jellium background. A similar calculation holds for the three remaining terms of H_I such that

$$\begin{aligned} H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\big[&\hat{\rho}_1(-\mathbf{q})V(\mathbf{q})\hat{\rho}_1(\mathbf{q}) + \hat{\rho}_2(-\mathbf{q})V(\mathbf{q})\hat{\rho}_2(\mathbf{q}) \\ &+\hat{\rho}_1(-\mathbf{q})V_I(\mathbf{q})\hat{\rho}_2(\mathbf{q}) + \hat{\rho}_2(-\mathbf{q})V_I(\mathbf{q})\hat{\rho}_1(\mathbf{q})\big]. \end{aligned}$$

Now, H_I can be written in the matrix form

$$\begin{aligned} H_I &= \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\begin{pmatrix}\hat{\rho}_1(-\mathbf{q}) & \hat{\rho}_2(-\mathbf{q})\end{pmatrix}\begin{pmatrix}V(\mathbf{q}) & V_I(\mathbf{q}) \\ V_I(\mathbf{q}) & V(\mathbf{q})\end{pmatrix}\begin{pmatrix}\hat{\rho}_1(\mathbf{q}) \\ \hat{\rho}_2(\mathbf{q})\end{pmatrix} \\ &= \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\frac{2\pi e^2}{\epsilon_0 q}\begin{pmatrix}\hat{\rho}_1(-\mathbf{q}) & \hat{\rho}_2(-\mathbf{q})\end{pmatrix}\begin{pmatrix}1 & e^{-qd} \\ e^{-qd} & 1\end{pmatrix}\begin{pmatrix}\hat{\rho}_1(\mathbf{q}) \\ \hat{\rho}_2(\mathbf{q})\end{pmatrix}. \end{aligned}$$

The eigenvalues of

$$M = \begin{pmatrix}1 & e^{-qd} \\ e^{-qd} & 1\end{pmatrix}$$

are

$$\lambda_\pm = 1 \pm e^{-qd},$$

and the corresponding eigenstates are $(1,1)$ corresponding to λ_+ and $(1,-1)$ corresponding to λ_-. Thus, the unitary matrix that diagonalize M is

$$U = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1 \\ 1 & -1\end{pmatrix},$$

which allows the interaction Hamiltonian to be written as

$$\begin{aligned} H_I &= \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\frac{2\pi e^2}{\epsilon_0 q}\begin{pmatrix}\hat{\rho}_1(-\mathbf{q}) & \hat{\rho}_2(-\mathbf{q})\end{pmatrix}UU^\dagger MUU^\dagger\begin{pmatrix}\hat{\rho}_1(\mathbf{q}) \\ \hat{\rho}_2(\mathbf{q})\end{pmatrix} \\ &= \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\frac{2\pi e^2}{\epsilon_0 q}\frac{1}{\sqrt{2}}\begin{pmatrix}\hat{\rho}_+(-\mathbf{q}) & \hat{\rho}_-(-\mathbf{q})\end{pmatrix}\begin{pmatrix}1+e^{-qd} & 0 \\ 0 & 1-e^{-qd}\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}\hat{\rho}_+(\mathbf{q}) \\ \hat{\rho}_-(\mathbf{q})\end{pmatrix}, \end{aligned}$$

where

$$\hat{\rho}_{\pm}(\mathbf{q}) := \hat{\rho}_1(\mathbf{q}) \pm \hat{\rho}_2(\mathbf{q}).$$

Furthermore, by defining

$$V_{\pm}(\mathbf{q}) := \frac{2\pi e^2}{\epsilon_0 q}\left(1 \pm e^{-qd}\right)/2$$

the interaction Hamiltonian simplifies to

$$H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\sum_{\alpha=\pm}\hat{\rho}_{\alpha}(-\mathbf{q})V_{\alpha}(\mathbf{q})\hat{\rho}_{\alpha}(\mathbf{q}). \tag{4.16}$$

It will become necessary to write H_I in terms of $\hat{\Phi}_{\alpha} \equiv \mathcal{M}^{\dagger}\tilde{\Psi}_{\alpha}$ (where $\tilde{\Psi}$ is the dimensionless version of $\hat{\Psi}$) which contain the four band creation and annihilation operators as its entries. By employing the Fourier transform of equation (4.15)

$$\begin{aligned}
\hat{\rho}_{\pm}(\mathbf{q}) = \hat{\rho}_1(\mathbf{q}) \pm \hat{\rho}_2(\mathbf{q}) &= \int d\mathbf{r}[\hat{\rho}_1(\mathbf{r}) \pm \hat{\rho}_2(\mathbf{r})]e^{-i\mathbf{r}\cdot\mathbf{q}} \\
&= \int d\mathbf{r}\left[\hat{\Psi}_1^{\dagger}(\mathbf{r})\hat{\Psi}_1(\mathbf{r}) \pm \hat{\Psi}_2^{\dagger}(\mathbf{r})\hat{\Psi}_2(\mathbf{r})\right]e^{-i\mathbf{r}\cdot\mathbf{q}}.
\end{aligned}$$

The Fourier transform of $\hat{\Psi}$ is

$$\hat{\Psi}(\mathbf{k}) := \int d\mathbf{r}\hat{\Psi}(\mathbf{r})e^{i\mathbf{k}\cdot\mathbf{r}} \Rightarrow \hat{\Psi}(\mathbf{r}) = \frac{1}{(2\pi)^2}\int d\mathbf{k}\hat{\Psi}(\mathbf{k})e^{-i\mathbf{k}\cdot\mathbf{r}}.$$

This leads to

$$\hat{\rho}_{\pm}(\mathbf{q}) = \frac{1}{(2\pi)^2}\int d\mathbf{k}_1 d\mathbf{k}_2\left[\hat{\Psi}_1^{\dagger}(\mathbf{k}_1)\hat{\Psi}_1(\mathbf{k}_2) \pm \hat{\Psi}_2^{\dagger}(\mathbf{k}_1)\hat{\Psi}_2(\mathbf{k}_2)\right]\frac{1}{(2\pi)^2}\int d\mathbf{r}e^{i\mathbf{r}\cdot(\mathbf{k}_1-\mathbf{k}_2-\mathbf{q})}.$$

Recognizing the delta function, completing the sum over $\mathbf{k}_1$ and going to the discrete limit yields

$$\hat{\rho}_{\pm}(\mathbf{q}) = \frac{1}{A_c}\sum_{\mathbf{k}}\left[\hat{\Psi}_1^{\dagger}(\mathbf{k}+\mathbf{q})\hat{\Psi}_1(\mathbf{k}) \pm \hat{\Psi}_2^{\dagger}(\mathbf{k}+\mathbf{q})\hat{\Psi}_2(\mathbf{k})\right].$$

Notice that since $[\hat{\rho}(\mathbf{r})] = 1/m^2$ then $[\hat{\Psi}(\mathbf{r})] = \sqrt{[\hat{\rho}(\mathbf{r})]} = 1/m$ (in SI units). Then, from the Fourier transforms it follows that $[\hat{\rho}(\mathbf{k})] = 1$ and $[\hat{\Psi}(\mathbf{k})] = m$. The diagonalizing matrix $\mathcal{M}$ only operates on the dimensionless version of the operator $\hat{\Psi}(\mathbf{k})$. Denote this operator by

$$\tilde{\Psi}(\mathbf{k}) = \frac{1}{\sqrt{A_c}}\hat{\Psi}(\mathbf{k}).$$

Then, the dimensionless density can be written as

$$\hat{\rho}_{\pm}(\mathbf{q}) = \sum_{\mathbf{k}}\left[\tilde{\Psi}_1^{\dagger}(\mathbf{k}+\mathbf{q})\tilde{\Psi}_1(\mathbf{k}) \pm \tilde{\Psi}_2^{\dagger}(\mathbf{k}+\mathbf{q})\tilde{\Psi}_2(\mathbf{k})\right].$$

Writing this in block matrix form gives

$$\begin{aligned}
\hat{\rho}_{\pm}(\mathbf{q}) &= \sum_{\mathbf{k}}\begin{pmatrix}\tilde{\Psi}_1^{\dagger}(\mathbf{k}+\mathbf{q}) & \tilde{\Psi}_2^{\dagger}(\mathbf{k}+\mathbf{q})\end{pmatrix}\begin{pmatrix}\mathbf{1}_2 & 0 \\ 0 & \pm\mathbf{1}_2\end{pmatrix}\begin{pmatrix}\tilde{\Psi}_1(\mathbf{k}) \\ \tilde{\Psi}_2(\mathbf{k})\end{pmatrix} \\
&= \sum_{\mathbf{k}}\begin{pmatrix}\hat{\Phi}_1^{\dagger}(\mathbf{k}+\mathbf{q}) & \hat{\Phi}_2^{\dagger}(\mathbf{k}+\mathbf{q})\end{pmatrix}\mathcal{M}^{\dagger}(\mathbf{k}+\mathbf{q})\begin{pmatrix}\mathbf{1}_2 & 0 \\ 0 & \pm\mathbf{1}_2\end{pmatrix}\mathcal{M}(\mathbf{k})\begin{pmatrix}\hat{\Phi}_1(\mathbf{k}) \\ \hat{\Phi}_2(\mathbf{k})\end{pmatrix}.
\end{aligned}$$

Defining

$$\chi^{\pm}(\mathbf{k}+\mathbf{q},\mathbf{k}) = \mathcal{M}^{\dagger}(\mathbf{k}+\mathbf{q})\begin{pmatrix}\mathbf{1}_2 & 0 \\ 0 & \pm\mathbf{1}_2\end{pmatrix}\mathcal{M}(\mathbf{k})$$

leads to the simplified expression

$$\hat{\rho}_{\pm}(\mathbf{q}) = \sum_{\mathbf{k}} \hat{\Phi}^{\dagger}(\mathbf{k}+\mathbf{q})\chi^{\pm}(\mathbf{k}+\mathbf{q},\mathbf{k})\hat{\Phi}(\mathbf{k}).$$

Substituting this result into equation (4.16) results in

$$H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\sum_{\mathbf{p},\mathbf{p}'}\sum_{\alpha=\pm}\sum_{i,j,m,n=1}^{4} \hat{\Phi}_i^{\dagger}(\mathbf{p}-\mathbf{q})\chi_{ij}^{\alpha}(\mathbf{p}-\mathbf{q},\mathbf{p})\hat{\Phi}_j(\mathbf{p})V_{\alpha}(\mathbf{q}) \\ \times\hat{\Phi}_m^{\dagger}(\mathbf{p}'+\mathbf{q})\chi_{mn}^{\alpha}(\mathbf{p}'+\mathbf{q},\mathbf{p}')\hat{\Phi}_n(\mathbf{p}'). \tag{4.17}$$

Now that H_I is written in the energy band representation given by $\{\hat{\Phi}_i\}$, the total interaction energy can be calculated in the same representation (as apposed to the layer representation given by $\{\hat{\Psi}_i\}$). Let $:\ :$ denote normal ordering. Then, the interaction energy of a Fock state $|\mathbf{N}\rangle$ is given by

$$E_I = \langle\mathbf{N}|:H_I:|\mathbf{N}\rangle, \tag{4.18}$$

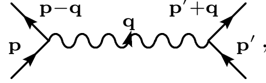
where

$$|\mathbf{N}\rangle = \prod_{\mathbf{k},n}\left[\hat{\Phi}_n^{\dagger}(\mathbf{k})\right]^{N_{\mathbf{k},n}}|0\rangle.$$

Note that since the system consists of Fermions $N_{\mathbf{k},n}\in\{0,1\}$. It is possible to represent the terms

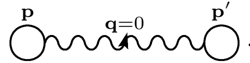
$$\mathcal{V} = \langle\mathbf{N}|:\hat{\Phi}_i^{\dagger}(\mathbf{p}-\mathbf{q})\hat{\Phi}_j(\mathbf{p})\hat{\Phi}_m^{\dagger}(\mathbf{p}'+\mathbf{q})\hat{\Phi}_n(\mathbf{p}'):|\mathbf{N}\rangle V_{\alpha}(\mathbf{q})$$

by the Feynman diagram

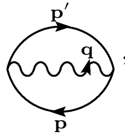


where time runs from the bottom of the diagram to the top of the diagram. In the above situation, the interaction potential V_{α} is the bare potential. Therefore, the interaction energy is an approximation that does not take into account higher order corrections above first order in the interaction. Thus, it is assumed that screening does not contribute significantly to the interaction energy. Later, it will be shown that this is indeed true for bilayer graphene but not for ABC-trilayer graphene.

Note that $\mathcal{V}$ will vanish if $\{\mathbf{p}',n\}$ differs from $\{\mathbf{p}'+\mathbf{q},m\}$ and $\{\mathbf{p}-\mathbf{q},i\}$ since this results in $\mathcal{V}=\langle\mathbf{N}_1|\mathbf{N}_2\rangle=0$. An identical argumentation shows that $\mathcal{V}=0$ if $\{\mathbf{p},j\}$ differs from $\{\mathbf{p}'+\mathbf{q},m\}$ and $\{\mathbf{p}-\mathbf{q},i\}$. Thus, in a sum over $\mathcal{V}$, there are only two types of terms that will survive. In the first type of term that survive $\{\mathbf{p}',n\}=\{\mathbf{p}'+\mathbf{q},m\}$ and $\{\mathbf{p},j\}=\{\mathbf{p}-\mathbf{q},i\}$. This leads to the Feynman diagram



This can be recognized as the Hartree contribution in perturbation theory. Since $V(0)=0$ in the Jellium model due to the positively charged background this term vanishes. In the second type of term $\{\mathbf{p}',n\}=\{\mathbf{p}-\mathbf{q},i\}$ and $\{\mathbf{p},j\}=\{\mathbf{p}'+\mathbf{q},m\}$. This leads to the Fock contribution



which equivalently is described by

$$\mathcal{V} = \langle\mathbf{N}|:\hat{\Phi}_i^{\dagger}(\mathbf{p}')\hat{\Phi}_j(\mathbf{p})\hat{\Phi}_j^{\dagger}(\mathbf{p})\hat{\Phi}_i(\mathbf{p}'):|\mathbf{N}\rangle V_{\alpha}(\mathbf{p}-\mathbf{p}').$$

Using

$$\left\{\hat{\Phi}_{\mathbf{p},i}, \hat{\Phi}^\dagger_{\mathbf{q},j}\right\} = \delta_{\mathbf{pq}}\delta_{ij}$$

combined with the knowledge that terms where $\mathbf{p} = \mathbf{p}'$ vanish, yields

$$\begin{aligned}\mathcal{V} &= -\langle \mathbf{N}|\hat{\Phi}^\dagger_i(\mathbf{p}')\hat{\Phi}^\dagger_j(\mathbf{p})\hat{\Phi}_j(\mathbf{p})\hat{\Phi}_i(\mathbf{p}')|\mathbf{N}\rangle V_\alpha(\mathbf{p}-\mathbf{p}') \\ &= -\langle \mathbf{N}|\hat{\Phi}^\dagger_j(\mathbf{p})\hat{\Phi}_j(\mathbf{p})\hat{\Phi}^\dagger_i(\mathbf{p}')\hat{\Phi}_i(\mathbf{p}')|\mathbf{N}\rangle V_\alpha(\mathbf{p}-\mathbf{p}').\end{aligned}$$

Recognizing the densities $n_i(\mathbf{p})$ and realizing that these, in principle, are dependent on both the spin σ and the K-point a one can write

$$\mathcal{V} = -n_{\sigma,j,a}(\mathbf{p})n_{\sigma,i,a}(\mathbf{p}')V_\alpha(\mathbf{p}-\mathbf{p}'),$$

where the densities (i.e. Fermi occupation functions) become theta functions in the zero temperature limit. Inserting $\mathcal{V}$ back into equation (4.17) and completing the $\mathbf{q}$, n and m sums with the Fock term restrictions $\{\mathbf{p}', n\} = \{\mathbf{p}-\mathbf{q}, i\}$ and $\{\mathbf{p}, j\} = \{\mathbf{p}'+\mathbf{q}, m\}$ leads to

$$E_I = -\frac{1}{2A_c}\sum_{\mathbf{p},\mathbf{p}'}\sum_{\alpha=\pm}\sum_{i,j=1}^{4} n_{\sigma,j,a}(\mathbf{p}')n_{\sigma,i,a}(\mathbf{p})\chi^\alpha_{ij}(\mathbf{p},\mathbf{p}')V_\alpha(\mathbf{p}'-\mathbf{p})\chi^\alpha_{ji}(\mathbf{p}',\mathbf{p}).$$

In order to take into account the possibility of differing momentum state occupations among the two spin channels σ and the two K-points a, these were introduced into the Fermi occupation function. Thus, an additional sum over a and σ are required to calculate the total interaction energy. Since only the Fock term, i.e. the exchange term, contributes to the interaction energy the interaction energy E_I will be referred to as the exchange energy E_{ex}. The exchange energy in the continuum limit therefore becomes

$$\begin{aligned}\frac{E_{ex}}{A_c} = -\frac{1}{2}\int\frac{d\mathbf{k}}{(2\pi)^2}\int\frac{d\mathbf{k}'}{(2\pi)^2}\sum_{\alpha=\pm}\sum_{i,j=1}^{4}\sum_{\sigma,a}&\chi^\alpha_{ij}(\mathbf{k},\mathbf{k}')\chi^\alpha_{ji}(\mathbf{k}',\mathbf{k}) \\ &\times n_{\sigma,a,i}(k)n_{\sigma,a,j}(k')V_\alpha(\mathbf{k}'-\mathbf{k}).\end{aligned} \tag{4.19}$$

The exchange integral becomes simpler to integrate if the variables are transformed to polar coordinates. The integral to be transformed has the form

$$I = \int d\mathbf{k}\int d\mathbf{k}'\xi(\mathbf{k},\mathbf{k}').$$

The inner integral

$$\int d\mathbf{k}'\xi(\mathbf{k},\mathbf{k}')$$

is integrated over all momenta $\mathbf{k}'$. Therefore, there exists a freedom to choose any basis for representing $\mathbf{k}$ and $\mathbf{k}'$. Let the basis be such that $\mathbf{k}||\hat{x}$ and denote the angle between $\mathbf{k}'$ and $\hat{x}$ by θ. This choice of basis has as a consequence that $\mathbf{k}$ is independent of any angle. Furthermore, the angle between $\mathbf{k}$ and $\mathbf{k}'$ is always θ. Thus,

$$\begin{aligned}I &= \int d\mathbf{k}\int_0^{2\pi}d\theta\int_0^\infty dk'k'\xi(k[1,0],k'[\cos\theta,\sin\theta]) \\ &= \int_0^{2\pi}d\theta'\int_0^\infty dkk\int_0^{2\pi}d\theta\int_0^\infty dk'k'\xi(k[1,0],k'[\cos\theta,\sin\theta]) \\ &= 2\pi\int_0^\infty dkk\int_0^{2\pi}d\theta\int_0^\infty dk'k'\xi(k[1,0],k'[\cos\theta,\sin\theta]).\end{aligned}$$

Introducing the cutoff Λ, equation (4.19) becomes

$$\frac{E_{ex}}{A_c} = -\frac{1}{2}\frac{1}{(2\pi)^3}\int_0^\Lambda dkk\int_0^\Lambda dk'k'\int_0^{2\pi} d\theta \sum_{\alpha=\pm}\sum_{i,j=1}^{4}\sum_{\sigma,a}\chi_{ij}^{\alpha}(k,k',\theta)\chi_{ji}^{\alpha}(k',k,\theta) \times n_{\sigma,a,i}(k)n_{\sigma,a,j}(k')V_\alpha(|\mathbf{k}'-\mathbf{k}|),$$

where the angular coordinate of $\mathbf{k}$ has been set to zero and the angular coordinate of $\mathbf{k}'$ to θ. Making the substitution $k = \tilde{k}\Lambda$ and $k' = \tilde{k}'\Lambda$ and then renaming the integration variables $\tilde{k}$ and $\tilde{k}'$ back to k and k' leads to

$$\frac{E_{ex}}{A_c} = -\frac{\Lambda^4}{2}\frac{1}{(2\pi)^3}\int_0^1 dkk\int_0^1 dk'k'\int_0^{2\pi} d\theta \sum_{\alpha=\pm}\sum_{i,j=1}^{4}\sum_{\sigma,a}\chi_{ij}^{\alpha}(k\Lambda,k'\Lambda,\theta)\chi_{ji}^{\alpha}(k'\Lambda,k\Lambda,\theta) \times n_{\sigma,a,i}(k\Lambda)n_{\sigma,a,j}(k'\Lambda)V_\alpha(|\mathbf{k}'-\mathbf{k}|), \quad (4.20)$$

where

$$V_\alpha(|\mathbf{k}'-\mathbf{k}|) := \frac{\pi e^2}{\Lambda\epsilon_0}\frac{1+\alpha e^{-d\Lambda\sqrt{k'^2-2kk'\cos\theta+k^2}}}{\sqrt{k'^2-2kk'\cos\theta+k^2}}.$$

Note that an effect of the last change of variables is to restrict the possible occupations of k and k' to a momentum between 0 and 1. This will be reflected in the Fermi occupation functions in that Q_d/Λ, $Q_\uparrow/\Lambda$ and $Q_\downarrow/\Lambda$ are also restricted to a range between 0 and 1.

4.2.4 The exchange energy difference

It is important to note that in the following standard band numbering of $\epsilon_i(k)$ will be used, as shown in figure 4.5, and therefore the entries of the matrix χ and the Fermi occupation functions must be redefined to correspond with this. For the half filling situation shown in figure 4.6 the fermi occupation functions are

$$n_{\uparrow,a,2}^{PM} = 1,\ n_{\downarrow,a,2}^{PM} = 1,$$
$$n_{\uparrow,a,4}^{PM} = 1,\ n_{\downarrow,a,4}^{PM} = 1$$

for the paramagnetic state. In this state the remaining occupation functions are set to zero. The variables $Q_\uparrow$ and $Q_\downarrow$ are the pocket sizes in momentum space. For the ferromagnetic phase the electron and hole pockets are equal in size such that $Q_\uparrow = Q_\downarrow = Q$ and the occupation functions become

$$n_{\uparrow,a,1}^{FM} = \Theta(Q-k),\ n_{\uparrow,a,2}^{FM} = 1,\ n_{\downarrow,a,1}^{FM} = 1-\Theta(Q-k)$$

and

$$n_{\uparrow,a,4}^{FM} = 1,\ n_{\downarrow,a,4}^{FM} = 1.$$

The remaining occupation functions are set to zero. Figure 4.7 shows a scenario with a doped paramagnetic phase and a ferromagnetic state with one type of charge carrier (i.e. electrons). This results in the fermi occupation functions:

$$n_{\uparrow,a,1}^{PM} = \Theta(Q_d-k),\ n_{\downarrow,a,1}^{PM} = \Theta(Q_d-k),$$
$$n_{\uparrow,a,2}^{PM} = 1,\ n_{\downarrow,a,2}^{PM} = 1,$$
$$n_{\uparrow,a,4}^{PM} = 1,\ n_{\downarrow,a,4}^{PM} = 1,$$
$$n_{\uparrow,a,1}^{FM} = \Theta(Q_\uparrow-k),\ n_{\downarrow,a,1}^{FM} = \Theta(Q_\downarrow-k),$$
$$n_{\uparrow,a,2}^{FM} = 1,\ n_{\downarrow,a,2}^{FM} = 1,$$
$$n_{\uparrow,a,4}^{FM} = 1,\ n_{\downarrow,a,4}^{FM} = 1.$$

Figure 4.8 shows a scenario with doped paramagnetic phase but with a ferromagnetic state containing both electron and hole charge carriers. The fermi occupation functions will in this case be:

$$
\begin{aligned}
&n^{PM}_{\uparrow,a,1} = \Theta(Q_d - k),\ n^{PM}_{\downarrow,a,1} = \Theta(Q_d - k),\\
&n^{PM}_{\uparrow,a,2} = 1,\ n^{PM}_{\downarrow,a,2} = 1,\\
&n^{PM}_{\uparrow,a,4} = 1,\ n^{PM}_{\downarrow,a,4} = 1,\\
&n^{FM}_{\uparrow,a,1} = \Theta(Q_\uparrow - k),\\
&n^{FM}_{\uparrow,a,2} = 1,\ n^{FM}_{\downarrow,a,2} = 1 - \Theta(Q_\downarrow - k),\\
&n^{FM}_{\uparrow,a,4} = 1,\ n^{FM}_{\downarrow,a,4} = 1.
\end{aligned}
$$

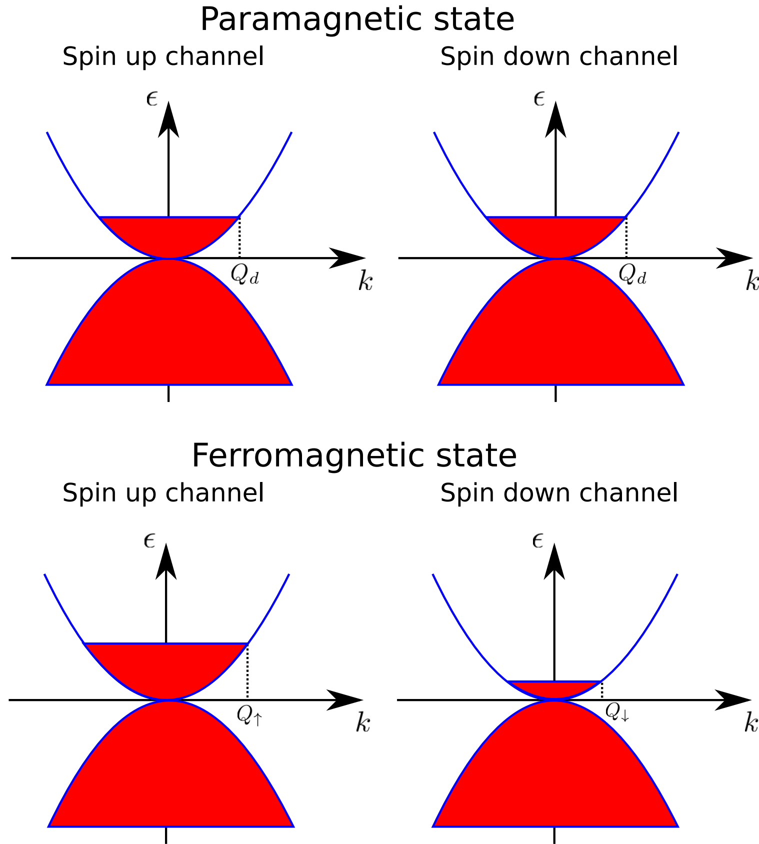


Figure 4.7: Doped paramagnetic state and one type of carrier in the ferromagnetic state.

In order to determine if the paramagnetic state or the ferromagnetic state is the most energetically favorable the difference between the exchange energy in the paramagnetic phase and the exchange energy in the ferromagnetic phase ΔE_{ex} must be calculated. To this end equation (4.20)

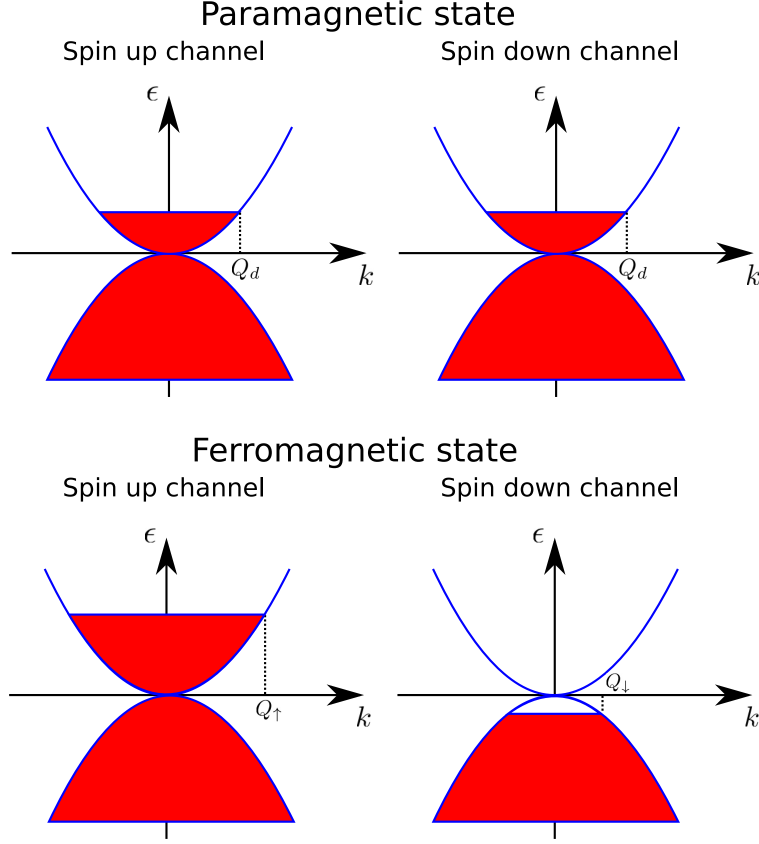


Figure 4.8: Doped paramagnetic state and two types of carriers in the ferromagnetic state.

is used to find that

$$\frac{\Delta E_{ex}}{A_c} = -\frac{\Lambda^4}{2}\frac{1}{(2\pi)^3}\int_0^1 dkk\int_0^1 dk'k'\int_0^{2\pi} d\theta \sum_{\alpha=\pm}\sum_{i,j=1}^{4}\sum_{\sigma,a}\chi^{\alpha}_{ij}(k\Lambda,k'\Lambda,\theta)\chi^{\alpha}_{ji}(k'\Lambda,k\Lambda,\theta)$$
$$\times\left[n^{FM}_{\sigma,a,i}(k\Lambda)n^{FM}_{\sigma,a,j}(k'\Lambda)-n^{PM}_{\sigma,a,i}(k\Lambda)n^{PM}_{\sigma,a,j}(k'\Lambda)\right]V_{\alpha}(|\mathbf{k'}-\mathbf{k}|). \qquad (4.21)$$

The integrand in the above integral has a singularity along $k = k'$ which makes it difficult to find a numerical solution. Performing a Duffy coordinate transformation will rotate the line of singularity to coincide with one of the coordinate axis. This makes it possible to move the singularities of the integrand to the integration boundaries such that it is easier to solve the numerical integration. In order to simplify the notation, make the following definitions:

$$F_{ij}(k,k',\theta) := -\frac{\Lambda e^2}{16\pi^2\epsilon_0}\sum_{\alpha=\pm}(\Lambda k)(\Lambda k')\chi^{\alpha}_{ij}(k\Lambda,k'\Lambda,\theta)\chi^{\alpha}_{ji}(k'\Lambda,k\Lambda,\theta)$$
$$\times\left[1+\alpha e^{-d\Lambda\sqrt{k'^2-2kk'\cos\theta+k^2}}\right] \qquad (4.22)$$

and

$$V_s(k,k',\theta) := \frac{1}{\sqrt{k'^2-2kk'\cos\theta+k^2}},$$

where the singular part of the Coulomb potential have been separated into its own function V_s. The exchange energy can now be written as

$$\frac{\Delta E_{ex}}{A_c} = \sum_{i,j=1,\sigma}^{4} \sum_{a} \int_0^1 dk \int_0^1 dk' \int_0^{2\pi} d\theta F_{ij}(k,k',\theta) V_s(k,k',\theta) \times \left[n^{FM}_{\sigma,a,i}(k\Lambda) n^{FM}_{\sigma,a,j}(k'\Lambda) - n^{PM}_{\sigma,a,i}(k\Lambda) n^{PM}_{\sigma,a,j}(k'\Lambda) \right].$$

The Fermi occupation functions corresponding to two types of carriers must now be inserted. This is followed by a change of variables such that the integration boundaries range from 0 to 1. This enables the use of a Duffy coordinate transformation. After applying a Duffy transformation all the singularities must be moved to one point along $k' = 1$. Since the double exponential algorithm for numerically solving integrals only allow singularities at the integration boundaries, the integrals must be split at $k' = 1$. The details of these calculations are shown in appendix D and leads to

$$\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = &\int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \big\{ 4Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\uparrow F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) \\
&- 8Q_d F_{11}(Q_d k, Q_d kk', \theta) - 8Q_d F_{12}(Q_d k, Q_d kk', \theta) \\
&+ 4Q_\uparrow F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) - 4Q_\downarrow F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
&- 8Q_d F_{14}(Q_d k, Q_d kk', \theta) \big\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} dk' 4Q_\uparrow \left\{ F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) + F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} dk' 4Q_\downarrow \left\{ F_{22}(Q_\downarrow k, Q_\downarrow kk', \theta) + F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} dk' 8Q_d \left\{ F_{12}(Q_d k, Q_d kk', \theta) + F_{14}(Q_d k, Q_d kk', \theta) \right\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\uparrow} dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\downarrow} dk' 4 \left\{ F_{22}(kk', k, \theta) + F_{24}(kk', k, \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_d} dk' 8 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta).
\end{aligned} \tag{4.23}$$

which is the expression for the exchange energy based on a ferromagnetic state of two types of charge carriers that will be evaluated numerically. The corresponding expression for one type of

charge carrier is

$$\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = & \int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \Big\{ 4Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\uparrow F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) \\
& + 4Q_\downarrow F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) + 4Q_\downarrow F_{11}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
& - 8Q_d F_{11}(Q_d k, Q_d kk', \theta) - 8Q_d F_{12}(Q_d k, Q_d kk', \theta) \\
& + 4Q_\uparrow F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\downarrow F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
& - 8Q_d F_{14}(Q_d k, Q_d kk', \theta) \Big\} V_s(k', \theta) \\
& + \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} dk' 4Q_\uparrow \left\{ F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) + F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) \right\} V_s(k', \theta) \\
& + \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} dk' 4Q_\downarrow \left\{ F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) + F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta) \right\} V_s(k', \theta) \\
& - \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} dk' 8Q_d \left\{ F_{12}(Q_d k, Q_d kk', \theta) + F_{14}(Q_d k, Q_d kk', \theta) \right\} V_s(k', \theta) \\
& + \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\uparrow} dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta) \\
& + \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\downarrow} dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta) \\
& - \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_d} dk' 8 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta).
\end{aligned} \tag{4.24}$$

4.2.5 The kinetic energy difference

Equation (4.9) gives the kinetic energy contribution for one electron pocket, or equivalently a hole pocket. In order to calculate the difference in energy due to transitioning from a paramagnetic to a ferromagnetic phase as shown in figures 4.6 the integral

$$\Delta E_k^T = \int_0^{\epsilon_1(Q_\uparrow)} d\epsilon \epsilon \mathcal{D}_1(\epsilon) + \int_{-\epsilon_2(Q_\downarrow)}^0 d\epsilon \epsilon \mathcal{D}_2(\epsilon)$$

must be calculated. Note that in contrast to the exchange energy calculations, the negative energy bands (valence bands) are integrated over hole pockets (i.e. not hole filling). This is due to a hole pocket being equivalent to an electron pocket which means that it is a hole pocket that gives a contribution to the kinetic energy. For the same reason the above equation can be written equivalently as

$$\Delta E_k^T = \frac{1}{Ac} \left\{ \Delta E_k(Q_\uparrow) + \Delta E_k(Q_\downarrow) \right\}.$$

In figures 4.7 and 4.8 the systems have a doped paramagnetic phase and in all cases

$$\frac{\Delta E_k^T}{A_c} = \Delta E_k(Q_\uparrow) + \Delta E_k(Q_\downarrow) - 2\Delta E_k(Q_d). \tag{4.25}$$

4.2.6 Bilayer phase diagram

Equations (4.23), (4.24) and (4.25) give the total change in energy between a paramagnetic and ferromagnetic phase

$$\frac{\Delta E^T}{A_c} = \frac{\Delta E_k^T}{A_c} + \frac{\Delta E_{ex}}{A_c}.$$

In the following, the bilayer g-Q_d phase diagram, where g is the electron-electron coupling constant, will be assembled. It is clear that equation (4.22) is independent of the cutoff if

$$\Lambda g \to g = \frac{e^2}{\epsilon_0} \frac{1}{v_F},$$

and $k\Lambda \to k$. This can be achieved by using units such that $v_F = \Lambda = 1$. Then,

$$v_F = \frac{3at}{2} = 1 \Rightarrow t = \frac{2}{3a}.$$

Furthermore,

$$\Lambda^2 = \frac{2\pi}{A_c} = \frac{4\pi}{\sqrt{27}a^2} = 1.$$

Solving these equations for t and a gives

$$t = \frac{2}{3}\sqrt{\frac{\sqrt{27}}{4\pi}} \approx 0.4287,$$

and

$$a = \sqrt{\frac{4\pi}{\sqrt{27}}} \approx 1.5551.$$

It has been found that $d \approx 2.4a$ where $a \approx 1.42$Å, $t \approx 3eV$ and $t_\perp \approx 0.35eV$ [13]. Thus, in the above choice of units

$$d \approx 2.4a \approx 3.7.$$

The ratio between t and $t_\perp$ can be used to find

$$\frac{t}{t_\perp} = \frac{3eV}{0.35eV} = \frac{0.4287}{t_\perp} \Rightarrow t_\perp \approx 0.4287 \cdot \frac{0.35eV}{3eV} \approx 0.05.$$

Furthermore, the units will be adjusted such that $\epsilon_0 = 1$. Thus $g = e^2$ where e is the electron charge.

At this point there are four free variables in equations (4.23), (4.24) and (4.25). These are $g, Q_d, Q_\uparrow$ and $Q_\downarrow$. However, particle conservation during transitions between paramagnetic and ferromagnetic states have not yet been considered. Thus, there is a relation between $Q_d, Q_\uparrow$ and $Q_\downarrow$. Looking at the case where the ferromagnetic state only has one type of carrier as shown in figure 4.7, the number of electrons in the paramagnetic spin-up channel is (i.e electrons of band 1 and electrons of band 2)

$$\begin{aligned} N_\uparrow^{PM} &= \sum_{\mathbf{k}} [1 + \Theta(Q_d - k)] \approx \frac{A_c}{(2\pi)^2} \int d\mathbf{k}[1 + \Theta(Q_d - k)] \\ &= N + \frac{A_c}{2\pi} \int_0^\infty dkk\Theta(Q_d - k) = N + \frac{A_c}{4\pi} Q_d^2, \end{aligned}$$

where $N := A_c/2\pi \int_0^\infty dkk$. Correspondingly, in the paramagnetic down channel one finds that

$$N_\downarrow^{PM} = N + \frac{A_c}{4\pi} Q_d^2.$$

The same methods can be applied to calculate the number of electrons and holes present in the ferromagnetic spin-up and spin-down channels. Thus,

$$N_\uparrow^{FM} = N + \frac{A_c}{4\pi} Q_\uparrow^2 \text{ and } N_\downarrow^{FM} = N + \frac{A_c}{4\pi} Q_\downarrow^2.$$

Since $N_\uparrow^{PM} + N_\downarrow^{PM} = N_\uparrow^{FM} + N_\downarrow^{FM}$ one finds that

$$2Q_d^2 = Q_\uparrow^2 + Q_\downarrow^2. \tag{4.26}$$

Note that bands 3 and 4 always have equal filling in all possible states and thus need not be considered.

For a ferromagnetic phase with two charge carriers the particle count is

$$N_\uparrow^{FM} = N + \frac{A_c}{4\pi} Q_\uparrow,$$

in the spin-up channel, which is identical to the case of one charge carrier. However, for the spin-down channel, looking at figure 4.8, it can be seen that

$$N_\downarrow^{FM} = \sum_{\mathbf{k}} [1 - \Theta(Q_\downarrow - k)] = N - \frac{A_c}{4\pi} Q_\downarrow.$$

Thus, particle conservation, in this case, yields

$$2Q_d^2 = Q_\uparrow^2 - Q_\downarrow^2. \tag{4.27}$$

$\Delta E^T/A_c$ will be plotted as a function of particle-hole pocket sizes. To accomplish this, equations (4.26) and (4.27) are parameterized by letting $x := Q_\downarrow^2$. Then, equation (4.26) results in

$$\begin{aligned} Q_\uparrow^2 &= 2Q_d^2 - x, \\ Q_\downarrow^2 &= x, \end{aligned} \tag{4.28}$$

while equation (4.27) results in

$$\begin{aligned} Q_\uparrow^2 &= 2Q_d^2 + x, \\ Q_\downarrow^2 &= x. \end{aligned} \tag{4.29}$$

Furthermore, it is preferable to plot $\Delta E^T/A_c$ as a function of x with both one and two types of charge carriers in the same plot. Note that $Q_i \in (0, 1]$ and therefore $x \in (0, 1]$ such that $x > 0$. It is therefore possible to use the negative x-axis to display $\Delta E^T/A_c$ of states with two types of carriers (and the positive x-axis for states with one type of carrier). Furthermore, note that for $x = 0$ the parametrizations for one and two types of carriers coincide. In order to parameterize states with two types of carriers on $x < 0$ then equation (4.29) must be changed to

$$\begin{aligned} Q_\uparrow^2 &= 2Q_d^2 + |x|, \\ Q_\downarrow^2 &= |x|. \end{aligned} \tag{4.30}$$

Note that because $Q_\uparrow^2 > 0$ then equation (4.28) leads to $x < 2Q_d^2$ for one type of charge carrier. Similarly, since $Q_\uparrow^2 \leq 1$ then equation (4.30) implies that $0 < |x| \leq 1 - 2Q_d^2$ and therefore $Q_d < 1/\sqrt{2}$ for two types of charge carriers.

In the parametrization above $Q_\downarrow = Q_\uparrow = Q_d$ at the point where $x = Q_d^2$. This is the point where the system is not perturbed, i.e. where the system "transitions" from a paramagnetic to the same paramagnetic phase. In order to determine if the phase transition is of first or second order it will be more instructive to map x to a new parametrization x' such that $x' = 0$ leads to no perturbation of the system. This is achieved by letting

$$x := x' + Q_d^2.$$

After imposing particle conservation the system has three degrees of freedom Q_d, g and x'. As stated earlier the system is in the ferromagnetic phase if $\Delta E^T/A_c < 0$ at its minimum, otherwise the system is in the paramagnetic phase. The critical point is defined to be the value of Q_d where the system transitions between the paramagnetic and ferromagnetic phase for a given value of g. In order to plot one critical point in a g-Q_d phase diagram it is necessary to find Q_d such that the minimum of $\Delta E^T/A_c(x')$ is zero. To accomplish this $\Delta E^T/A_c$ versus x' will be plotted for eight values of Q_d close to where the transition is expect to occur. A binary search pattern is employed to find the value of Q_d to use for the next plot such that the curves quickly converge to the critical curve.

The double exponential algorithm is used to calculate integrals occurring in the expression for $\Delta E^T/A_c$. For each function $\Delta E^T/A_c(x')$ the minimum is found by finding the minimum of a second order polynomial interpolation of the set of three points that include the minimum. Eight curves are calculated and thus eight such minima are found. By interpolation of the eight minima an approximate value of Q_d is found for the current value of g.

This procedure is demonstrated for $g = 6$. Figure 4.9 shows a plot of $\Delta E^T/A_c$ versus $-x'$ for some values of Q_d. The thick line is the curve at critical doping level. Since this curve has minima at both $-x' = 0$ and $-x' \neq 0$ the phase transition is of first order (i.e. discontinuous). All phase points in the range $g \in (0, 6]$ are seen to behave in the same manner. Also notice that for a non-doped system the minimum of $\Delta E^T/A_c$ is always negative which leads to a ferromagnetic phase.

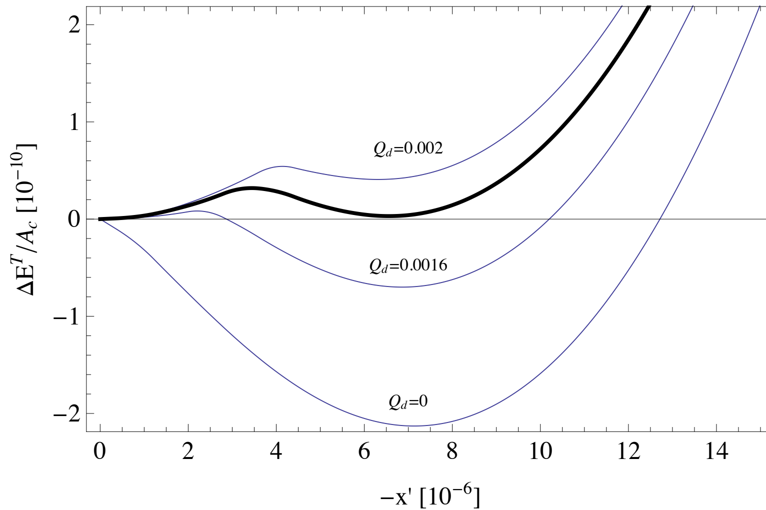


Figure 4.9: $\Delta E^T/A_c$ versus $-x'$ for $g = 6$.

Figure 4.11 shows the minima $\Delta E^T_{min}/A_c$ that is found for the eight curves in figure 4.10 as a function of doping Q_d. In figure 4.10 each curve is plotted as a function of x (i.e. not x'). Thus, it is apparent that all minima are located at $x < 0$, which implies that the preferred ferromagnetic states have two types of charge carriers. This indicates why the phase transition is of the first order; the paramagnetic phase only has one type of carrier and it is a discontinuous process to transition to a phase with two types of carriers. The critical doping is found by reading off the value of Q_d where $\Delta E^T_{min} = 0$ in figure 4.11. At $g = 6$ one finds $Q_d^{crit} \approx 0.001875$.

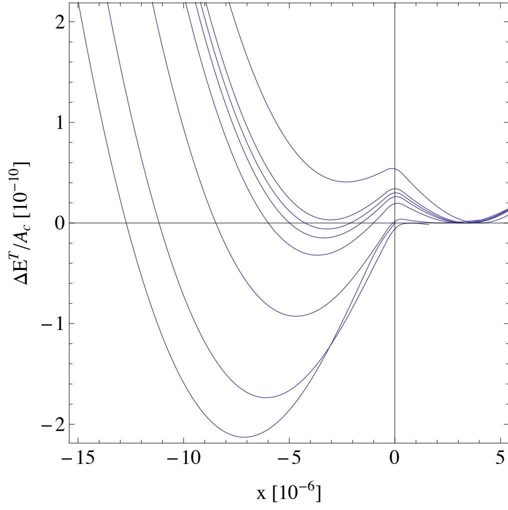


Figure 4.10: $\Delta E^T/A_c$ versus x for $g = 6$.

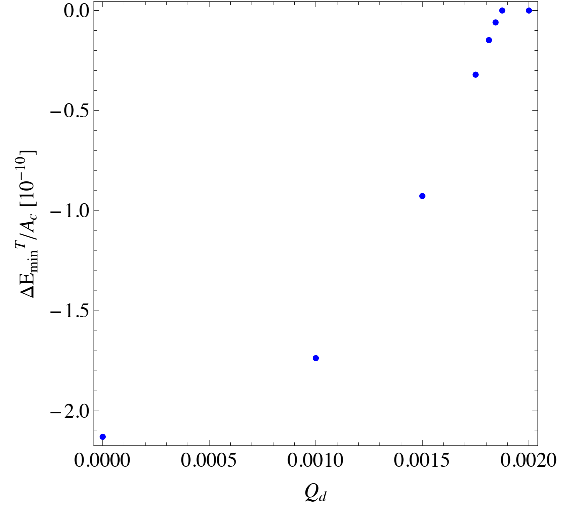


Figure 4.11: $\Delta E^T_{min}/A_c$ versus Q_d for $g = 6$.

Repeating this procedure for several values of g running from $g = 0.1$ to $g = 6$ produces the phase diagram shown in figure 4.12. The dots indicate the values found numerically while the continuous curve is a second order interpolation between the numerically found values. Note that the doping in the phase diagram is found by $n = Q_d^2/2$ which is in units of electrons per carbon atom.

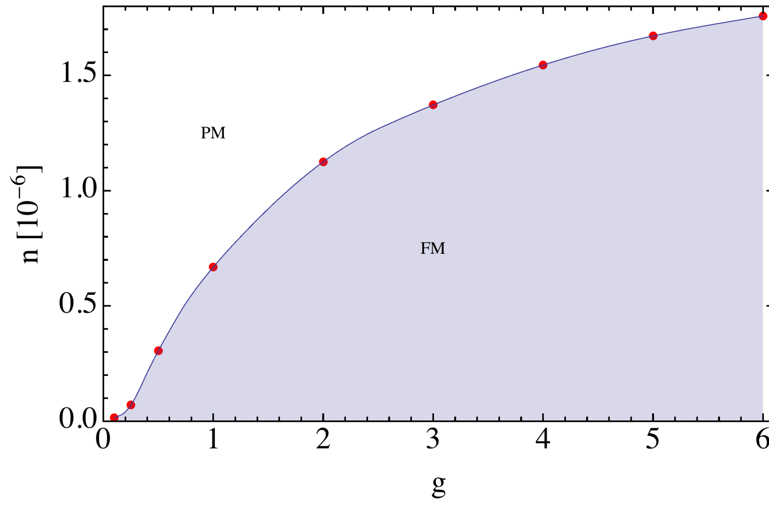


Figure 4.12: Phase diagram for bilayer graphene: doping n versus electron-electron coupling g.

CHAPTER 5

ABC-Trilayer graphene

In previous chapters, the ferromagnetic properties of monolayer and bilayer graphene were calculated based on existing literature. This chapter will go beyond existing literature and compute the ferromagnetic properties of ABC-stacked trilayer graphene, by utilizing the same techniques as was used for monolayer and bilayer graphene. Once the Hamiltonian has been defined and the band structure for ABC-trilayer has been calculated, this will be used to find an expression for the kinetic energy. The expression for the exchange energy is calculated, based on the low energy Hamiltonian of ABC-trilayer graphene, where an expression very similar to the bilayer exchange energy is found. Eventually, the phase diagram is calculated numerically. The actual program is shown in appendix G. At the end of the chapter, the effects of Coloumb screening is outlined.

5.1 Band structure

5.1.1 ABC-trilayer Hamiltonian

Figure 5.1 shows the structure of ABC-trilayer graphene. In analogy with bilayer graphene, the tight binding Hamiltonian corresponding to this structure is

$$H = -t \sum_{\langle i,j \rangle, m, \sigma} \left[\hat{a}^{\dagger}_{i,\sigma,m} \hat{b}_{j,\sigma,m} + \text{h.c.} \right] - t_{\perp} \sum_{i,\sigma} \left[\hat{a}^{\dagger}_{i,\sigma,1} \hat{a}_{i,\sigma,2} + \text{h.c.} \right] - t_{\perp} \sum_{i,\sigma} \left[\hat{b}^{\dagger}_{i,\sigma,2} \hat{b}_{i,\sigma,3} + \text{h.c.} \right],$$

where only nearest neighbor hopping parameters are included. Note that m denotes the three layers while σ denotes the spin, in analogy with the bilayer Hamiltonian. The Hamiltonian can be Fourier transformed in the same manner as for the bilayer Hamiltonian and written in the form

$$H = \sum_{\mathbf{k},\sigma,a} \hat{\Psi}^{\dagger}_{\mathbf{k},\sigma,a} \mathcal{H} \hat{\Psi}_{\mathbf{k},\sigma,a},$$

where

$$\hat{\Psi}^{\dagger}_{\mathbf{k},\sigma,a} = \begin{pmatrix} \hat{a}^{\dagger}_{\mathbf{k},\sigma,1,a} & \hat{b}^{\dagger}_{\mathbf{k},\sigma,1,a} & \hat{a}^{\dagger}_{\mathbf{k},\sigma,2,a} & \hat{b}^{\dagger}_{\mathbf{k},\sigma,2,a} & \hat{a}^{\dagger}_{\mathbf{k},\sigma,3,a} & \hat{b}^{\dagger}_{\mathbf{k},\sigma,3,a} \end{pmatrix},$$

$$f(\mathbf{k}) := -t \sum_{n=1}^{3} e^{i\mathbf{k}\cdot\boldsymbol{\delta}_n},$$

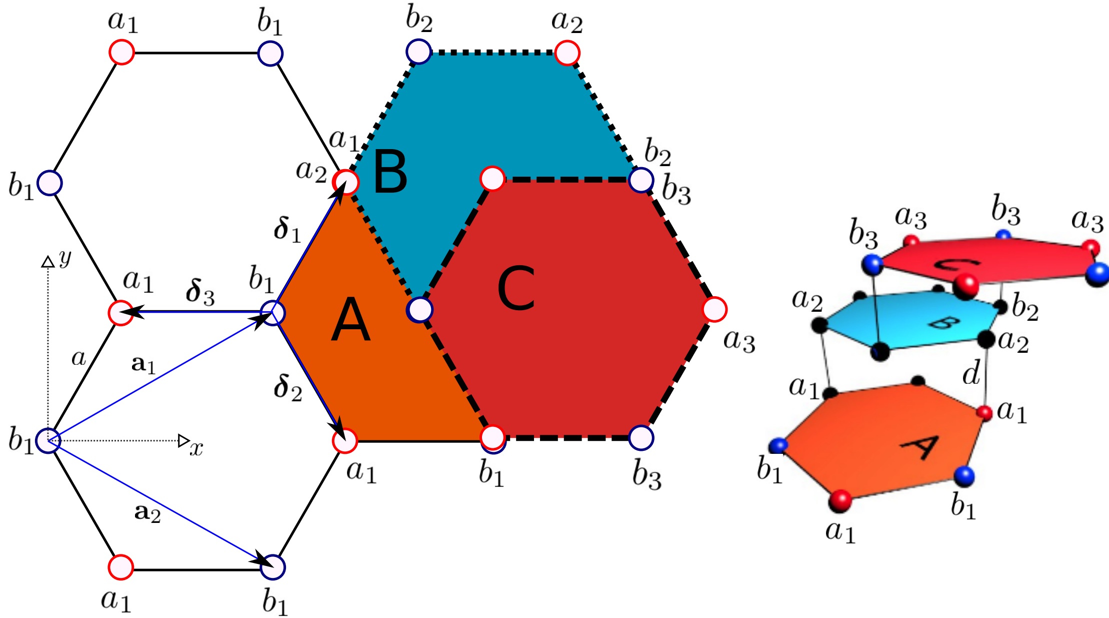


Figure 5.1: Structure of ABC-trilayer graphene.

and

$$\mathcal{H} = \begin{pmatrix} 0 & f(-\mathbf{k}) & -t_\perp & 0 & 0 & 0 \\ f(\mathbf{k}) & 0 & 0 & 0 & 0 & 0 \\ -t_\perp & 0 & 0 & f(\mathbf{k}) & 0 & 0 \\ 0 & 0 & f(-\mathbf{k}) & 0 & 0 & -t_\perp \\ 0 & 0 & 0 & 0 & 0 & f(-\mathbf{k}) \\ 0 & 0 & 0 & -t_\perp & f(\mathbf{k}) \end{pmatrix}.$$

The low energy approximation is found by expanding around the K-point $\mathbf{K}$ using equations (3.18), (3.19) and (3.20). This yields

$$\mathcal{H} = \begin{pmatrix} 0 & ke^{-i\phi(k)} & -t_\perp & 0 & 0 & 0 \\ ke^{i\phi(k)} & 0 & 0 & 0 & 0 & 0 \\ -t_\perp & 0 & 0 & ke^{i\phi(k)} & 0 & 0 \\ 0 & 0 & ke^{-i\phi(k)} & 0 & 0 & -t_\perp \\ 0 & 0 & 0 & 0 & 0 & ke^{-i\phi(k)} \\ 0 & 0 & 0 & -t_\perp & ke^{i\phi(k)} & \end{pmatrix},$$

where the units are such that $v_F = 1$.

5.1.2 ABC-trilayer dispersion

In order to calculate the full dispersion of ABC-trilayer graphene the full Hamiltonian $\mathcal{H}$ is numerically diagonalized using the Jacobi diagonalization algorithm for some range of $\mathbf{k}$. The resulting diagonal matrix then contains six distinct entries corresponding to the energy bands. These six bands are plotted in figure 5.2. From these figures, the dispersion looks quite similar to the bilayer dispersion except for the two extra bands. More interesting features appear when zooming in to the low energy regime. The same Jacobi diagonlization has been done for the low energy approximation of $\mathcal{H}$ (which is expanded around the K-point). A two dimensional intersection of the resulting dispersion is shown in figure 5.3. Making a least square fit to the lowest energy bands reveals that these bands are cubic. It is then interesting to note that the bilayer dispersion is quadratic, which means that it costs less energy to fill the cubic bands of ABC-trilayer graphene.

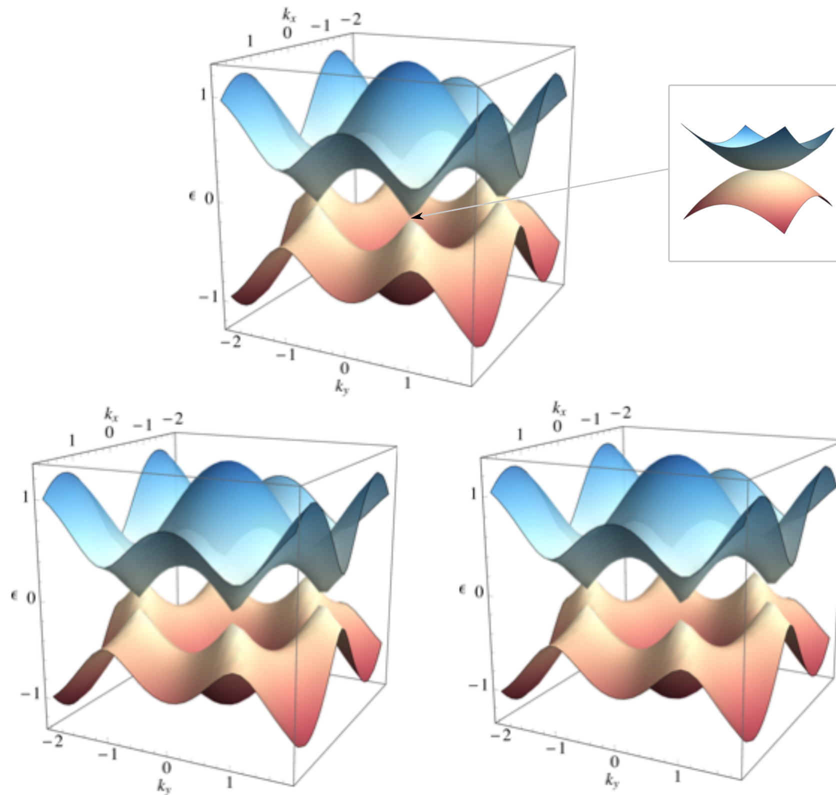


Figure 5.2: Full dispersion of ABC-trilayer graphene.

Since there is a competition between the kinetic energy and the exchange energy it is reasonable to expect that ABC-trilayer graphene therefore will have a stronger ferromagnetic behavior than bilayer graphene. In what follows, the correctness of this statement will be investigated in detail.

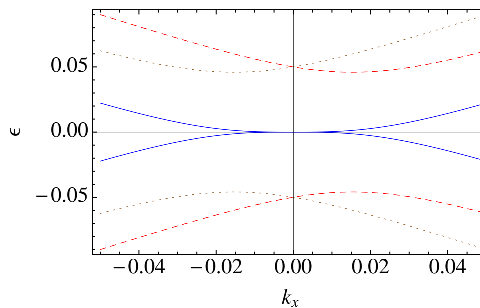


Figure 5.3: Low energy dispersion of ABC-trilayer graphene.

5.2 Ferromagnetism in ABC-trilayer graphene

5.2.1 Average kinetic energy

In the bilayer case the average kinetic energy could be calculated based on the analytical results of the dispersion relation. However, the ABC trilayer dispersion was not calculated analytically but numerically which means that the kinetic energy also must be calculated numerically. This is

achieved by calculating the inverse of the dispersion $k_\alpha(\epsilon)$ as shown in section 2.4 and using this to calculate

$$N_\alpha = \sum_{\mathbf{k}} \Theta(\epsilon - \epsilon_\alpha(\mathbf{k})) = \frac{A_c}{4\pi}(k_\alpha(\epsilon))^2,$$

where α is a band index (see section 4.2.2). The density of states $\mathcal{D}_\alpha$ could be calculated numerically by $\partial N_\alpha/\partial\epsilon$. However, this is not needed since $\mathcal{D}_\alpha$ is going to be used in the integration

$$\Delta E_{ex} = \int_0^{\epsilon_1(Q)} d\epsilon \epsilon \mathcal{D}_\alpha(\epsilon) = \int_0^{\epsilon_1(Q)} d\epsilon \epsilon \frac{\partial N_\alpha}{\partial \epsilon}.$$

A partial integration yields

$$\Delta E_{ex} = \epsilon_1(Q) N_\alpha(\epsilon_1(Q)) - \int_0^{\epsilon_1(Q)} d\epsilon N_\alpha(\epsilon), \tag{5.1}$$

which avoids any numerical differentiations. The double exponential algorithm described in section 2.1 is used to perform the required integration.

5.2.2 Exchange energy due to Coulomb interactions

Equation (4.11) gives the interaction Hamiltonian for bilayer graphene. Due to the added layer in ABC-trilayer graphene, this Hamiltonian becomes

$$\begin{aligned}
H_I = \frac{1}{2}\int d\mathbf{r}_1 \int d\mathbf{r}_2 \big[&\hat{\Psi}_1^\dagger(\mathbf{r}_1)\hat{\Psi}_1^\dagger(\mathbf{r}_2)V_{11}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_1)\\
&+ \hat{\Psi}_1^\dagger(\mathbf{r}_1)\hat{\Psi}_2^\dagger(\mathbf{r}_2)V_{12}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_1) + \hat{\Psi}_1^\dagger(\mathbf{r}_1)\hat{\Psi}_3^\dagger(\mathbf{r}_2)V_{13}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_3(\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_1)\\
&+ \hat{\Psi}_2^\dagger(\mathbf{r}_1)\hat{\Psi}_1^\dagger(\mathbf{r}_2)V_{21}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_1) + \hat{\Psi}_2^\dagger(\mathbf{r}_1)\hat{\Psi}_2^\dagger(\mathbf{r}_2)V_{22}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_1)\\
&+ \hat{\Psi}_2^\dagger(\mathbf{r}_1)\hat{\Psi}_3^\dagger(\mathbf{r}_2)V_{23}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_3(\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_1) + \hat{\Psi}_3^\dagger(\mathbf{r}_1)\hat{\Psi}_1^\dagger(\mathbf{r}_2)V_{31}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_1(\mathbf{r}_2)\hat{\Psi}_3(\mathbf{r}_1)\\
&+ \hat{\Psi}_3^\dagger(\mathbf{r}_1)\hat{\Psi}_2^\dagger(\mathbf{r}_2)V_{32}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_2(\mathbf{r}_2)\hat{\Psi}_3(\mathbf{r}_1) + \hat{\Psi}_3^\dagger(\mathbf{r}_1)\hat{\Psi}_3^\dagger(\mathbf{r}_2)V_{33}(\mathbf{r}_1-\mathbf{r}_2)\hat{\Psi}_3(\mathbf{r}_2)\hat{\Psi}_3(\mathbf{r}_1)\big].
\end{aligned} \tag{5.2}$$

As in the bilayer case, the interaction Hamiltonian can be written in terms of

$$\hat{\rho} := \hat{\Psi}^\dagger(\mathbf{r})\hat{\Psi}(\mathbf{r}).$$

This yields

$$\begin{aligned}
H_I = \frac{1}{2}\int d\mathbf{r}_1 \int d\mathbf{r}_2 \big[&V(\mathbf{r}_1-\mathbf{r}_2)\left\{\hat{\rho}_1(\mathbf{r}_1)\hat{\rho}_1(\mathbf{r}_2) + \hat{\rho}_2(\mathbf{r}_1)\hat{\rho}_2(\mathbf{r}_2) + \hat{\rho}_3(\mathbf{r}_1)\hat{\rho}_3(\mathbf{r}_2)\right\}\\
&+V_I(\mathbf{r}_1-\mathbf{r}_2)\left\{\hat{\rho}_1(\mathbf{r}_1)\hat{\rho}_2(\mathbf{r}_2) + \hat{\rho}_2(\mathbf{r}_1)\hat{\rho}_1(\mathbf{r}_2) + \hat{\rho}_2(\mathbf{r}_1)\hat{\rho}_3(\mathbf{r}_2) + \hat{\rho}_3(\mathbf{r}_1)\hat{\rho}_2(\mathbf{r}_2)\right\}\\
&+V_{II}(\mathbf{r}_1-\mathbf{r}_2)\left\{\hat{\rho}_1(\mathbf{r}_1)\hat{\rho}_3(\mathbf{r}_2) + \hat{\rho}_3(\mathbf{r}_1)\hat{\rho}_1(\mathbf{r}_2)\right\}\big],
\end{aligned}$$

where V and V_I are given by equations (4.12) and (4.13) and

$$V_{II}(\mathbf{r}_1-\mathbf{r}_2) = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|} = \frac{e^2}{\epsilon_0}\frac{1}{|\mathbf{r}+2d[0,0,1]|}\frac{e^2}{\epsilon_0}\frac{1}{\sqrt{r^2+(2d)^2}}$$

is the intra layer potential between layers 1 and 3. The Fourier transform is similar to V_I and becomes

$$V_{II}(\mathbf{k}) = \frac{2\pi e^2}{\epsilon_0}\frac{1}{k}e^{-2kd}.$$

The interaction Hamiltonian can now be Fourier transformed by using equation (4.15) which leads to

$$\begin{aligned}
H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\big[&\hat{\rho}_1(-\mathbf{q})V(\mathbf{q})\hat{\rho}_1(\mathbf{q}) + \hat{\rho}_2(-\mathbf{q})V(\mathbf{q})\hat{\rho}_2(\mathbf{q}) + \hat{\rho}_3(-\mathbf{q})V(\mathbf{q})\hat{\rho}_3(\mathbf{q})\\
&+\hat{\rho}_1(-\mathbf{q})V_I(\mathbf{q})\hat{\rho}_2(\mathbf{q}) + \hat{\rho}_2(-\mathbf{q})V_I(\mathbf{q})\hat{\rho}_1(\mathbf{q}) + \hat{\rho}_2(-\mathbf{q})V_I(\mathbf{q})\hat{\rho}_3(\mathbf{q}) + \hat{\rho}_3(-\mathbf{q})V_I(\mathbf{q})\hat{\rho}_2(\mathbf{q})\\
&+\hat{\rho}_1(-\mathbf{q})V_{II}(\mathbf{q})\hat{\rho}_3(\mathbf{q}) + \hat{\rho}_3(-\mathbf{q})V_{II}(\mathbf{q})\hat{\rho}_1(\mathbf{q})\big].
\end{aligned}$$

In matrix form this becomes

$$H_I = \frac{1}{2A_c} \sum_{\mathbf{q}\neq 0} \begin{pmatrix} \hat{\rho}_1(-\mathbf{q}) & \hat{\rho}_2(-\mathbf{q}) & \hat{\rho}_3(-\mathbf{q}) \end{pmatrix} \begin{pmatrix} V(\mathbf{q}) & V_I(\mathbf{q}) & V_{II}(\mathbf{q}) \\ V_I(\mathbf{q}) & V(\mathbf{q}) & V_I(\mathbf{q}) \\ V_{II}(\mathbf{q}) & V_I(\mathbf{q}) & V(\mathbf{q}) \end{pmatrix} \begin{pmatrix} \hat{\rho}_1(\mathbf{q}) \\ \hat{\rho}_2(\mathbf{q}) \\ \hat{\rho}_3(\mathbf{q}) \end{pmatrix}.$$

The matrix

$$M = \begin{pmatrix} V(\mathbf{q}) & V_I(\mathbf{q}) & V_{II}(\mathbf{q}) \\ V_I(\mathbf{q}) & V(\mathbf{q}) & V_I(\mathbf{q}) \\ V_{II}(\mathbf{q}) & V_I(\mathbf{q}) & V(\mathbf{q}) \end{pmatrix} \tag{5.3}$$

is complicated to diagonalize and leads to correspondingly complicated expressions for the eigenvalues and eigenvectors. This problem can be circumvented by splitting M into a tridiagonal and a residual matrix:

$$M := M_t + M_r = \begin{pmatrix} V(\mathbf{q}) & V_I(\mathbf{q}) & 0 \\ V_I(\mathbf{q}) & V(\mathbf{q}) & V_I(\mathbf{q}) \\ 0 & V_I(\mathbf{q}) & V(\mathbf{q}) \end{pmatrix} + \begin{pmatrix} 0 & 0 & V_{II}(\mathbf{q}) \\ 0 & 0 & 0 \\ V_{II}(\mathbf{q}) & 0 & 0 \end{pmatrix}$$

and instead diagonalizing each term separately. Notice that the matrix of the second term is augmented to a 3×3 matrix from the off diagonal 2×2 matrix

$$\begin{pmatrix} 0 & V_{II}(\mathbf{q}) \\ V_{II}(\mathbf{q}) & 0 \end{pmatrix}.$$

If the 2×2 matrix was used directly, this would eventually require the bilayer diagonalization matrix $\mathcal{M}$ in order to write the Hamiltonian in a band oriented basis. However, by augmenting to the 3×3 matrix will require only the ABC-trilayer diagonalization matrix to write the Hamiltonian in a band oriented basis. This will become clear in following calculations.

The tridiagonal matrix M_t is diagonalized by

$$U_t = \frac{1}{2} \begin{pmatrix} -\sqrt{2} & 0 & \sqrt{2} \\ 1 & -\sqrt{2} & 1 \\ 1 & \sqrt{2} & 1 \end{pmatrix}$$

with eigenvalues

$$D_t = \begin{pmatrix} V & 0 & 0 \\ 0 & V - \sqrt{2}V_I & 0 \\ 0 & 0 & V + \sqrt{2}V_I \end{pmatrix},$$

such that $U_t^T D_t U_t = M_t$. The residual matrix M_r is diagonalized by

$$U_r = \begin{pmatrix} 0 & 1 & 0 \\ -1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} \end{pmatrix}$$

with eigenvalues

$$D_r = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -V_{II} & 0 \\ 0 & 0 & V_{II} \end{pmatrix},$$

such that $U_r^T D_r U_r = M_r$.

The Hamiltonian can now be written as

$$\begin{aligned} H_I = &\frac{1}{2A_c} \sum_{\mathbf{q}\neq 0} \begin{pmatrix} \hat{\rho}_1(-\mathbf{q}) & \hat{\rho}_2(-\mathbf{q}) & \hat{\rho}_3(-\mathbf{q}) \end{pmatrix} U_t^\dagger U_t M_t U_t^\dagger U_t \begin{pmatrix} \hat{\rho}_1(\mathbf{q}) \\ \hat{\rho}_2(\mathbf{q}) \\ \hat{\rho}_3(\mathbf{q}) \end{pmatrix} \\ &+ \frac{1}{2A_c} \sum_{\mathbf{q}\neq 0} \begin{pmatrix} \hat{\rho}_1(-\mathbf{q}) & \hat{\rho}_2(-\mathbf{q}) & \hat{\rho}_3(-\mathbf{q}) \end{pmatrix} U_r^\dagger U_r M_r U_r^\dagger U_r \begin{pmatrix} \hat{\rho}_1(\mathbf{q}) \\ \hat{\rho}_2(\mathbf{q}) \\ \hat{\rho}_3(\mathbf{q}) \end{pmatrix}, \end{aligned}$$

By writing out the matrices one finds

$$H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\frac{2\pi e^2}{\epsilon_0 q}\frac{1}{\sqrt{2}}\begin{pmatrix}\tilde{\rho}_1^* & \tilde{\rho}_2^* & \tilde{\rho}_3^*\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & 1-\sqrt{2}e^{-qd} & 0\\ 0 & 0 & 1+\sqrt{2}e^{-qd}\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}\tilde{\rho}_1\\ \tilde{\rho}_2\\ \tilde{\rho}_3\end{pmatrix}$$
$$+\frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\frac{2\pi e^2}{\epsilon_0 q}\frac{1}{\sqrt{2}}\begin{pmatrix}\tilde{\rho}_4^* & \tilde{\rho}_5^* & \tilde{\rho}_6^*\end{pmatrix}\begin{pmatrix}0 & 0 & 0\\ 0 & -e^{-2qd} & 0\\ 0 & 0 & e^{-2qd}\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}\tilde{\rho}_4\\ \tilde{\rho}_5\\ \tilde{\rho}_6\end{pmatrix},$$

where $\tilde{\rho}_i^* := \tilde{\rho}_i(-\mathbf{q})$ and

$$\frac{1}{\sqrt{2}}\begin{pmatrix}\tilde{\rho}_1\\ \tilde{\rho}_2\\ \tilde{\rho}_3\end{pmatrix} := U_t\begin{pmatrix}\hat{\rho}_1\\ \hat{\rho}_2\\ \hat{\rho}_3\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}-\hat{\rho}_1+\hat{\rho}_3\\ \hat{\rho}_1/\sqrt{2}-\hat{\rho}_2+\hat{\rho}_3/\sqrt{2}\\ \hat{\rho}_1/\sqrt{2}+\hat{\rho}_2+\hat{\rho}_3/\sqrt{2}\end{pmatrix},$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix}\tilde{\rho}_4\\ \tilde{\rho}_5\\ \tilde{\rho}_6\end{pmatrix} := U_r\begin{pmatrix}\hat{\rho}_1\\ \hat{\rho}_2\\ \hat{\rho}_3\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}\sqrt{2}\hat{\rho}_2\\ -\hat{\rho}_1+\hat{\rho}_3\\ \hat{\rho}_1+\hat{\rho}_3\end{pmatrix}.$$

With the definitions

$$V_1(q) := \frac{2\pi e^2}{\epsilon_0 q}/2, \tag{5.4}$$
$$V_2(q) := \frac{2\pi e^2}{\epsilon_0 q}(1-\sqrt{2}e^{-qd})/2, \tag{5.5}$$
$$V_3(q) := \frac{2\pi e^2}{\epsilon_0 q}(1+\sqrt{2}e^{-qd})/2, \tag{5.6}$$
$$V_4(q) := 0, \tag{5.7}$$
$$V_5(q) := \frac{2\pi e^2}{\epsilon_0 q}(-e^{-2qd})/2, \tag{5.8}$$
$$V_6(q) := \frac{2\pi e^2}{\epsilon_0 q}(e^{-2qd})/2, \tag{5.9}$$

the interaction Hamiltonian becomes

$$H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\sum_{\alpha=1}^{6}\tilde{\rho}_\alpha(-\mathbf{q})V_\alpha(\mathbf{q})\tilde{\rho}_\alpha(\mathbf{q}),$$

in analogy with equation (4.16) for bilayer graphene.

Fourier transforming the three $\hat{\rho}_i$ operators leads to

$$\hat{\rho}_i(\mathbf{q}) = \int d\mathbf{r}\hat{\rho}_i(\mathbf{r})e^{-i\mathbf{r}\cdot\mathbf{q}} = \int d\mathbf{r}\hat{\Psi}_i^\dagger(\mathbf{r})\hat{\Psi}_i(\mathbf{r})e^{-i\mathbf{r}\cdot\mathbf{q}}.$$

Now, as was also done for bilayer graphene, the $\hat{\Psi}_i$ operators are Fourier transformed. This yields

$$\hat{\rho}_i(\mathbf{q}) = \frac{1}{A_c}\sum_{\mathbf{k}}\hat{\Psi}_i^\dagger(\mathbf{k}+\mathbf{q})\hat{\Psi}_i(\mathbf{k}) = \sum_{\mathbf{k}}\tilde{\Psi}_i^\dagger(\mathbf{k}+\mathbf{q})\tilde{\Psi}_i(\mathbf{k})$$

in the discrete limit. Utilizing this result, the six $\tilde{\rho}_\alpha$ operators can be written as

$$\tilde{\rho}_\alpha(\mathbf{q}) = \sum_{\mathbf{k}}\hat{\Psi}^\dagger(\mathbf{q})\tilde{\chi}^\alpha\hat{\Psi}(\mathbf{q}) = \sum_{\mathbf{k}}\hat{\Phi}^\dagger(\mathbf{q})\mathcal{M}^\dagger(\mathbf{k}+\mathbf{q})\tilde{\chi}^\alpha\mathcal{M}(\mathbf{k})\hat{\Phi}(\mathbf{q}),$$

where

$$\hat{\Psi}^\dagger(\mathbf{q}) := \left(\tilde{\Psi}_1^\dagger(\mathbf{k}+\mathbf{q}) \quad \tilde{\Psi}_2^\dagger(\mathbf{k}+\mathbf{q}) \quad \tilde{\Psi}_3^\dagger(\mathbf{k}+\mathbf{q})\right),$$
$$\hat{\Phi}^\dagger(\mathbf{q}) := \left(\hat{\Phi}_1^\dagger(\mathbf{k}+\mathbf{q}) \quad \hat{\Phi}_2^\dagger(\mathbf{k}+\mathbf{q}) \quad \hat{\Phi}_3^\dagger(\mathbf{k}+\mathbf{q})\right),$$

and $\mathcal{M}$ is the diagonalizing matrix for the ABC-trilayer Hamiltonian. Since $\{\hat{\Phi}_i\}$ are band annihilation operators, the operators $\{\tilde{\rho}_\alpha\}$ are written in a band oriented manner. For example

$$\tilde{\rho}_2 = \frac{1}{\sqrt{2}}\hat{\rho}_1 - \hat{\rho}_2 + \frac{1}{\sqrt{2}}\hat{\rho}_3 = \sum_{\mathbf{k}} \hat{\Phi}^\dagger(\mathbf{q})\mathcal{M}^\dagger(\mathbf{k}+\mathbf{q}) \begin{pmatrix} \mathbf{1}_2/\sqrt{2} & 0 & 0 \\ 0 & -\mathbf{1}_2 & 0 \\ 0 & 0 & \mathbf{1}_2/\sqrt{2} \end{pmatrix} \mathcal{M}(\mathbf{k})\hat{\Phi}(\mathbf{q}).$$

The above definitions leads to the following matrices:

$$\tilde{\chi}^1 := \begin{pmatrix} -\mathbf{1}_2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{1}_2 \end{pmatrix},$$
$$\tilde{\chi}^2 := \begin{pmatrix} \mathbf{1}_2/\sqrt{2} & 0 & 0 \\ 0 & -\mathbf{1}_2 & 0 \\ 0 & 0 & \mathbf{1}_2/\sqrt{2} \end{pmatrix},$$
$$\tilde{\chi}^3 := \begin{pmatrix} \mathbf{1}_2/\sqrt{2} & 0 & 0 \\ 0 & \mathbf{1}_2 & 0 \\ 0 & 0 & \mathbf{1}_2/\sqrt{2} \end{pmatrix},$$
$$\tilde{\chi}^4 := \begin{pmatrix} 0 & 0 & 0 \\ 0 & \sqrt{2}\mathbf{1}_2 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$
$$\tilde{\chi}^5 := \begin{pmatrix} -\mathbf{1}_2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{1}_2 \end{pmatrix},$$
$$\tilde{\chi}^6 := \begin{pmatrix} \mathbf{1}_2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{1}_2 \end{pmatrix}.$$

Finally, by defining

$$\chi^\alpha = \mathcal{M}^\dagger(\mathbf{k}+\mathbf{q})\tilde{\chi}^\alpha\mathcal{M}(\mathbf{k}),$$

the interaction Hamiltonian can be written in the form

$$H_I = \frac{1}{2A_c}\sum_{\mathbf{q}\neq 0}\sum_{\mathbf{p},\mathbf{p}'}\sum_{\alpha=1}^{6}\sum_{i,j,m,n=1}^{6} \hat{\Phi}_i^\dagger(\mathbf{p}-\mathbf{q})\chi_{ij}^\alpha(\mathbf{p}-\mathbf{q},\mathbf{p})\hat{\Phi}_j(\mathbf{p})V_\alpha(\mathbf{q}) \times\hat{\Phi}_m^\dagger(\mathbf{p}'+\mathbf{q})\chi_{mn}^\alpha(\mathbf{p}'+\mathbf{q},\mathbf{p}')\hat{\Phi}_n(\mathbf{p}').$$

Notice the resemblance to equation (4.17) for bilayer graphene. It is then clear from equation (4.20) that the interaction Hamiltonian for ABC-trilayer graphene becomes

$$\frac{E_{ex}}{A_c} = -\frac{\Lambda^4}{2}\frac{1}{(2\pi)^3}\int_0^1 dkk\int_0^1 dk'k'\int_0^{2\pi} d\theta\sum_{\alpha=1}^{6}\sum_{i,j=1}^{6}\sum_{\sigma,a}\chi_{ij}^\alpha(k\Lambda,k'\Lambda,\theta)\chi_{ji}^\alpha(k'\Lambda,k\Lambda,\theta) \times n_{\sigma,a,i}(k\Lambda)n_{\sigma,a,j}(k'\Lambda)V_\alpha(|\mathbf{k}'-\mathbf{k}|). \quad (5.10)$$

5.2.3 The kinetic and exchange energy differences

The ferromagnetic and paramagnetic states that will be compared is similar to the ones that were compared for bilayer graphene since all the high energy hole bands are filled and all the high

energy particle bands are empty. That is, for ABC-trilayer graphene bands 3 and 5 are empty and bands 4 and 6 are filled. The only difference is that the low energy bands of ABC-trilayer graphene are cubic instead of quadratic and that the high energy bands differs slightly from the high energy bands of bilayer graphene. Therefore, figures 4.6, 4.7 and 4.8 illustrates the possible transitions of the system: undoped, doped with one type of charge carrier and doped with two types of charge carrier.

The kinetic energy difference between a ferromagnetic and a paramagnetic state is calculated in the same manner as for bilayer graphene using equation (4.25) together with the numerical results of equation (5.1).

The calculations of the exchange energy difference is also quite similar to the calculations done for bilayer graphene, resulting in equations (4.23) and (4.24). The only difference is the additional band that is filled, i.e. band 6. The filling of band 6 enters into the expression in the same manner as band 4. In principle it is necessary to include the effects of interactions between bands 4 and 6. However, due to these bands being filled in both the paramagnetic and ferromagnetic phase, the interaction effects between bands 4 and 6 cancel between the two phases. Thus, for two types of charge carriers

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = \int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \big\{ & 4Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\uparrow F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) \\
& - 8Q_d F_{11}(Q_d k, Q_d kk', \theta) - 8Q_d F_{12}(Q_d k, Q_d kk', \theta) \\
& + 4Q_\uparrow F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) - 4Q_\downarrow F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
& + 4Q_\uparrow F_{16}(Q_\uparrow k, Q_\uparrow kk', \theta) - 4Q_\downarrow F_{26}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
& - 8Q_d F_{14}(Q_d k, Q_d kk', \theta) - 8Q_d F_{16}(Q_d k, Q_d kk', \theta) \big\} V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} & dk' 4Q_\uparrow \left\{ F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) + F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) \right\} V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} & dk' 4Q_\uparrow F_{16}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} & dk' 4Q_\downarrow \left\{ F_{22}(Q_\downarrow k, Q_\downarrow kk', \theta) + F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) \right\} V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} & dk' 4Q_\downarrow F_{26}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} & dk' 8Q_d \left\{ F_{12}(Q_d k, Q_d kk', \theta) + F_{14}(Q_d k, Q_d kk', \theta) \right\} V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} & dk' 8Q_d F_{16}(Q_d k, Q_d kk', \theta) V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\uparrow} & dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) + F_{16}(kk', k, \theta) \right\} V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\downarrow} & dk' 4 \left\{ F_{22}(kk', k, \theta) + F_{24}(kk', k, \theta) + F_{26}(kk', k, \theta) \right\} V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_d} & dk' 8 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) + F_{16}(kk', k, \theta) \right\} V_s(k', \theta)
\end{aligned}
\tag{5.11}
$$

and for one type of charge carrier

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = &\int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \Big\{ 4Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\uparrow F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) \\
&+ 4Q_\downarrow F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) + 4Q_\downarrow F_{11}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
&- 8Q_d F_{11}(Q_d k, Q_d kk', \theta) - 8Q_d F_{12}(Q_d k, Q_d kk', \theta) \\
&+ 4Q_\uparrow F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\downarrow F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
&+ 4Q_\uparrow F_{16}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\downarrow F_{16}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
&- 8Q_d F_{14}(Q_d k, Q_d kk', \theta) - 8Q_d F_{16}(Q_d k, Q_d kk', \theta) \Big\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} dk' 4Q_\uparrow \{F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) + F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta)\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} dk' 4Q_\uparrow F_{16}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} dk' 4Q_\downarrow \{F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) + F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta)\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} dk' 4Q_\downarrow F_{16}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} dk' 8Q_d \{F_{12}(Q_d k, Q_d kk', \theta) + F_{14}(Q_d k, Q_d kk', \theta)\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} dk' 8Q_d F_{16}(Q_d k, Q_d kk', \theta) V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\uparrow} dk' 4 \{F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) + F_{16}(kk', k, \theta)\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\downarrow} dk' 4 \{F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) + F_{16}(kk', k, \theta)\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_d} dk' 8 \{F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) + F_{16}(kk', k, \theta)\} V_s(k', \theta). \quad (5.12)
\end{aligned}
$$

5.2.4 ABC-trilayer phase diagram - no screening

Equations (5.11) and (5.12) are calculated numerically using the double exponential algorithm for integration while the necessary matrix diagonalizations needed for F_{ij} is calculated using the Jacobi algorithm. Adding $\Delta E_k/A_c$ and $\Delta E_{ex}/A_c$ gives the total difference in energy between a paramagnetic and ferromagnetic phase. As was the case for bilayer graphene, curves of ΔE versus x' are plotted in order to find the critical doping Q_d. Figure 5.4 shows a plot of ΔE versus $-x'$ for several values of the electron-electron coupling g, including the curve at critical doping. Figures 5.5 and 5.6 illustratetes how one point in a $n = Q_d^2/2$ versus g phase diagram is found. The resulting phase diagram for ABC-trilayer graphene is shown in figure 5.7. This result does not take into account the effects of screening.

5.3 Effects of screening

In this chapter the ferromagnetic behavior of ABC-trilayer graphene was calculated by using unscreened Coulomb potentials for both interlayer as well as interlayer interactions. For a three dimensional gas and also for a solid state system described by the Jellium model it is known that the corrections due to screening is contained in the Thomas-Fermi screening vector k_{tf} [57]. This

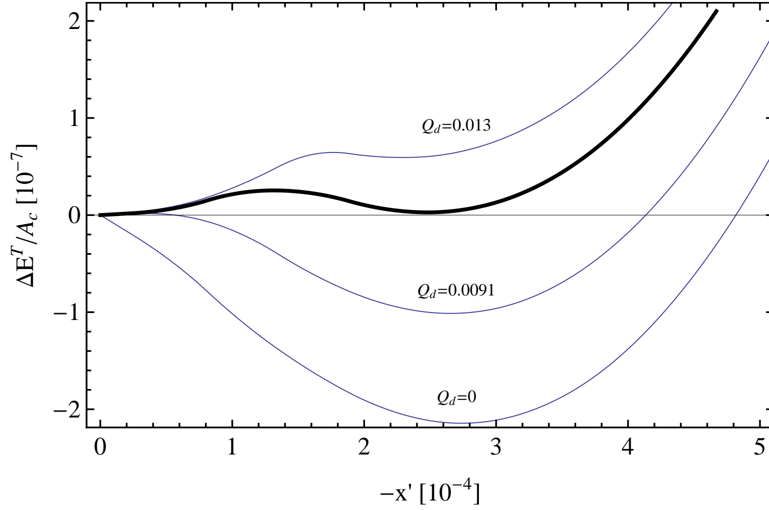


Figure 5.4: $\Delta E^T/A_c$ versus $-x'$ for $g=6$.

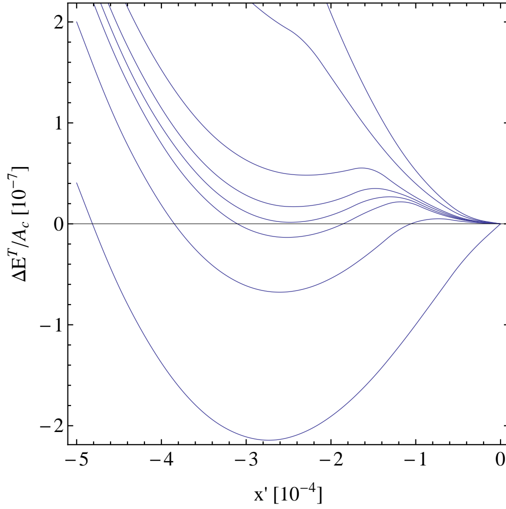


Figure 5.5: $\Delta E^T/A_c$ versus x for $g=6$.

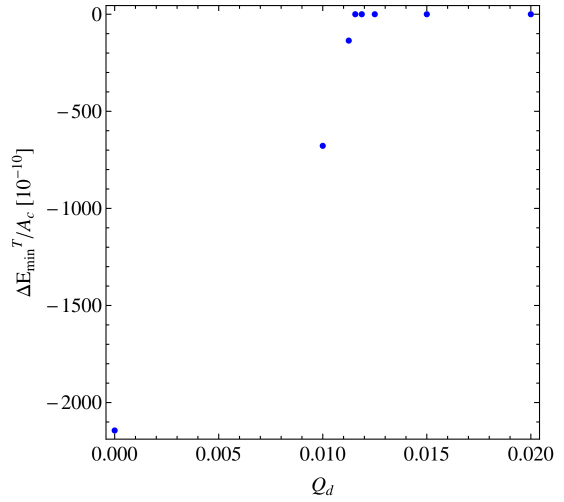


Figure 5.6: $\Delta E^T_{min}/A_c$ versus Q_d for $g=6$.

can be described by

$$V \sim \frac{1}{k} \to \frac{1}{k+k_{tf}}.$$

It turns out that for a 3D gas k_{tf} is proportional to the density of states $\mathcal{D}(\epsilon_F)$. Since the density of states for bilayer graphene is constant the corresponding effects of screening will be negligible. However, it was shown that the density of states for ABC-trilayer graphene grew rapidly as $\epsilon \to 0$. Therefore, in the low energy regime (i.e. for small Fermi energies) the screening effects in ABC-trilayer graphene can be expected to be considerable.

There exists a more modern view of this phenomenon. As already mentioned, the ferromagnetic effects calculated in this chapter included two contributions. The first contribution came from the kinetic energy, derived from the single particle dispersion relation. The second contribution came from the Hartree and Fock contributions, which correspond to first order corrections in perturbation theory. For a 3D gas, it turns out that adding only terms that have a maximum divergence factor (up to infinite order) leads to the same renormalization of the potential as that

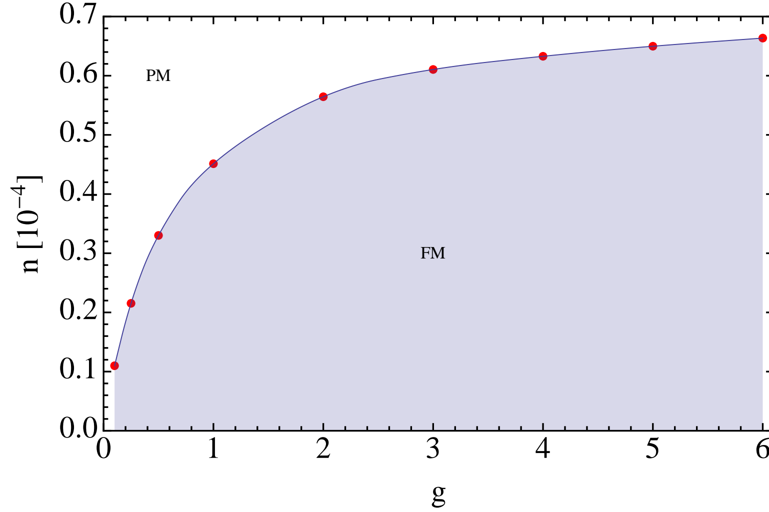


Figure 5.7: Phase diagram for ABC-trilayer graphene: doping n versus electron-electron coupling g, without screening effects.

of Thomas-Fermi screening. This is referred to as random phase approximation (RPA) [58]. In the following sections the RPA calculations will be repeated in detail for ABC-trilayer graphene, as is done for a 3D gas in references [56] and [58].

5.3.1 The ABC-trilayer action in the language of path integrals

The interaction Hamiltonian in equation (5.2) can be written in the simple form

$$H_I = \sum_{ij} \frac{1}{2} \int d\mathbf{r}_1 \int d\mathbf{r}_2 \hat{\Psi}_i^\dagger(\mathbf{r}_1)\hat{\Psi}_j^\dagger(\mathbf{r}_2)V_{ij}(\mathbf{r}_1 - \mathbf{r}_2)\hat{\Psi}_j(\mathbf{r}_2)\hat{\Psi}_i(\mathbf{r}_1),$$

where i, j run over all possible graphene layers. So far, the non-interacting Hamiltonian has been written in the form

$$H_0 = \sum_{\mathbf{k},\sigma,\alpha,a} \epsilon_{\mathbf{k},\sigma,\alpha,a} \hat{a}^\dagger_{\mathbf{k},\sigma,\alpha,a} \hat{a}_{\mathbf{k},\sigma,\alpha,a},$$

where α runs over the six energy bands and a runs over the two K-points of the Brillouin zone. In order to write H_0 in terms of field operators the ABC-trilayer version of equation (4.10) is dotted with $\Phi^*_{\mathbf{k},\sigma,\alpha,a}(\mathbf{r}')$ and integrated with respect to $\mathbf{r}'$. Using the completeness of the wave function $\Phi_{\mathbf{k},\sigma,\alpha,a}(\mathbf{r})$ (with normalization constant A_c) leads to

$$\hat{a}_{\mathbf{k},\sigma,\alpha,a} = \frac{1}{\sqrt{A_c}} \int d\mathbf{r} \hat{\Psi}(\mathbf{r}) \Phi^*_{\mathbf{k},\sigma,\alpha,a}(\mathbf{r}). \tag{5.13}$$

For a 3D gas the dispersion is of the form $\mathbf{p}^2/2m$ and the potential is translationally invariant such that

$$\epsilon_{\mathbf{k},\sigma} \Phi_{\mathbf{k},\sigma}(\mathbf{r}) = \left\{ -\frac{\nabla^2}{2m} + V^{ex}(\mathbf{r}) \right\} \Phi_{\mathbf{k},\sigma}(\mathbf{r}).$$

The dispersion $\epsilon_{\mathbf{k},\sigma,\alpha,a}$ of ABC-trilayer graphene is not quadratic and the real space potential is dependent on the layer indices. Therefore, this Schroedinger type expression cannot be used directly. Thus,

$$H_0 \neq \sum_i \int d\mathbf{r} \hat{\Psi}_i^\dagger(\mathbf{r}) \left\{ -\frac{\nabla^2}{2m} + V^{ex}(\mathbf{r}) \right\} \hat{\Psi}_i(\mathbf{r}),$$

which is the expression that would be expected for a 3D gas. It will turn out that an explicit expression is required for H_0 in order to find the non-interacting Green's function, and it can no longer be assumed that it is diagonal in any of the variables or indices (such as the layer indices). This also makes sense on physical grounds. If the Greens function was diagonal, this would mean that an electron could not propagate between layers. But, the Hamiltonian that is used includes interlaying hopping. Thus, the propagator must support electron hopping between layers and cannot be diagonal in the layer indices. This section will not cover the actual calculation of the non-interacting Green's function in detail, but will only outline how the screening can be included in ABC-trilayer graphene.

The state

$$|\phi\rangle := \exp\left\{-\sum_{\mathbf{k},\sigma,\alpha,a} \phi_{\mathbf{k},\sigma,\alpha,a}\hat{a}^\dagger_{\mathbf{k},\sigma,\alpha,a}\right\}|0\rangle$$

is an eigenstate of the annihilation operator and therefore a coherent state. By inserting equation (5.13) into this equation one finds

$$|\phi\rangle = \exp\left\{-\sum_i \int d\mathbf{r}\phi_i(\mathbf{r})\hat{\Psi}_i^\dagger(\mathbf{r})\right\}|0\rangle,$$

where

$$\phi_i(\mathbf{r}) := \frac{1}{\sqrt{A_c}}\sum_{\mathbf{k},\sigma,\alpha,a} \Phi_{\mathbf{k},\sigma,\alpha,a,i}(\mathbf{r})\phi_{\mathbf{k},\sigma,\alpha,a}$$

is a Grassman valued quantum field. Having established H_0 and H_I in terms of second quantized fields operators and established the quantum field $\phi_i(\mathbf{r})$, the partition function for ABC-trilayer graphene can be written as [56]

$$Z = \sum_{\mathbf{N}}\langle\mathbf{N}|e^{-\beta(\hat{H}-\mu\hat{N})}|\mathbf{N}\rangle = \int d[\phi^*]d[\phi]e^{-S[\phi^*,\phi]},$$

where

$$S[\phi^*,\phi] = \int_0^\beta d\tau\left\{\sum_i\int d\mathbf{r}\phi_i^*(\mathbf{r},\tau)\frac{\partial}{\partial\tau}\phi(\mathbf{r},\tau) + H[\phi^*(\tau),\phi(\tau)]\right\},$$

$$H_I[\phi^*,\phi] = \frac{1}{2}\sum_{ij}\int d\mathbf{r}_1\int d\mathbf{r}_2\phi_i^*(\mathbf{r}_1,\tau)\phi_j^*(\mathbf{r}_2,\tau)V_{ij}(\mathbf{r}_1-\mathbf{r}_2)\phi_j(\mathbf{r}_2,\tau)\phi_i(\mathbf{r}_1,\tau)$$

and $H[\phi^*,\phi] = H_0[\phi^*,\phi] + H_I[\phi^*,\phi]$.

5.3.2 Momentum space Feynman rules

The interacting Greens function is defined as

$$\begin{aligned}G_{ij}(\mathbf{r}_1,\tau_1;\mathbf{r}_2,\tau_2) :&= -\langle\phi_i(\mathbf{r}_1,\tau_1)\phi_j^*(\mathbf{r}_2,\tau_2)\rangle\\ &= -\frac{1}{Z}\int d[\phi^*]d[\phi]\phi_i(\mathbf{r}_1,\tau_1)\phi_j^*(\mathbf{r}_2,\tau_2)e^{-S_0[\phi^*,\phi]-S_I[\phi^*,\phi]},\end{aligned}$$

where

$$S_0[\phi^*,\phi] = \int_0^\beta d\tau\left\{\sum_i\int d\mathbf{r}\phi_i^*(\mathbf{r},\tau)\frac{\partial}{\partial\tau}\phi(\mathbf{r},\tau) + H_0[\phi^*(\tau),\phi(\tau)]\right\}$$

and

$$S_I[\phi^*,\phi] = \frac{1}{2}\sum_{ij}\int_0^\beta d\tau\int d\mathbf{r}_1\int d\mathbf{r}_2\phi_i^*(\mathbf{r}_1,\tau)\phi_j^*(\mathbf{r}_2,\tau)V_{ij}(\mathbf{r}_1-\mathbf{r}_2)\phi_j(\mathbf{r}_2,\tau)\phi_i(\mathbf{r}_1,\tau).$$

Expanding the numerator and denominator to first order in S_I gives

$$G_{ij}(\mathbf{r}_1, \tau_1; \mathbf{r}_2, \tau_2) = \frac{-\langle \phi_i(\mathbf{r}_1, \tau_1)\phi_j^*(\mathbf{r_2}, \tau_2)\rangle_0 + \langle \phi_i(\mathbf{r}_1, \tau_1)\phi_j^*(\mathbf{r_2}, \tau_2)S_I[\phi^*, \phi]\rangle_0}{1 - \langle S_I[\phi^*, \phi]\rangle_0},$$

where the subscript "0" indicates a non-interacting expectation value. To first order

$$\frac{1}{1 - \langle S_I[\phi^*, \phi]\rangle_0} = 1 + \langle S_I[\phi^*, \phi]\rangle_0.$$

Furthermore, Wicks theorem gives

$$\begin{aligned}\langle S_I[\phi^*, \phi]\rangle_0 &= \frac{1}{2}\sum_{ij}\int_0^\beta d\tau \int d\mathbf{r}_1 \int d\mathbf{r}_2 \langle \phi_i^*(\mathbf{r}_1, \tau)\phi_j^*(\mathbf{r}_2, \tau)V_{ij}(\mathbf{r}_1 - \mathbf{r}_2)\phi_j(\mathbf{r}_2, \tau)\phi_i(\mathbf{r}_1, \tau)\rangle_0 \\ &= \frac{1}{2}\sum_{ij}\int_0^\beta d\tau \int d\mathbf{r}_1 \int d\mathbf{r}_2 V_{ij}(\mathbf{r}_1 - \mathbf{r}_2)G_0^{ii}(\mathbf{r}_1, \tau; \mathbf{r}_1, \tau^+)G_0^{jj}(\mathbf{r}_2, \tau; \mathbf{r}_2, \tau^+) \\ &- \frac{1}{2}\sum_{ij}\int_0^\beta d\tau \int d\mathbf{r}_1 \int d\mathbf{r}_2 V_{ij}(\mathbf{r}_1 - \mathbf{r}_2)G_0^{ji}(\mathbf{r}_2, \tau; \mathbf{r}_1, \tau^+)G_0^{ij}(\mathbf{r}_1, \tau; \mathbf{r}_2, \tau^+).\end{aligned}$$

These two terms can analogously be represented by the real space Feynman diagrams. Thus,

$$1 + \langle S_I[\phi^*, \phi]\rangle_0 = 1 - \frac{1}{2}\,[\text{diagram}] + \frac{1}{2}\,[\text{diagram}] + \mathcal{O}(S_I^2),$$

where the fermions as well as the potentials carry layer indices i and j. This differs slightly from the 3D gas, where each fermion carries a spin index due to Zeeman splitting of the energy levels. Similarly, one finds

$$\langle \phi_i(\mathbf{r}_1, \tau_1)\phi_j^*(\mathbf{r_2}, \tau_2)S_I[\phi^*, \phi]\rangle_0 = \frac{1}{2}\,[\text{diagram}] \times [\text{diagram}] - \frac{1}{2}\,[\text{diagram}] \times [\text{diagram}] - [\text{diagram}] + [\text{diagram}] + \mathcal{O}(S_I^2),$$

and of course

$$-\langle \phi_i(\mathbf{r}_1, \tau_1)\phi_j^*(\mathbf{r_2}, \tau_2)\rangle_0 = + [\text{diagram}] + \mathcal{O}(S_I^2).$$

Then, by keeping only the first order terms

$$[\text{diagram}] := G^{ij}(\mathbf{r}_1, \tau_1; \mathbf{r}_2, \tau_2) = [\text{diagram}] - [\text{diagram}] + [\text{diagram}], \tag{5.14}$$

which are the Hartree and Fock contributions to the interacting Greens function, respectively.

The Feynman rules only differ from the case of a 3D gas by the potential and the fermions carrying layer indices, where indices of internal vertices are summed over. This can be viewed equivalently as each vertex having a layer index. Thus,

$$\begin{aligned}G^{ij}(\mathbf{r}_1, \tau_1; \mathbf{r}_2, \tau_2) &= G_0^{ij}(\mathbf{r}_1, \tau_1; \mathbf{r}_2, \tau_2) + \sum_{kl}\int_0^\beta d\tau \int d\mathbf{r}_3 \int d\mathbf{r}_4 \\ &\times G_0^{ik}(\mathbf{r}_1, \tau_1; \mathbf{r}_3, \tau)G_0^{kl}(\mathbf{r}_3, \tau; \mathbf{r}_4, \tau^+)G_0^{lj}(\mathbf{r}_4, \tau; \mathbf{r}_2, \tau_2)[-V_{kl}(\mathbf{r}_3 - \mathbf{r}_4)],\end{aligned}$$

where the Hartree contribution was neglected due to the positive Jellium background. Fourier transforming this expression yields

$$
\begin{aligned}
G^{ij}(\mathbf{r}_1,\tau_1;\mathbf{r}_2,\tau_2) = \frac{1}{\beta}\int d\omega_1 \int \frac{d\mathbf{k_1}}{(2\pi)^2} G_0^{ij}(\mathbf{k}_1, i\omega_1) e^{i\mathbf{k}_1\cdot(\mathbf{r}_1-\mathbf{r}_2)} e^{i\omega_1(\tau_1-\tau_2)} \\
+ \sum_{kl} \int \frac{d\mathbf{k_1}}{(2\pi)^2}\frac{d\mathbf{k_2}}{(2\pi)^2}\frac{d\mathbf{k_3}}{(2\pi)^2}\frac{d\mathbf{k_4}}{(2\pi)^2} \int \frac{d\omega_1}{\beta}\frac{d\omega_2}{\beta}\frac{d\omega_3}{\beta} \\
\times G_0^{ik}(\mathbf{k}_1, i\omega_1) G_0^{kl}(\mathbf{k}_2, i\omega_2) G_0^{lj}(\mathbf{k}_3, i\omega_3)[-V_{kl}(\mathbf{k}_4)] \\
\times (2\pi)^2\delta^{(2)}(\mathbf{k}_2 - \mathbf{k}_1 + \mathbf{k}_4)(2\pi)^2\delta^{(2)}(\mathbf{k}_3 - \mathbf{k}_2 - \mathbf{k}_4)\beta\delta(\omega_3 - \omega_1) \\
\times e^{i\mathbf{k}_1\cdot(\mathbf{r}_1-\mathbf{r}_2)} e^{i\omega_1(\tau_1-\tau_2)},
\end{aligned}
$$

where the equalities $\mathbf{k}_3 = \mathbf{k}_1$, $\omega_3 = \omega_1$, coming from the delta functions, were used to rewrite the exponentials of the second term. Fourier transforming the left hand side of the expression gives

$$
\begin{aligned}
G^{ij}(\mathbf{k}_1, i\omega_1) = G_0^{ij}(\mathbf{k}_1, i\omega_1) + \sum_{kl} \int \frac{d\mathbf{k_2}}{(2\pi)^2}\frac{d\mathbf{k_3}}{(2\pi)^2}\frac{d\mathbf{k_4}}{(2\pi)^2} \int \frac{d\omega_2}{\beta} \\
\times G_0^{ik}(\mathbf{k}_1, i\omega_1) G_0^{kl}(\mathbf{k}_2, i\omega_2) G_0^{lj}(\mathbf{k}_3, i\omega_1)[-V_{kl}(\mathbf{k}_4)] \\
\times (2\pi)^2\delta^{(2)}(\mathbf{k}_2 - \mathbf{k}_1 + \mathbf{k}_4)(2\pi)^2\delta^{(2)}(\mathbf{k}_3 - \mathbf{k}_2 - \mathbf{k}_4).
\end{aligned}
\tag{5.15}
$$

From the above expression it is fairly straight forward to deduce the Feynman rules in Fourier space for ABC-trilayer graphene, where only the connected diagrams survive. These are shown in table 5.1.

Diagram	*Descritption*	*Association*
$\mathbf{k}_1,\omega_1$ $\mathbf{p}$ $\mathbf{k}_2,\omega_2$	Internal vertex	$(2\pi)^2\delta^{(2)}(\mathbf{k}_2 - \mathbf{k}_1 - \mathbf{p})$
$\mathbf{p}$ i j	Potential line	$-V_{ij}(\mathbf{k})$
i j	Internal layer indices	$\sum_{ij}$
$\mathbf{k},\omega$ i j	Fermion line	$G_0^{ij}(\mathbf{k}, i\omega)$
$\mathbf{k}_1,\omega_1$ $\mathbf{k}_2,\omega_2$	External fermion lines	$\beta\delta(\omega_2 - \omega_1)$
$\mathbf{k}$	Free momenta	$\int \frac{d\mathbf{k}}{(2\pi)^2}$
$\mathbf{k},\omega$	Free momenta	$\int \frac{d\omega}{\beta} \int \frac{d\mathbf{k}}{(2\pi)^2}$
F	Number of fermion loops F	$(-1)^F$

Table 5.1: Momentum space Feynman rules for ABC-trilayer graphene.

Each piece of the Feynman diagram can be thought of as a matrix. It is therefore possible to write equation (5.14) in the form

where Σ consists of all irreducible and amputated diagrams (i.e. it is the self energy). This is the Dyson equation. Solving this equation leads to a renormalization of the single particle dispersion relation by adding the self energy Σ. The renormalized dispersion is referred to as the quasiparticle dispersion relation since it rather represents the 'single particle' dispersion of a collection of particles. This is an alternative way to look at the effects of additional corrections to the system without interactions. If the self energy only contains the Fock term, then the previously calculated exchange energy can be re-derived using this technique.

5.3.3 Random phase approximation

By treating each part of the Feynman diagram as matrices, considering a positive Jellium background and only keeping diagrams with the highest divergence factor one finds that

if only diagrams with the highest divergence factors are kept. This is referred to as a random phase approximation (RPA). Thus, the self energy Σ can be written as the Fock self energy with a renormalized potential $\mathcal{V}_{ij}$. It turns out that the renormalized potential in RPA for a 3D gas exactly corresponds to a potential which includes Thomas-Fermi screening. The renormalized potential can be written as

(5.16)

This is a Dyson-like equation that needs to be solved with respect to $\mathcal{V}_{ij}$. Using the Feynman rules of table 5.1 and completing the integration of the delta functions the second term becomes

$$= -\sum_{kl} V_{ik}(\mathbf{k}) \int \frac{d\mathbf{k}'}{(2\pi)^2} \int \frac{d\omega'}{\beta} G_0^{kl}(\mathbf{k}', i\omega') G_0^{lk}(\mathbf{k}' - \mathbf{k}, i\omega' - i\omega) \mathcal{V}_{lj}(\mathbf{k}).$$

Let

$$\Gamma_{il}(\mathbf{k}) := \sum_k V_{ik}(\mathbf{k}) \int \frac{d\mathbf{k}'}{(2\pi)^2} \int \frac{d\omega'}{\beta} G_0^{kl}(\mathbf{k}', i\omega') G_0^{lk}(\mathbf{k}' - \mathbf{k}, i\omega' - \omega).$$

Then equation (5.16) can be written as the matrix equation

$$\mathcal{V}_{ij}(\mathbf{k}) = V_{ij}(\mathbf{k}) + \sum_l \Gamma_{il}(\mathbf{k}) \mathcal{V}_{lj}(\mathbf{k}),$$

which can be written as

$$\sum_l \{\delta_{il} - \Gamma_{il}\} \mathcal{V}_{lj} = V_{ij},$$

where the momentum variables are omitted. This consists of three equations in three unknowns for $i = 1$, $i = 2$ and $i = 3$, respectively. In matrix form, this becomes

$$\begin{pmatrix} 1-\Gamma_{11} & \Gamma_{12} & \Gamma_{13} \\ \Gamma_{21} & 1-\Gamma_{22} & \Gamma_{23} \\ \Gamma_{31} & \Gamma_{32} & 1-\Gamma_{33} \end{pmatrix} \begin{pmatrix} \mathcal{V}_{j1} \\ \mathcal{V}_{j2} \\ \mathcal{V}_{j3} \end{pmatrix} = \begin{pmatrix} V_{1j} \\ V_{2j} \\ V_{3j} \end{pmatrix}, \tag{5.17}$$

At this point the potential V_{ij}, which is in the layer representation, has been renormalized to the potential $\mathcal{V}_{ij}$, which is also in the layer representation. However, when calculating the ferromagnetic properties of graphene the potentials V_1 to V_6 in the band representation were used. These potentials are listed in equations (5.4) to (5.9) and can be written equivalently as

$$\begin{aligned}
&V_1 := V := V_{11} \Rightarrow \mathcal{V}_1 = \mathcal{V}_{11}, \\
&V_2 := V - \sqrt{2}V_I := V_{11} - \sqrt{2}V_{12} \Rightarrow \mathcal{V}_2 = \mathcal{V}_{11} - \sqrt{2}\mathcal{V}_{12}, \\
&V_3 := V + \sqrt{2}V_I := V_{11} + \sqrt{2}V_{12} \Rightarrow \mathcal{V}_3 = \mathcal{V}_{11} + \sqrt{2}\mathcal{V}_{12}, \\
&V_4 := 0 \Rightarrow \mathcal{V}_4 = 0, \\
&V_5 := -V_{II} := -V_{13} \Rightarrow \mathcal{V}_5 = -\mathcal{V}_{13}, \\
&V_6 := V_{II} := V_{13} \Rightarrow \mathcal{V}_6 = \mathcal{V}_{13},
\end{aligned}$$

where also the resulting renormalized potentials in the band representation $\mathcal{V}_1$ to $\mathcal{V}_6$ are shown.

Note that the field operator for ABC-trilayer graphene is constructed by summing over all spins, effectively making it spin independent. Thus, the Hamiltonian and the energies derived from it must be considered to be of one spin species. Therefore, the Fermi energy in a calculation depends on the spin channel where the calculation is performed. In order to explicitly calculate the effects of screening, one needs to find the expressions for $\mathcal{V}_{j1}$ to $\mathcal{V}_{j3}$ in terms of Γ_{ij}. Subsequently, the explicit Green's function must be found, such that the values of Γ_{ij} can be evaluated.

CHAPTER 6

Conclusions

A large part of this thesis was dedicated to the review of ferromagnetism in monolayer and bilayer graphene. The resulting phase diagrams emerging from these calculations have already been described in the literature and the results are well established. However, these results were all calculated analytically whereas the same results were, in the thesis, reproduced by numerical calculations. The calculations of the ferromagnetic properties of ABC-trilayer graphene have been extended from these numerical calculations using a tight-binding approach, only including nearest neighbor hopping parameters t and $t_\perp$ (equivalent notation is γ_0 and γ_1, respectively).

The thesis started with an introduction to numerical methods which have been used throughout the thesis. As stated early on, it is possible to find an analytic expression that approximate the exchange integral ΔE_{ex} for bilayer graphene. However, as it can be seen from the resulting expression calculated by Nilsson et al. [13], this expression is quite long and complicated. Due to the summation over the electronic bands increasing to six terms compared to four in bilayer graphene and due to the number of matrices χ^α increasing from two to six, the equivalent expression for trilayer graphene would grow even larger. The exchange integral was therefore calculated using numerical methods. In particular, since higher order Newton-Cotes methods are slow to converge, the double exponential integration algorithm extended to multiple variables was chosen as the most suitable candidate to handle the integration due to its speed of convergence and ability to handle integrands tending to infinity at the integration boundary. After a Duffy coordinate transformation and appropriate splitting of the integrals, all singularities of the Coloumb potential were moved to the integration boundaries before performing the integrations.

In the work done by Nilsson et al. [13], the low energy approximation of the momentum space Hamiltonian in a 4×4 matrix form is diagonalized analytically. Thus, for bilayer graphene there exists an analytic expression for the single-particle dispersion relation and hence for the density of states. The calculation for bilayer graphene was first reproduced using these analytic expressions combined with numerical integration of ΔE_{ex}. Since the corresponding 6×6 matrix for ABC-stacked trilayer graphene is complicated to diagonalize analytically, and even more complicated when an extended number of hopping parameters are involved, a numerical diagonalization was chosen. The diagonalization was at first attempted by using a computer algebra package, where only hopping parameters t and $t_\perp$ were included, but resulted in extremely complicated expressions. There are many diagonalization algorithms available. The most common of these will first reduce the matrices to tri-diagonal form and then employ an algorithm capable of diagonalizing tri-diagonal matrices efficiently. These algorithms are quite complicated but fast for large matrices.

However, for smaller matrices, the much simpler Jacobi diagonalization algorithm is sufficient and can in some cases be faster [51]. Thus, Jacobi diagonalization was the algorithm of choice. The electronic bands in the single particle dispersion relation are simply the entries of the diagonalized matrix. However, the much needed density of states and the kinetic energy are derived from one of the electronic bands through differentiations, single variable integrations and the inverse of the single particle dispersion. This means that these properties have to be calculated numerically. Without taking screening effects into account, the numerical differentiation could be avoided by utilizing the partial integration leading to equation (5.1). However, a single variable version of the double exponential algorithm was needed to find the kinetic energy (although in this case there are no singularities and a Newton-Cotes algorithm would have worked just as well). Also, an algorithm was created to find the inverse of the dispersion. In principle, given a cardinality n set of values $\{\epsilon(k_i)\}$ of the dispersion, one could use a linear $\mathcal{O}(n)$ search algorithm to find the inverse. This was however found to be a slow algorithm and a more suitable $\mathcal{O}(\log_2 n)$ binary search algorithm has been constructed for this set, combined with a linear interpolation to better approximate the inverse value. In order to validate the numerical calculations just mentioned, they were implemented into the calculations of the bilayer phase diagram. Thus, the bilayer code listed in appendix F presents a choice between analytic and numerical diagonalization. It turns out that finding a faster way of performing these diagonalizations would give the largest speedup of the calculations compared to the implementations shown in the appendices.

In order to extend the bilayer calculations to calculate the ABC-trilayer phase diagram, new matrices χ^α needed to be calculated based on the diagonalization matrices. The matrices χ^α arise from diagonalizing the potential matrix M of equation (5.3). Directly diagonalizing M is possible, but leads to a complicated expression. Doing this would make it difficult to write the exchange energy ΔE_{ex} in a way equivalent to the corresponding expression for bilayer graphene. However, by splitting this matrix into a tridiagonal term and an off diagonal term, it is possible to obtain simple eigenvalues and eigenvectors by the separate diagonalization of these terms. The off diagonal matrix can in principle be written as a 2×2 matrix, but later in the calculations it becomes convenient if it is augmented to a 3×3 matrix with a row of zeros. This is due to the diagonalization matrix $\mathcal{M}$ that diagonalize the Hamiltonian being a 6×6 matrix. By manipulating the matrix M in this manner it was possible to get to the expression in equation (5.10), which is nearly identical to the corresponding expression for bilayer graphene which is shown in equation (4.19).

Once the expressions for the kinetic energy E_k and the exchange energy E_{ex} are in place and algorithms have been implemented that are able to solve these quantities, values $E = E_k + E_{ex}$ are calculated as a function of Fermi energies of the two existing spin channels. The Fermi momenta for the ferromagnetic spin up channel, ferromagnetic spin down channel and paramagnetic spin channels are given by $Q_\uparrow$, $Q_\downarrow$ and Q_d, respectively. By obeying particle conservation, E is calculated as a function of one parameter x. For one type of charge carrier, $x = Q_\downarrow^2$ and $Q_\uparrow^2 = 2Q_d^2 - x$, where $x \geq 0$. For two types of charge carriers, $|x| = Q_\downarrow^2$ and $Q_\uparrow^2 = 2Q_d^2 + |x|$, where $x < 0$. The Fermi levels of the two spin channels determine whether a phase of the material is ferromagnetic or paramagnetic. Subsequently, the difference in energy E between a prepared ferromagnetic phase and a prepared paramagnetic phase ΔE can be plotted as a function of the parameter x. The minimum of $\Delta E(x)$ determines the preferred phase of the material. Thus, we developed an algorithm to estimate this minimum, given a set of points $\{\Delta E(x_i)\}$. A further approximation of the minimum is found by quadratic interpolation.

The last step is to find a point in the electron-electron coupling g versus doping $n = Q_d^2/2$ plane. By fixing g and using the techniques described above it is possible to find several minima of ΔE as a function of the doping level n. An example of this was shown in figure 5.4. The critical point is where the system undergoes a phase transition from a paramagnetic to a ferromagnetic phase, i.e. at the doping level where the minimum of ΔE vanishes. In order to efficiently converge

to the critical point, a binary search algorithm has been used.

The resulting phase diagram for ABC-stacked trilayer graphene, without including the effects of screening of the Coloumb potential, is shown in figure 5.7. Furthermore, it is found that all the phase transitions are of first order. An example of this is shown in figure 5.4 for $g = 6$, where it can be seen that the critical curve (thick black line) has two minima, which will result in a discontinuous jump of the order parameter to zero. Another interesting feature of the phase diagram is the ferromagnetic behavior at zero doping. This can be compared with monolayer graphene, which exhibits both first and second order phase transitions [12]. Figure 3.7 illustrates a first order phase transition and figure 3.8 illustrates a second order one. Thus, the phase diagram of monolayer graphene has two distinct regions, which is not the case for the diagram of ABC-trilayer graphene. Also, monolayer graphene has been found to be paramagnetic at zero doping. Comparing the phase diagram of ABC-trilayer graphene with that of bilayer graphene in figure 4.12 and that of ABA-trilayer graphene [23] one finds that they are quite similar. In fact they all share the ferromagnetic behavior at zero doping and the fact that there are only first order phase transitions. However, looking at the strength of the doping around $g \approx 2.8$, which is the electron-electron coupling found for graphite, it is evident that the critical doping n is approximately 50 times stronger for the ABC-trilayer than for bilayer graphene, while the ferromagnetism in bilayer is stronger than in monolayer graphene. Comparing with ABA-trilayer graphene, the ferromagnetic effects are about 270 times stronger in ABC-trilayer graphene. A strong ferromagnetic behavior is advantageous in trying to detect its presence. This is because it is experimentally difficult to produce graphene samples with low doping levels.

So far, screening of the Coloumb potential has not been taken into effect. These effects are outlined in section 5.3 by first devising a path integral representation of the system. Once this has been done, both the effects of exchange interactions and the effects of screening can be calculated using a perturbative approach. The effects of exchange interactions is seen as first order corrections, where only the Fock term survives due to the positive Jellium background. The screening effects are calculated by using corrections to infinite order, but including only terms with highest divergence factor and applying a Dyson like equation to renormalize the potential. This is the well known random phase approximation which leads to a Thomas-Fermi screened potential in a 3D gas. Due to the diverging nature of the density of states for ABC-trilayer graphene, these effects could have a significant implication on the ferromagnetic behavior. The effects of screening should therefore be taken into account in future calculations.

Only two hopping parameters have been taken into account. However, it has been found that $t_3 \sim 0.315$eV [24], which is sizable in comparison to $t \sim 3$eV and $t_\perp \sim 0.35$eV. These hopping parameters are described in the Slovenczewski-Weiss-McClure model for graphite [59, 60]. Thus, it is reasonable to believe that including t_3 will change the dispersion and therefore the ferromagnetic behavior. Because of the numerical nature of the integrations it is now much easier to extend the calculations to include more hopping parameters. However, in order to specify the Fermi momenta Q_i in the current model, it is assumed that the dispersion has rotational symmetry in the (k_x, k_y)-plane. This symmetry will most likely be broken by including t_3, which will make the integration boundaries of the exchange integration difficult to handle numerically. It will also be a challenge to define the Fermi momenta Q_i in an equivalent manner to the existing model since Q_i today represents a radius in the (k_x, k_y)-plane. Nevertheless, this work should pave the road to possible extensions of the existing models for the investigation of ferromagnetism in multi-layer graphene using numerical methods.

APPENDIX A

Introduction to solid state physics

A.1 Crystal Structures

We will in this section cover the basic definitions of a crystal structure following the treatment in chapter 1 of ref. [57].

A.1.1 Definition of a crystal structure

A *lattice* is a repetitive structure that can be described fully by its *lattice translation vectors*, $\{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$. Any site on the lattice $\mathbf{r}'$, can be reached, relative to any lattice site $\mathbf{r}$ through

$$\mathbf{r}' = \sum_{i=1}^{d} u_i \mathbf{a}_i$$

where $u_i \in \mathbb{Z}$ and $d \in \{1, 2, 3\}$ is the dimension of the lattice. The lattice translation vectors are depicted in figure A.1 for a 2D lattice.

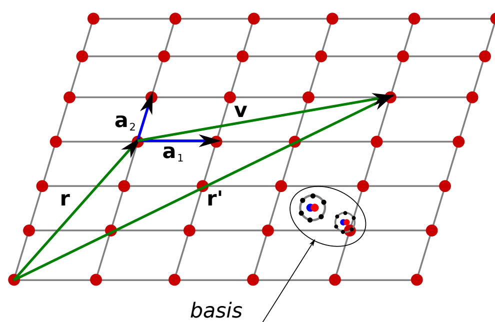


Figure A.1: A 2D lattice with its associated lattice translation vectors $\mathbf{a}_1$ and $\mathbf{a}_2$. In this example $\mathbf{v} = 3\mathbf{a}_1 + \mathbf{a}_2$.

Figure A.1 also shows the concept of a *basis*. A *lattice cell* is the space spanned by the lattice translation vectors $\{\mathbf{a}_i\}$ restricted to the ranges $[0, |\mathbf{a}_i|)$, such that the volume of one cell in 3D is

$$V = |\mathbf{a}_3 \cdot (\mathbf{a}_1 \times \mathbf{a}_2)|$$

and the area of a 2D cell is

$$A = |\mathbf{a}_1 \times \mathbf{a}_2|$$

Then, a basis consists of a group of atoms, within a lattice cell, making a structure that is repeated throughout the lattice. In figure A.1, the basis consists of two different atoms (depicted as one large and one small atom). Note that only one basis is depicted in the figure, and that the basis is repeated throughout the lattice. If we index the atoms by the index j, then the position $\mathbf{r}_j$ of each atom within the basis can be described relative to any cell by

$$\mathbf{r}_j = \sum_{i=1}^{d} x_{i,j}\mathbf{a}_i$$

where, by definition we will have $x_{i,j} \in [0, 1)$.

A *crystal structure* is defined by the lattice of a crystal plus the associated basis. It is worth noting that there are many ways of choosing the lattice basis and the associated translation vectors $\{\mathbf{a}_i\}$ for a lattice. In figure A.1, we could for example move the lattice translation vectors such that either of the two atoms in the basis would line up with one of the lattice sites. This would constitute two different choices. We could also rotate the lattice translation vectors to align the two atoms differently relative to the resulting lattice cells.

A.1.2 Symmetries

A *symmetry operation* is defined as an operation that, when invoked on the crystal structure, transforms the crystal structure onto itself. Note that it is not enough to turn the lattice into itself. The basis must also remain the same after a symmetry operation.

There are three types of possible symmetry operations:

- *Translational symmetry*, by $\mathbf{T} = \sum_{i=1}^{d} u_i\mathbf{a}_i$
- *Rotational symmetry*, which is a subtype of a *point symmetry*
- *Reflection symmetry*, which is a subtype of a *point symmetry*

The lattice of figure A.1 will, for arbitrary $\{\mathbf{a}_i\}$, only be invariant under rotations of π and 2π. This type of lattice is called an *oblique lattice*. Additional symmetries can be imposed by further restrictions on the lattice and on $\{\mathbf{a}_i\}$. There are four further restrictions that can be made. The oblique lattice in addition to the four restrictions constitute the five *Bravais lattices*:

- Oblique lattice: Generic 2D lattice $\{\mathbf{a}_i\}$ with $i = 1, 2$
- Square lattice: $|\mathbf{a}_1| = |\mathbf{a}_2|$ and $\angle(\mathbf{a}_1, \mathbf{a}_2) = 90°$
- Hexagonal lattice: $|\mathbf{a}_1| = |\mathbf{a}_2|$ and $\angle(\mathbf{a}_1, \mathbf{a}_2) = 120°$. Every second row is translated 1/2 of the *lattice parameter* (i.e. spacing between lattice sites), relative to the first row of lattice sites.
- Rectangular lattice: $|\mathbf{a}_1| \neq |\mathbf{a}_2|$ and $\angle(\mathbf{a}_1, \mathbf{a}_2) = 90°$
- Centered rectangular: $|\mathbf{a}_1| \neq |\mathbf{a}_2|$. Every second row is translated 1/2 of the *lattice parameter* (i.e. spacing between lattice sites), relative to the first row of lattice sites.

In the same manner, we can impose restrictions on 3D lattices and there will be 14 subgroups of lattices exhibiting increased symmetries compared to the *triclinic* lattice where $|\mathbf{a}_1| \neq |\mathbf{a}_2| \neq |\mathbf{a}_3|$ and all angles are unequal. Three of these lattices are found in the *cubic lattice* and they are shown in figure A.2.

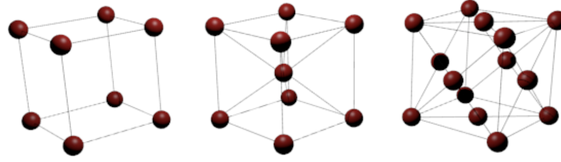


Figure A.2: The cubic lattice. From left to right: Simple Cubic (SC), Body Centered Cubic (BCC), Face Centered Cubic (FCC).

A.1.3 The notion of a primitive cell

Figure A.3 shows the concept of *primitive axes* and that of a *primitive cell.* In the figure, $\{\mathbf{a}_i\}$ and $\{\mathbf{a}'_i\}$ constitute primitive sets of axes, while the shaded area spanned by these axes form the primitive cells. Note that the lattice l, which is in the shaded region spanned by $\{\mathbf{a}_i\}$, can be reached by a linear combination of the set $\{\mathbf{a}_i\}$ using integral coefficients. The same is true for the lattice site m, that can be reached by a linear combination of $\{\mathbf{a}'_i\}$ using integral coefficients. However, the lattice site n can only be reached by a non-integer linear combination of the set $\{\mathbf{a}''_i\}$, which makes these axes non-primitive, and forming a non-primitive lattice cell.

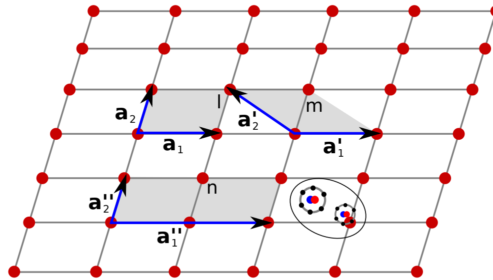


Figure A.3: A 2D lattice with two primitive set of axes $\mathbf{a}_1$ and $\mathbf{a}_2$, and one non primitive set of axes $\mathbf{a}'_1$ and $\mathbf{a}'_2$.

In both of the primitive cells in figure A.3, we see that there are

$$4\,\frac{\text{cells}}{\text{lattice site}}$$

shared (i.e. each lattice site has 4 cells connected to it). Which means that we can only contribute

$$\frac{1}{4}\frac{\text{lattice site}}{\text{cell}}$$

for each lattice site in a cell. There are 4 lattice sites (that are all shared) in a cell, and therefore an average of $\frac{1}{4}\cdot 4 = 1$ lattice site per cell, in total. The same argument holds in 3D, where each lattice site is shared between 8 cells $\Rightarrow \frac{1}{8}$ of a lattice site per cell for each site. Since there are 8 lattice sites (that are all shared) in a cell, then there is on average $\frac{1}{8}\cdot 8 = 1$ lattice site per cell, in total.

One possible way of constructing a primitive cell, given a lattice, is the following:

1. Pick a lattice site, lets call it c
2. From c, extend lines connecting every lattice site, lets call them l_i, closest to c in all directions
3. Draw lines perpendicular to all of these lines, crossing midway between c and l_i
4. The smallest area resulting from the perpendicular lines is a primitive cell

The resulting primitive cell is called a *Wigner-Seitz cell* and a 2D example is shown in figure A.4. It is easy to see that the Wigner-Seitz cells, when spread throughout the lattice, will cover the whole lattice. If we chose any of the lattice sites l_i to be the center of the cell, then the cell border that resulted from the line c to l_i is the exact same border that will be drawn due to the line l_i to c (in the case where l_i is the center of the cell). Thus, all the borders of all the cells must exactly overlap.

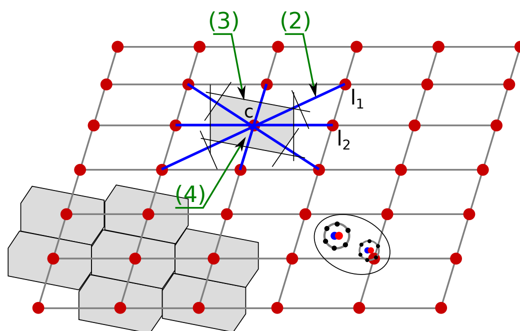


Figure A.4: An example of a Wigner-Seitz cell.

A.1.4 ABA and ABC lattice structures

We can create a lattice structure by stacking spheres in such a way that all the surfaces of the spheres touch. Figure A.5 outlines the possible stacking schemes. We always start out stacking the bottom layer of spheres as shown in figure A.5, where all the centers of the spheres are placed on the A-type layer lattice cells. The next layer of spheres must be positioned, with their center, on the B-type layer lattice cells in such a way that all the spheres in the B-type layer touch three other spheres on the A-type layer. Note that it would not matter if we redefined B $\leftrightarrow$ C as long as we stay consistent throughout the layers (B $\leftrightarrow$ C amounts to looking at the lattice from the other side). We only have two possible options for the next layer, if we wish each sphere on this layer to touch three spheres on the layer below. The first option is to place the centers of the spheres on the B-type layer lattice sites, in which case we get an ABA stacking (the stacking would continue ABABABA...). The second option would be to stack the new layer of spheres with their centers on the C-type layer lattice cells, in which case we have an ABC stacking scheme.

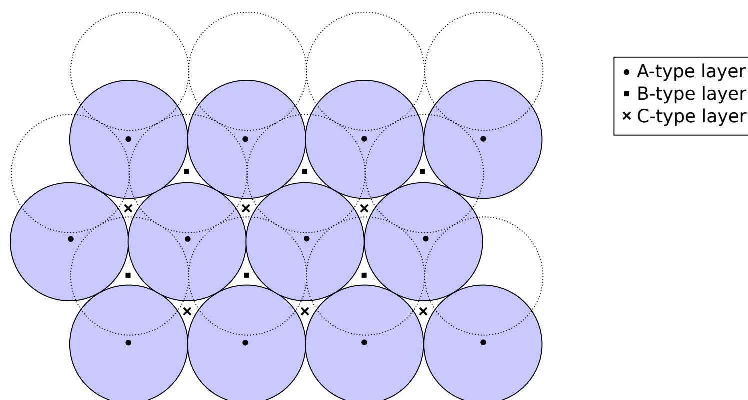


Figure A.5: Possible stacking schemes, using closely packed spheres.

Figure A.6 shows a 3D figure of the ABA stacking scheme, where we have indicated a primary

cell within this scheme by thin cylinders connecting the lattice points. The closely packed spheres of the bottom layer is also indicated. We clearly see that the ABA type stacking scheme leads to a hexagonal primary cell structure, but with the B-layer translated. This is called a *hexagonal closed-packed structure*.

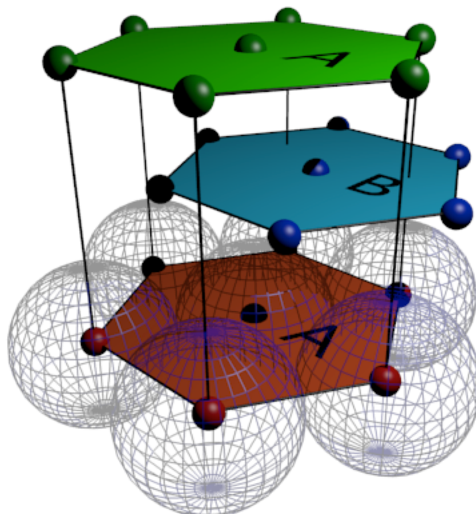


Figure A.6: The *hexagonal closed-packed structure* resulting from ABA stacking of the spheres.

Figure A.7 shows the ABC stacking scheme. We see that this primary cell becomes of the face centered cubic type.

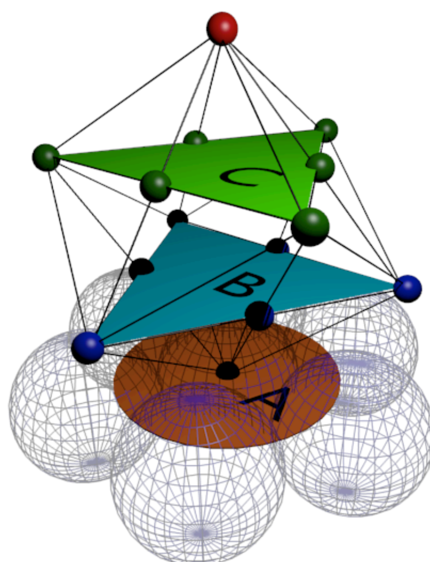


Figure A.7: The *face-centered cubic structure* resulting from ABC stacking of the spheres.

A.1.5 Indexing of planes

In order to give a plane an *index*, we first denote the plane by the three coordinates (x, y, z) where the plane intersects the $\{\mathbf{a}_i\}$ axis. Then, the corresponding index of the plane is denoted by three

numbers in parenthesis

$$\left(\left\{K\frac{1}{x}\right\}\left\{K\frac{1}{y}\right\}\left\{K\frac{1}{z}\right\}\right)$$

where K is the minimum multiple of x, y and z.

A.2 Diffraction and the Reciprocal Lattice

In this section we will cover the basic theory of diffraction and the definition of the reciprocal lattice following the treatment in chapter 2 of ref. [57].

A.2.1 Bragg scatteing

We can imagine the different planes of a crystal being stacked on top of each other, separated by the lattice separation d as shown in figure A.8. There are two beams, A and B, either an electromagnetic beam or a particle beam, incident on the planes at an angle θ. We assume that the planes are not totally reflecting, but they do reflect the same amount of the beam at an output angle θ. If we further assume a wavelength λ of the incoming beam and elastic scattering, then the output beam also has a wavelength λ. The B beam will travel a distance $L = d\sin\theta$ further than beam A, before hitting a plane at the same horizontal point. When the beams exit, then the B beam has traveled $2L$ further than the A beam. Since the beams are parallel, they can interfere. Thus, we need to have $2L = n\lambda$, where $n \in \mathbb{Z}$ to achieve constructive interference, i.e.

$$2d\sin\theta = n\lambda$$

This is the Bragg scattering condition.

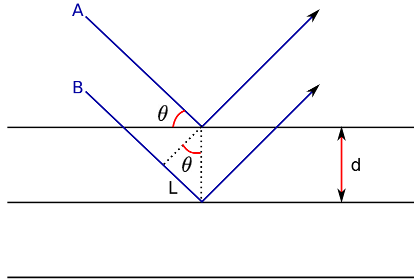


Figure A.8: The principle of Bragg scattering.

A.2.2 The reciprocal lattice

A lattice is periodic in the translation vector

$$\mathbf{T} = \sum_{i=1}^{d} v_i \mathbf{a}_i,$$

where $v_i \in \mathbb{Z}$. This means that the charge density satisfies

$$n(\mathbf{r} + \mathbf{T}) = n(\mathbf{r}). \tag{A.1}$$

Thus, we can describe $n(\mathbf{r})$ in terms of its Fourier components

$$n(\mathbf{r}) = \sum_{\mathbf{G}} n_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{r}} \tag{A.2}$$

as long as we make sure that (A.1) is satisfied. To this end, we choose

$$\mathbf{b}_1 = \tfrac{2\pi}{V}\mathbf{a}_2 \times \mathbf{a}_3 \quad \mathbf{b}_2 = \tfrac{2\pi}{V}\mathbf{a}_3 \times \mathbf{a}_1 \quad \mathbf{b}_3 = \tfrac{2\pi}{V}\mathbf{a}_1 \times \mathbf{a}_2,$$

where

$$V = \mathbf{a}_1 \cdot (\mathbf{a}_2 \times \mathbf{a}_3).$$

Furthermore, if

$$\mathbf{G} = \sum_{i=1}^{d} u_i \mathbf{b}_i,$$

where $u_i \in \mathbb{Z}$, then we see that

$$\mathbf{G} \cdot \mathbf{T} = \sum_{i,j=1}^{d} u_i v_j \mathbf{a}_i \cdot \mathbf{b}_j = \sum_{i,j=1}^{d} u_i v_j 2\pi \delta_{ij} = 2\pi \sum_{i=1}^{d} u_i v_i,$$

where we used that

$$\mathbf{a}_i \cdot (\mathbf{a}_2 \times \mathbf{a}_3) = V\delta_{i,1} \quad \mathbf{a}_i \cdot (\mathbf{a}_3 \times \mathbf{a}_1) = V\delta_{i,2} \quad \mathbf{a}_i \cdot (\mathbf{a}_1 \times \mathbf{a}_2) = V\delta_{i,3}.$$

Since $u_i, v_j \in \mathbb{Z}$, we have that

$$e^{\mathbf{G}\cdot\mathbf{T}} = e^{2\pi \sum_{i=1}^{d} u_i v_i} = 1$$

and

$$n(\mathbf{r} + \mathbf{T}) = \sum_{\mathbf{G}} n_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{r}} e^{i\mathbf{G}\cdot\mathbf{T}} = n(\mathbf{r}),$$

which proves that our choice of $\mathbf{G}$ satisfies (A.1). The vectors $\{\mathbf{a}_i\}$ have units of length, while the vectors $\{\mathbf{b}_i\}$ have units of inverse length and are therefore called the *reciprocal lattice vectors*. The lattice formed by the set of translation vectors $\{\mathbf{b}_i\}$ is called the *reciprocal lattice*.

A.2.3 The scattering amplitude

We now look at figure A.9 and consider an amplitude dF of a beam scattered of a volume element dV. The amplitude dF for the beam B alone will be determined by the total volume dV that the beam reflects from times the charge density $n(\mathbf{r})$ centered at the element dV. Let us now consider a second beam, A, a distance $\mathbf{r}$ apart. As seen from the subsection on Bragg scattering, there will be an interference between the two beams if they are close enough. The resulting phase difference ϕ between the incoming $\mathbf{k}$ beams becomes

$$\frac{\phi}{2\pi} = \frac{L}{\lambda} \Rightarrow \phi = \frac{2\pi}{\lambda}|\mathbf{r}|\cos\theta = k|\mathbf{r}|\cos\theta = \mathbf{k}\cdot\mathbf{r},$$

where $\mathbf{k}$ and $\mathbf{k}'$ are wave vectors of the incoming and outgoing beams, respectively. Thus, we find a phase difference between the incoming and outgoing beams of $\Delta\phi = (\mathbf{k} - \mathbf{k}')\cdot\mathbf{r}$, leading to the phase factor

$$e^{i(\mathbf{k}-\mathbf{k}')\cdot\mathbf{r}} \equiv e^{-i\Delta\mathbf{k}\cdot\mathbf{r}},$$

that needs to be multiplied with $dVn(\mathbf{r})$ to yield the final amplitude dF. We can now integrate over the entire volume to obtain the total scattering amplitude F:

$$F = \int dV n(\mathbf{r}) e^{-i\Delta\mathbf{k}\cdot\mathbf{r}}. \tag{A.3}$$

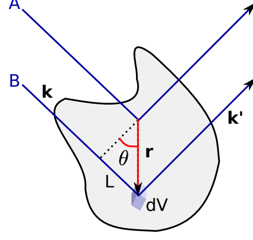


Figure A.9: Visualization of the phase difference between an incoming beam and an outgoing refracted beam.

A.2.4 The diffraction condition

By inserting equation (A.2) into equation (A.3), we obtain

$$F = \int dV \sum_{\mathbf{G}} n_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{r}} e^{-i\Delta\mathbf{k}\cdot\mathbf{r}} = \sum_{\mathbf{G}} n_{\mathbf{G}} \int d\mathbf{r} e^{i(\mathbf{G}-\Delta\mathbf{k})\cdot\mathbf{r}} = \sum_{\mathbf{G}} n_{\mathbf{G}} V \delta_{\mathbf{G}-\Delta\mathbf{k}} = V n_{\Delta\mathbf{k}}$$

where we integrated over the entire volume V. Thus, since the Fourier components $n_{\Delta\mathbf{k}}$ can only exist on the reciprocal lattice points defined by $\mathbf{G}$, we must have the condition

$$\Delta\mathbf{k} = \mathbf{G}$$

This is the *diffraction condition*. For elastic scattering, it holds that $|\mathbf{k}| = |\mathbf{k}'|$. We then have that

$$\Delta\mathbf{k} = \mathbf{k}' - \mathbf{k} = \mathbf{G} \Rightarrow (\mathbf{G} + \mathbf{k})^2 = \mathbf{k}^2,$$

which becomes

$$G^2 + 2\mathbf{G}\cdot\mathbf{k} + k^2 = k^2 \Rightarrow 2\mathbf{G}\cdot\mathbf{k} = -G^2.$$

However, since $-\mathbf{G}$ also defines a reciprocal lattice, we can write the diffraction condition as

$$2\mathbf{G}\cdot\mathbf{k} = G^2.$$

A.2.5 The first Brillouin zone

We define the *first Brillouin zone* as a Wigner-Seitz cell in reciprocal lattice space (i.e. Fourier space or **k**-space). The construction is the same as for a Wigner-Seitz cell in normal space, but is done in reciprocal lattice space. Firstly, choose a reciprocal lattice vector $\mathbf{G}$, then place a plane perpendicular to this vector at half the length of $\mathbf{G}$. Continue to do this for all the lattice vectors $\mathbf{G}$ closest to the center of the Brillouin zone to be constructed. Then, the resulting planes encapsulating the smallest volume define the first Brillouin zone.

If we rewrite the diffraction condition by dividing both sides with 4, we find

$$\frac{1}{2}\mathbf{G}\cdot\mathbf{k} = \left(\frac{1}{2}G\right)^2. \tag{A.4}$$

This shows that $\mathbf{k}$ must meet the Brillouin zone walls exactly in order for the scattering condition to hold.

If we let $\mathcal{V}_c$ be the volume of a single Brillouin zone, then the scattering $F_{\mathbf{G}}$ from N cells is

$$F_{\mathbf{G}} = N \int_{\mathcal{V}_c} dV n(\mathbf{r}) e^{-i\mathbf{G}\cdot\mathbf{r}} \equiv N S_{\mathbf{G}},$$

where $S_{\mathbf{G}}$ is called the *structure factor*. We can further divide the charge density $n(\mathbf{r})$ into a sum of charge densities resulting from each individual atom within the Brillouin zone, to obtain

$$S_{\mathbf{G}} = \sum_{i=1}^{M} \int_{\mathcal{V}_c} dV n_i(\mathbf{r} - \mathbf{r}_i) e^{-i\mathbf{G}\cdot\mathbf{r}},$$

where M is the number of atoms in the zone. Substituting $\boldsymbol{\rho} = \mathbf{r} - \mathbf{r}_i$, we find that $d\boldsymbol{\rho} = d\mathbf{r}$ and

$$S_{\mathbf{G}} = \sum_{i=1}^{M} \int_{\mathcal{V}_c} dV n_i(\boldsymbol{\rho}) e^{-i\mathbf{G}\cdot(\boldsymbol{\rho}+\mathbf{r}_i)} = \sum_{i=1}^{M} e^{-i\mathbf{G}\cdot\mathbf{r}_i} \int_{\mathcal{V}_c} dV n_i(\boldsymbol{\rho}) e^{-i\mathbf{G}\cdot\boldsymbol{\rho}} \equiv \sum_{i=1}^{M} e^{-i\mathbf{G}\cdot\mathbf{r}_i} f_i$$

where f_i is called the *atomic form factor*. The details of the atomic structures are now fully described by f_i, while the details of the lattice is contained in $\mathbf{G}$. It can be shown that different reciprocal lattice structures $\{\mathbf{b}_j\}$ will lead to requirements on the factors $\{u_j\}$ in $\mathbf{G}$ in order for $S_{\mathbf{G}}$ to become non-vanishing. For example, the bcc lattice requires that $\sum_{j=1}^{d} u_j$ is even for a non-vanishing $S_{\mathbf{G}}$ (i.e. a non-vanishing scattering amplitude). This reflects the fact that some of the lattice sites do not scatter an incoming beam. This provides a way of identifying lattice structures by diffraction.

A.3 Band structures

In this section, we will cover the basics of band structures following the treatment in chapter 7 of ref. [57].

A.3.1 Nearly free electron model

We saw in section A.2.5 that diffraction occurs only when the momentum vectors $\mathbf{k}$ lie on the boundary of the Brillouin zone (in reciprocal lattice space). This means that when applying a potential to a material having free electrons, the electron wave functions will reflect off lattice sites exactly when the momentum vectors reach the Brillouin zone, leading these electrons to localize.

When $\mathbf{k}$ is at the Brillouin zone, the electron wave functions form standing waves due to total reflection, thus forming a periodic buildup of electrons in normal lattice space. Because of these buildups of charge, a potential difference will arise between these negative charges and the positive charges of the ions on the lattice.

There are two types of total reflections that can occur, one is when the left and right mover waves have the same phase, the other is when they are π degrees out of phase. We consider the plane wave

$$\psi(t, \mathbf{x}) = Ae^{i\omega t - i\mathbf{k}\cdot\mathbf{x}}.$$

Then, we assume $\mathbf{k}||\mathbf{G}$. In that case, $\mathbf{k} = \pm\frac{1}{2}\mathbf{G}$, in order for $\mathbf{k}$ to lie on the Brillouin zone (i.e. to be a solution to equation (A.4)). Thus, we get a reflection, causing left and right movers:

$$\psi_R(t, \mathbf{x}) = Ae^{i\omega t - i\frac{1}{2}\mathbf{G}\cdot\mathbf{x}},$$

$$\psi_L(t, \mathbf{x}) = Ae^{i\omega t + i\frac{1}{2}\mathbf{G}\cdot\mathbf{x}},$$

resulting in total wave functions

$$\psi_\pm(t, \mathbf{x}) = \psi_L(t, \mathbf{x}) \pm \psi_R(t, \mathbf{x}) = Ae^{i\omega t}\left(e^{+i\frac{1}{2}\mathbf{G}\cdot\mathbf{x}} \pm e^{-i\frac{1}{2}\mathbf{G}\cdot\mathbf{x}}\right) = \begin{cases} 2Ae^{i\omega t}\cos(\frac{1}{2}\mathbf{G}\cdot\mathbf{x}) \\ 2iAe^{i\omega t}\sin(\frac{1}{2}\mathbf{G}\cdot\mathbf{x}). \end{cases}$$

We see that the charge buildups $\rho_+ = |\psi_+|^2$ for ψ_+ are at the lattice sites, while for ψ_- the charge buildups are in between the lattice sites. Thus, the potential difference between ψ_+ and ions at

the lattice is lower than between ψ_- and the corresponding ions. Since it is energetically more favorable to have the situation of ψ_+, we obtain the following behavior of the free electrons on the lattice as a function of external potential V:

1. For $V = 0$ the electrons are free.
2. For increasing V, the electron momenta $\mathbf{k}$ increase and the electrons move through the material.
3. At some V_l, the electron momenta $\mathbf{k}$ reach the Brillouin zone. Diffractions ψ_+ form and the electrons no longer move.
4. As V increases further, the excess energy is used to form more and more diffractions ψ_-; during this increase in V there is still no net transport.
5. At some V_h, all diffractions are of the type ψ_+, and there is still no net transport.
6. When $V > V_h$, the electron momena $\mathbf{k}$ will grow beyond the Brillouin zone and once again the electrons behave freely, such that a net transport takes place.

The energy range from V_l to V_h constitutes a forbidden energy region, called a *band gap*. Notice that we assumed no interactions from the lattice structure if $\mathbf{k}$ is away from the Brillouin zone (thus, away from the Brillouin zone we consider the system as a free electron gas). This model is therefore called the *nearly free electron model*. The energy of the band gap is calculated by

$$E_g = \int_{\mathcal{V}_c} d\mathbf{x}\rho(x)U(\mathbf{x}) = \int_{\mathcal{V}_c} d\mathbf{x}(|\psi_+(\mathbf{x})|^2 - |\psi_-(\mathbf{x})|^2)U(\mathbf{x})$$

where $U(\mathbf{x}) = U\cos(\mathbf{G}\cdot\mathbf{x})$ is the potential of the ions on the lattice and ρ is the charge difference of electron buildups. $\mathcal{V}_c$ is the volume of a cell in the normal lattice space. With this choice of $U(\mathbf{x})$, we get

$$\begin{aligned} E_g &= \int_{\mathcal{V}_c} d\mathbf{x}4A^2\left[\cos^2\left(\frac{1}{2}\mathbf{G}\cdot\mathbf{x}\right) - \sin^2\left(\frac{1}{2}\mathbf{G}\cdot\mathbf{x}\right)\right]U\cos(\mathbf{G}\cdot\mathbf{x}) \\ &= 4A^2U\int_{\mathcal{V}_c} d\mathbf{x}\cos^2(\mathbf{G}\cdot\mathbf{x}) = 2A^2U\int_{\mathcal{V}_c} d\mathbf{x}(1+\cos(2\mathbf{G}\cdot\mathbf{x})) \\ &= 2A^2U\mathcal{V}_c + 2A^2U\oint_{\mathcal{A}_c} d\mathbf{a}\cdot\nabla\cos(2\mathbf{G}\cdot\mathbf{x}) = 2A^2U\mathcal{V}_c, \end{aligned}$$

where we used the identity $\cos^2(x) - \sin^2(x) = \cos(2x)$ and the fact that $\sin 2\mathbf{G}\cdot\mathbf{x}$ vanishes on the boundary of the cell, $\mathcal{A}_c$. Thus, with the normalization

$$A = \frac{1}{\sqrt{2\mathcal{V}_c}}$$

we find that $E_g = U$, i.e. the energy gap equals the Fourier component U of the potential $U(\mathbf{x})$.

A.3.2 The Kronig-Penney model - square potential

The *Kronig-Penny model* makes the simplification of assuming a square potential over the real space lattice as shown in figure A.10. Here, b is the width of the potential, while a is the lattice parameter. In section A.3.3 we derive equation (A.6), called the central equation, which can be used for solving general periodic potentials. This equation can be used to find solutions to the special case of the Kronig-Penney model.

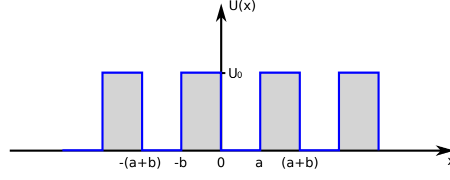


Figure A.10: The Kronig-Penney potential.

A.3.3 General periodic potential model

A general potential $U(\mathbf{x})$ can be Fourier expanded as

$$\begin{aligned} U(\mathbf{x}) &= \sum_{\mathbf{G}} U_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{x}} = \sum_{\mathbf{G}>0} U_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{x}} + \sum_{\mathbf{G}<0} U_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{x}} \\ &= \sum_{\mathbf{G}>0} U_{\mathbf{G}} \left\{ e^{i\mathbf{G}\cdot\mathbf{x}} + e^{-i\mathbf{G}\cdot\mathbf{x}} \right\}, \end{aligned}$$

where we have assumed that $U_{\mathbf{G}} = U_{-\mathbf{G}}$ and that $U_{\mathbf{0}} = 0$. For real Fourier components, $U(\mathbf{x})$ is real. We can now solve the time independent Schroedinger equation for an electron traveling in this potential. To this end, we denote the electron wave function by $\psi(\mathbf{x})$ and Fourier expand it,

$$\psi(\mathbf{x}) = \sum_{\mathbf{k}} C_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{x}}.$$

We then have that (setting $\hbar = 1$)

$$\left\{ -\frac{1}{2m}\nabla^2 + U(\mathbf{x}) \right\} \psi(\mathbf{x}) = \left\{ \frac{\mathbf{k}^2}{2m} + \sum_{\mathbf{G}} U_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{x}} \right\} \sum_{\mathbf{k}} C_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{x}} = \epsilon \sum_{\mathbf{k}} C_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{x}}. \tag{A.5}$$

We now define $\mathbf{k}' = \mathbf{k} + \mathbf{G}$ and perform the following calculation:

$$\sum_{\mathbf{G}} U_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{x}} \sum_{\mathbf{k}} C_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{x}} = \sum_{\mathbf{k},\mathbf{G}} U_{\mathbf{G}} C_{\mathbf{k}} e^{i(\mathbf{k}+\mathbf{G})\cdot\mathbf{x}} = \sum_{\mathbf{k}',\mathbf{G}} U_{\mathbf{G}} C_{\mathbf{k}'-\mathbf{G}} e^{i\mathbf{k}'\cdot\mathbf{x}},$$

where we were allowed to sum over $\mathbf{k}'$ instead of over $\mathbf{k}' - \mathbf{G}$ since $\mathbf{k}$ runs over all $k_i \in (-\infty, \infty)$. Combining this with equation (A.5), we get

$$\lambda_{\mathbf{k}} C_{\mathbf{k}} + \sum_{\mathbf{G}} U_{\mathbf{G}} C_{\mathbf{k}-\mathbf{G}} = \epsilon C_{\mathbf{k}},$$

where we defined $\lambda_{\mathbf{k}} \equiv \frac{\mathbf{k}^2}{2m}$. After a slight rewriting, we obtain the *central equation*:

$$\{\lambda_{\mathbf{k}} - \epsilon\} C_{\mathbf{k}} + \sum_{\mathbf{G}} U_{\mathbf{G}} C_{\mathbf{k}-\mathbf{G}} = 0. \tag{A.6}$$

This equation can be written in matrix form. In principle, the matrix is an infinite matrix and we will only show part of it. We vary $\mathbf{k}$ from $\mathbf{k} - 2\mathbf{g}$ to $\mathbf{k} + 2\mathbf{g}$ in steps of $\mathbf{g} = \min \mathbf{G} > \mathbf{0}$, where we pick some direction of $\mathbf{G}$. Each $\mathbf{k}$ yields one central equation. Furthermore, we assume only one non-vanishing Fourier component $U_{\mathbf{g}} = U_{-\mathbf{g}} \equiv U$:

$$\{\lambda_{\mathbf{k}-2\mathbf{g}} - \epsilon\} C_{\mathbf{k}-2\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}-3\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}-\mathbf{g}} = 0,$$

$$\{\lambda_{\mathbf{k}-\mathbf{g}} - \epsilon\} C_{\mathbf{k}-\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}-2\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}} = 0,$$

$$\{\lambda_{\mathbf{k}} - \epsilon\} C_{\mathbf{k}} + U_{\mathbf{g}} C_{\mathbf{k}-\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}+\mathbf{g}} = 0,$$
$$\{\lambda_{\mathbf{k}+\mathbf{g}} - \epsilon\} C_{\mathbf{k}+\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}} + U_{\mathbf{g}} C_{\mathbf{k}+2\mathbf{g}} = 0,$$
$$\{\lambda_{\mathbf{k}+2\mathbf{g}} - \epsilon\} C_{\mathbf{k}+2\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}+\mathbf{g}} + U_{\mathbf{g}} C_{\mathbf{k}+3\mathbf{g}} = 0,$$

resulting in the matrix equation

$$M\mathbf{c} = \begin{pmatrix} U & \lambda_{\mathbf{k}-2\mathbf{g}} - \epsilon & U & 0 & 0 & 0 & 0 \\ 0 & U & \lambda_{\mathbf{k}-\mathbf{g}} - \epsilon & U & 0 & 0 & 0 \\ 0 & 0 & U & \lambda_{\mathbf{k}} - \epsilon & U & 0 & 0 \\ 0 & 0 & 0 & U & \lambda_{\mathbf{k}+\mathbf{g}} - \epsilon & U & 0 \\ 0 & 0 & 0 & 0 & U & \lambda_{\mathbf{k}+2\mathbf{g}} - \epsilon & U \end{pmatrix} \begin{pmatrix} C_{\mathbf{k}-3\mathbf{g}} \\ C_{\mathbf{k}-2\mathbf{g}} \\ C_{\mathbf{k}-\mathbf{g}} \\ C_{\mathbf{k}} \\ C_{\mathbf{k}+\mathbf{g}} \\ C_{\mathbf{k}+2\mathbf{g}} \\ C_{\mathbf{k}+3\mathbf{g}} \end{pmatrix} = 0.$$

In order for this equation to have non-trivial solutions for $\mathbf{c}$, then M must satisfy $|M| = 0$. Thus, we see that we can find, for a given $\mathbf{k}$, the energy ϵ, which can take several possible values that satisfy $|M| = 0$.

A.3.4 The Bloch theorem

The *Bloch theorem* states that:

- For a periodic potential, the solutions of the Schroedinger equations must be of the form $\psi_k(\mathbf{r}) = u_k(\mathbf{r})e^{i\mathbf{k}\cdot\mathbf{r}}$, where u_k is periodic in the crystal lattice, satisfying $u_k(\mathbf{r}) = u_k(\mathbf{r} + \mathbf{T})$.

We prove this by first Fourier expanding the wave function

$$\psi_k(\mathbf{r}) = \sum_{\mathbf{k}} C_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{r}} = \sum_{\mathbf{k}'-\mathbf{G}} C_{\mathbf{k}'-\mathbf{G}} e^{i(\mathbf{k}'-\mathbf{G})\cdot\mathbf{r}}.$$

Since it does not matter if we perform a sum over $k_i \in (-\infty, \infty)$ or $-G_i \in (-\infty, \infty)$, and we can redefine $\psi_{k'}$ to ψ_k, we can write

$$\psi_k(\mathbf{r}) = \sum_{\mathbf{G}} C_{\mathbf{k}-\mathbf{G}} e^{i(\mathbf{k}-\mathbf{G})\cdot\mathbf{r}} = \left\{ \sum_{\mathbf{G}} C_{\mathbf{k}-\mathbf{G}} e^{-i\mathbf{G}\cdot\mathbf{r}} \right\} e^{i\mathbf{k}\cdot\mathbf{r}} \equiv u_k(\mathbf{r}) e^{i\mathbf{k}\cdot\mathbf{r}},$$

where $u_k(\mathbf{r}) = u_k(\mathbf{r} + \mathbf{T})$ because $e^{-i\mathbf{G}\cdot\mathbf{T}} = 1$.

A.3.5 Metals and insulators

It is possible to count the number of allowed momentum states that an electron can have in a band. Start by assuming N lattice sites, where N is even. Furthermore, we let L denote the length of a one dimensional lattice. An electron gas having electron wave functions with periodic boundary conditions over the length L have the allowed momenta

$$k \in \left\{ \pm\frac{2\pi n}{L}, n \in \mathbb{Z} \right\}.$$

For a crystal lattice structure with the above constraints there will be an energy gap when the momentum reaches the first Brillouin zone. Therefore, the allowed momenta cannot exceed $k = 2\pi/a$, where $a = L/N$. Thus, the allowed momenta are:

$$k \in \left\{ 0, \pm\frac{2\pi}{L}, \pm\frac{4\pi}{L}, \ldots, \frac{2\pi N}{L} \right\}.$$

This set contains exactly N momenta. In three dimensions we have the three momentum vectors k_x, k_y and k_z, each satisfying the same conditions as imposed on k. Therefore, we still have N

allowed momenta, but we can now have two electrons of opposite spin occupying the same momentum. Therefore, the total number of allowed states in three dimensions are $2N$.

If each lattice cell contains one valence electron, then there are N electrons occupying N of the $2N$ possible momentum states. Such a band is therefore half filled. In this case the valence electrons can move to higher momentum states if the crystal is subjected to a potential. The material will then be a conductor.

If each lattice cell contains two valence electrons, then the $2N$ possible momentum states will be occupied by the valence electrons. In this case the electrons cannot move to higher momentum states because all the available states are filled and the band gap prevents any higher momentum states. We then have an insulator.

A.4 Fermi surfaces

We will in this section cover the basic construction of Fermi surfaces following the treatment in chapter 9 of ref. [61].

A.4.1 Definition

A Fermi surface consists of the set of points swept out, in momentum space, by a constant energy ϵ_F. The energy ϵ_F is defined to be the highest occupied energy state at zero temperature.

Since the Fermi surface specifies the occupancy of electrons in the energy bands of a material it will give information about the electrical properties of the material.

A.4.2 Reduced Zone Scheme

It is always possible to translate a momentum vector that is outside the first Brillouin zone into a vector inside of the first Brillouin zone. This is because of the periodicity of the lattice in reciprocal space. The translation is achieved by forming a new wave vector $\mathbf{k}' = \mathbf{k} + \mathbf{G}$, where $\mathbf{k}'$ is an equivalent momentum inside the Brillouin zone. This procedure is depicted in figure A.11.

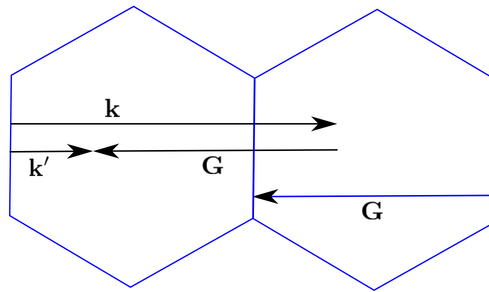


Figure A.11: Translation of a momentum $\mathbf{k}$ outside the Brillouin zone back into a vector $\mathbf{k}'$ inside the Brillouin zone.

Let $\psi_{\mathbf{k}}(\mathbf{r})$, be the Bloch function

$$\psi_{\mathbf{k}}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r})$$

with $\mathbf{k}$ outside the first Brillouin zone. We translate $\mathbf{k}$ to a vector inside the first Brillouin zone, which yields

$$\psi_{\mathbf{k}}(\mathbf{r}) = \psi_{\mathbf{k}'-\mathbf{G}}(\mathbf{r}) = e^{i(\mathbf{k}'-\mathbf{G})\cdot\mathbf{r}} u_{\mathbf{k}'-\mathbf{G}}(\mathbf{r}) = e^{i\mathbf{k}'\cdot\mathbf{r}} \left(e^{-i\mathbf{G}\cdot\mathbf{r}} u_{\mathbf{k}'-\mathbf{G}}(\mathbf{r})\right) \equiv e^{i\mathbf{k}'\cdot\mathbf{r}} u_{\mathbf{k}'}(\mathbf{r}) = \psi_{\mathbf{k}'}(\mathbf{r}).$$

The function $\psi_{\mathbf{k}'}(\mathbf{r})$ is a Bloch function since $u_{\mathbf{k}'}(\mathbf{r})$ is periodic with periodicity $\mathbf{T}$. By translating all momentum outside the zone by a suitable vector $\mathbf{G}$ it is clear that only a solution in the first Brillouin zone is needed. The result of this operation is referred to as the *reduced zone scheme*. Figure A.12 shows an example of the reduced zone scheme for a free electron dispersion.

From figure A.12 we see that, for some $\mathbf{k}$, there are different allowed energies $\epsilon_{\mathbf{k}}$. Each such energy is placed in a different energy band and each energy band will in principle have different functions $u_{\mathbf{k}}(\mathbf{r})$. Therefore, we must have different wave functions associated with each energy band n. Thus, we have

$$\psi_{n,\mathbf{k}}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}} u_{n,\mathbf{k}}(\mathbf{r}) = \sum_{\mathbf{G}} C_{n,\mathbf{k}+\mathbf{G}} e^{i(\mathbf{k}+\mathbf{G})\cdot\mathbf{r}}.$$

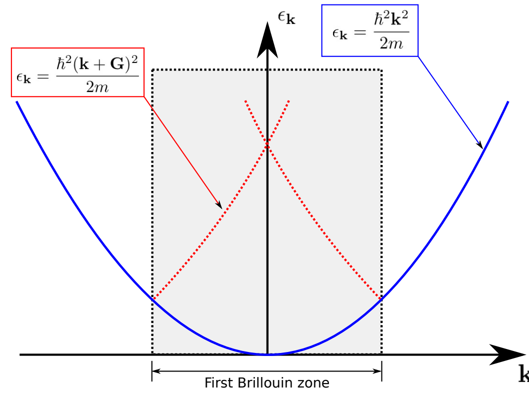


Figure A.12: Example of reduced zone scheme for a free electron.

A.4.3 Periodic and Extended Zone Scheme

If we employ the reduced zone scheme, all energy bands are within the same Brillouin boundaries in $\mathbf{k}$-space. When we have that $\psi_{\mathbf{k}} = \psi_{\mathbf{k}+\mathbf{G}}$, we are allowed to repeat the reduced zone (i.e. the Brillouin zone where all vectors $\mathbf{k}$ are translated to) throughout $\mathbf{k}$-space. This yields a periodic band structure throughout $\mathbf{k}$-space. This is referred to as a *periodic zone scheme*.

In the reduced zone scheme, there can arise several allowed energies for the same momentum $\mathbf{k}$. By the invariance of translations by the lattice translation $\mathbf{G}$ it is possible to have different bands appear in neighboring zones, instead of in the reduced zone. This is referred to as an *extended zone scheme*.

A.4.4 Calculating energy bands - tight binding method

In the method of *tight binding*, one starts out with the wave function of the electron of one atom at position $\mathbf{r}$. A linear combination of single atom wave functions placed at different lattice sites in position space is used to form the wave functions of the electrons on the crystal lattice.

When bringing atoms closer together, the potentials of the nuclei and the other electrons will combine and form a new potential. In reality, this is the potential that should be used together with the Schroedinger equation to solve for the dynamics of free electrons. By instead taking the linear combination of single atom wave functions, we assume a weak interaction between the atoms.

Let, $\phi(\mathbf{r})$ be the ground state single atom wave function. By assuming tight binding we approximate the wave function of the lattice by the linear combination

$$\psi_{\mathbf{k}}(\mathbf{r}) = \sum_i C_{i,\mathbf{k}}\phi(\mathbf{r}-\mathbf{r}_i).$$

Note that, a Bloch function $\tilde{\psi}_{\mathbf{k}}$ has the property

$$\tilde{\psi}_{\mathbf{k}}(\mathbf{r}+\mathbf{T}) = u_{\mathbf{k}}(\mathbf{r}+\mathbf{T})e^{i\mathbf{k}\cdot(\mathbf{r}+\mathbf{T})} = e^{i\mathbf{k}\cdot\mathbf{T}}\tilde{\psi}_{\mathbf{k}}(\mathbf{r}).$$

Choosing $C_{i,\mathbf{k}} = Ce^{i\mathbf{k}\cdot\mathbf{r}_i}$ results in

$$\psi_{\mathbf{k}}(\mathbf{r}+\mathbf{T}) = C\sum_i e^{i\mathbf{k}\cdot\mathbf{r}_i}\phi(\mathbf{r}-[\mathbf{r}_i-\mathbf{T}]) = Ce^{i\mathbf{k}\cdot\mathbf{T}}\sum_i e^{i\mathbf{k}\cdot(\mathbf{r}_i-\mathbf{T})}\phi(\mathbf{r}-[\mathbf{r}_i-\mathbf{T}]) = e^{i\mathbf{k}\cdot\mathbf{T}}\psi_{\mathbf{k}}(\mathbf{r}),$$

which proves that this choice transforms $\psi_{\mathbf{k}}(\mathbf{r})$ into the Bloch form. Thus, $\psi_{\mathbf{k}}$ still satisfies the Schroedinger equation. We now require

$$\int d\mathbf{r}\psi^*_{\mathbf{k}}(\mathbf{r})\psi_{\mathbf{k}}(\mathbf{r}) = 1.$$

This yields

$$C^2\int d\mathbf{r}\sum_{i,j} e^{i\mathbf{k}\cdot(\mathbf{r}_i-\mathbf{r}_j)}\phi^*(\mathbf{r}-\mathbf{r}_j)\phi(\mathbf{r}-\mathbf{r}_i) = C^2\sum_{i,j} e^{i\mathbf{k}\cdot(\mathbf{r}_i-\mathbf{r}_j)}\underbrace{\int d\mathbf{r}\phi^*(\mathbf{r}-\mathbf{r}_j)\phi(\mathbf{r}-\mathbf{r}_i)}_{\delta_{ij}} = C^2N = 1,$$

where N is the number of atoms on the lattice. Thus, $C = 1/\sqrt{N}$ and we have the lattice wave function

$$\psi_{\mathbf{k}}(\mathbf{r}) = \frac{1}{\sqrt{N}}\sum_i e^{i\mathbf{k}\cdot\mathbf{r}_i}\phi(\mathbf{r}-\mathbf{r}_i),$$

which is equivalent to

$$|\mathbf{k}\rangle = \frac{1}{\sqrt{N}}\sum_i e^{i\mathbf{k}\cdot\mathbf{r}_i}|\phi_i\rangle.$$

Let H be the Hamiltonian of the system. We will now use it to calculate the electron energy, in the lattice potential, as a function of the momentum $\mathbf{k}$. Note that the momentum state $|\mathbf{k}\rangle$ depends on $\mathbf{r}$, but $\mathbf{r}$ is integrated out by forming any matrix element. We find that

$$\epsilon_{\mathbf{r}} = \langle\mathbf{k}|H|\mathbf{k}\rangle = \frac{1}{N}\sum_{i,j} e^{i\mathbf{k}\cdot(\mathbf{r}_i-\mathbf{r}_j)}\langle\phi_j|H|\phi_i\rangle.$$

We now consider a uniform lattice such that each lattice site j is equivalent. By denoting all neighboring sites of j by the vector $\boldsymbol{\rho}_i = \mathbf{r}_j - \mathbf{r}_i$, where i runs from 1 to N, the electron energy becomes

$$\begin{aligned}\epsilon_{\mathbf{k}} &= \frac{1}{N}\left(\sum_j 1\right)\sum_i e^{-i\mathbf{k}\cdot\boldsymbol{\rho}_i}\int d\mathbf{r}\phi^*(\mathbf{r}-\mathbf{r}_i-\boldsymbol{\rho}_i)H\phi(\mathbf{r}-\mathbf{r}_i)\\ &= \sum_i e^{-i\mathbf{k}\cdot\boldsymbol{\rho}_i}\int d\mathbf{r}\phi^*(\mathbf{r}-\boldsymbol{\rho}_i)H\phi(\mathbf{r}).\end{aligned}$$

Since the interactions between the atoms are small we let $\langle\phi_j|H|\phi_i\rangle \neq 0$ only for nearest neighbors i, j and for $i = j$. Obviously $\boldsymbol{\rho}_i = 0$ for $i = j$. Furthermore, we only need to consider $\boldsymbol{\rho}_m$ where m

runs over nearest neighbors. We find that

$$\begin{aligned}\epsilon_{\mathbf{k}} &= \int d\mathbf{r}\phi^*(\mathbf{r})H\phi(\mathbf{r}) + \sum_m e^{-i\mathbf{k}\cdot\boldsymbol{\rho}_m} \int d\mathbf{r}\phi^*(\mathbf{r}-\boldsymbol{\rho}_m)H\phi(\mathbf{r}) \\ &= \int d\mathbf{r}\phi^*(\mathbf{r})H\phi(\mathbf{r}) + \left[\int d\mathbf{r}\phi^*(\mathbf{r}-\boldsymbol{\rho})H\phi(\mathbf{r})\right] \sum_m e^{-i\mathbf{k}\cdot\boldsymbol{\rho}_m} \\ &\equiv -\alpha - \gamma \sum_m e^{-i\mathbf{k}\cdot\boldsymbol{\rho}_m}.\end{aligned}$$

In the next last step we used that $\phi(\mathbf{r}-\boldsymbol{\rho}_{m_1}) = \phi(\mathbf{r}-\boldsymbol{\rho}_{m_2})$ for $m_1 \neq m_2$, such that we could replace $\boldsymbol{\rho}_m$ with $\boldsymbol{\rho}$ in the integration.

In order to plot a Fermi-surface we set $\epsilon_{\mathbf{k}}$ to a constant value and search for the corresponding values of $\mathbf{k}$. We can also find the effective mass of the electrons by Taylor expanding $\epsilon_{\mathbf{k}}$ and note the coefficient of the $\mathbf{k}^2$ term for comparison with $\mathbf{k}^2/(2m)$.

A.4.5 Calculating energy bands - Wigner-Seitz method

Any Bloch function will satisfy the Schroedinger equation:

$$\left[-\frac{\hbar^2\nabla^2}{2m} + U(\mathbf{r})\right] e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) = \epsilon_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}). \tag{A.7}$$

We perform the differentiations on the exponential:

$$\begin{aligned}\left[-\frac{\hbar^2\nabla^2}{2m} + U(\mathbf{r})\right] e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) &= -\frac{\hbar^2\nabla}{2m} \cdot \left[i\mathbf{k}e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) + e^{i\mathbf{k}\cdot\mathbf{r}} \nabla u_{\mathbf{k}}(\mathbf{r})\right] + U(\mathbf{r}) e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) \\ &= -\frac{\hbar^2}{2m}\left[-\mathbf{k}^2 e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) + 2i\mathbf{k}e^{i\mathbf{k}\cdot\mathbf{r}} \nabla u_{\mathbf{k}}(\mathbf{r}) + e^{i\mathbf{k}\cdot\mathbf{r}} \nabla^2 u_{\mathbf{k}}(\mathbf{r})\right] \\ &+ U(\mathbf{r}) e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) \\ &= e^{i\mathbf{k}\cdot r}\left\{\frac{\hbar^2}{2m}\left[\mathbf{k}^2 - 2\mathbf{k}i\nabla + (i\nabla)^2\right] u_{\mathbf{k}}(\mathbf{r}) + U(\mathbf{r}) u_{\mathbf{k}}(\mathbf{r})\right\} \\ &= e^{i\mathbf{k}\cdot r}\left\{\frac{\hbar^2}{2m}(\mathbf{k}-i\nabla)^2 + U(\mathbf{r})\right\} u_{\mathbf{k}}(\mathbf{r}) \\ &= \epsilon_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}),\end{aligned}$$

where in the last step equation (A.7) was used. The exponential factors cancel on each side of the equality, leading to

$$\left[\frac{\hbar^2}{2m}(\mathbf{k}-i\nabla)^2 + U(\mathbf{r})\right] u_{\mathbf{k}}(\mathbf{r}) = \epsilon_{\mathbf{k}}(\mathbf{r}). \tag{A.8}$$

We can set $\mathbf{k} = 0$ and denote the solution as $u_0(\mathbf{r})$. From the u_0 solution we can construct the solution

$$\psi_{\mathbf{k}}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) \equiv |\mathbf{k}\rangle$$

that includes a dependence on $\mathbf{k}$ and that is of the Bloch form. Note that this is only an approximate solution to equation (A.8). We will now calculate the energy expectation value of $\psi_{\mathbf{k}}$ in the case that $\mathbf{k}\cdot\mathbf{p} = 0$, where $\mathbf{p} \equiv -i\nabla$. We start by noting that

$$\nabla^2\psi_{\mathbf{k}} = \nabla\left\{i\mathbf{k}\psi_{\mathbf{k}} + e^{i\mathbf{k}\cdot\mathbf{r}}\nabla u_{\mathbf{k}}\right\} = -\mathbf{k}^2\psi_{\mathbf{k}} + 2e^{i\mathbf{k}\cdot\mathbf{r}}\mathbf{k}\cdot i\nabla u_{\mathbf{k}} + e^{i\mathbf{k}\cdot\mathbf{r}}\nabla^2 u_{\mathbf{k}} = -\mathbf{k}^2\psi_{\mathbf{k}} + e^{i\mathbf{k}\cdot\mathbf{r}}\nabla^2 u_{\mathbf{k}},$$

where the middle term vanish in the last step due to $\mathbf{k} \cdot \mathbf{p} = 0$. Using this, we find that

$$\begin{aligned}
\epsilon_{\mathbf{k}} = \langle \mathbf{k}|H|\mathbf{k}\rangle &= \int d\mathbf{r} e^{-i\mathbf{k}\cdot\mathbf{r}} u^*_{\mathbf{k}}(\mathbf{r}) \left\{ \frac{\hbar^2}{2m}(\mathbf{k}^2 - \nabla^2) + U \right\} \left(e^{i\mathbf{k}\cdot\mathbf{r}} u_{\mathbf{k}}(\mathbf{r}) \right) \\
&= \int d\mathbf{r} u^*_{\mathbf{k}}(\mathbf{r}) \left\{ \frac{\hbar^2}{2m}(\mathbf{k}^2 - \nabla^2) + U \right\} u_{\mathbf{k}}(\mathbf{r}) + \int d\mathbf{r} u^*_{\mathbf{k}}(\mathbf{r}) \left\{ \frac{\hbar^2}{2m}(\mathbf{k}^2) \right\} u_{\mathbf{k}}(\mathbf{r}) \\
&= \epsilon_0 \int d\mathbf{r} u^*_{\mathbf{k}}(\mathbf{r}) u_{\mathbf{k}}(\mathbf{r}) + \frac{\mathbf{k}^2\hbar^2}{2m} \int d\mathbf{r} u^*_{\mathbf{k}}(\mathbf{r}) u_{\mathbf{k}}(\mathbf{r}) = \epsilon_0 + \frac{\mathbf{k}^2\hbar^2}{2m}.
\end{aligned}$$

A.4.6 Calculating energy bands - pseudopotential method

Between lattice sites on a crystal the conduction electron wave functions behave nearly as electrons in a weak Coulomb field, shielded from the ion cores by the orbital electrons of the ion. The behavior of the wave function is only complicated close to the ion cores. It turns out that the potential close to an ion core is close to zero. A good approximation of the lattice potential can be constructed by using an effective potential that is nearly zero near the core, but results in a wave function that behaves as electrons in a weak Coulomb field outside the core. This approximation is referred to as the *pseudo potential method*. The *Empty Core Model* (ECM), utilize a potential of the form

$$U(\mathbf{r}) = \begin{cases} 0 & , r < R_e \\ -e^2/r & , r > R_e \end{cases},$$

for some radius R_e.

A.4.7 Electron orbits

An electron in a magnetic field $\mathbf{B}$ will follow a circular trajectory. When a magnetic field is applied to a material the electrons are subjected to an electric field produced by the crystal ions, as well as the external magnetic field. Electrons need to be in a conduction band in order to move. Furthermore, the movement must be along the lowest energy trajectory. This trajectory will thus be along a surface of constant energy (in $\mathbf{k}$ space) that is not filled by other electrons. This surface is exactly the Fermi surface.

We assume a magnetic field pointing out of the plane. If the electron moves along a closed trajectory on a Fermi surface that is filled inside the trajectory, then the motion will be anticlockwise. This is referred to as an *electronlike orbit*.

If the Fermi surface is filled outside the closed trajectory, the motion will be clockwise. The trajectory is referred to as a *holelike orbit* because it moves as a $+e$ charge in a magnetic field.

It can also be that the electron moves from lattice cell to lattice cell in a periodic fashion along the Fermi surface. This is referred to as an *open orbit*.

APPENDIX B

The Brillouin zone of Graphene

The Brillouin zone for Graphene is constructed from the reciprocal lattice vectors in equation (3.2) and equation (3.3). The construction is shown in figure B.1.

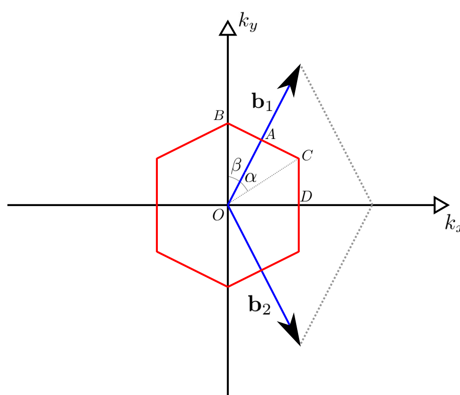


Figure B.1: The Brillouin zone of single layer Graphene.

Looking at the figure one finds that

$$K_x := OD = \frac{1}{2}|\mathbf{b}_1 + \mathbf{b}_2| = \frac{2\pi}{3a}.$$

Furthermore,

$$OA = \frac{1}{2}|\mathbf{b}_1| = \frac{2\pi}{3a},$$

which implies that $OA = OD$ and $\alpha = 0.5\angle(\mathbf{b}_1, \hat{k}_x)$. The angle between $\mathbf{b}_1$ and $\hat{k}_x$ can be found by

$$\cos[\angle(\mathbf{b}_1, \hat{k}_x)] = \frac{\mathbf{b}_1 \cdot \hat{k}_x}{|\mathbf{b}_1|} = \frac{1}{2} \Rightarrow \angle(\mathbf{b}_1, \hat{k}_x) = \frac{\pi}{3},$$

which gives

$$\alpha = \frac{\pi}{6}.$$

Thus,

$$K_y := DC = K_x \tan\alpha = \frac{2\pi}{3\sqrt{3}a},$$

and the first corner of the Brillouin zone (C) is found to be

$$\mathbf{K} := (K_x, K_y) = \left(\frac{2\pi}{3a}, \frac{2\pi}{3\sqrt{3}a}\right).$$

The second corner of the Brillouin zone (B) can be found by noting that

$$\beta = \frac{\pi}{2} - \angle(\mathbf{b}_1, \hat{k}_x) = \frac{\pi}{6},$$

which means that

$$K_y^{''} := OB = \frac{OA}{\cos\beta} = \frac{4\pi}{3\sqrt{3}a},$$

and thus

$$\mathbf{K}^{'} = \left(0, \frac{4\pi}{3\sqrt{3}a}\right).$$

The remaining points of the Brillouin zone are found by symmetry. The Dirac points are the six points where the two bands of the dispersion meet, as can be seen from figure 3.2. The Dirac points can be found by looking for the minimum of the electron band (dispersion with positive energy) or by finding the maximums of the hole band (dispersion with negative energy). It turns out that the corners of the Brillouin zone exactly corresponds to the six Dirac points. This is shown in appendix C.

APPENDIX C

The Dirac points

In the following the maximum and minimum points of the single layer dispersion relation will be found. These points are in this case Dirac points. To find these points, calculate the derivative

$$\frac{\partial \epsilon_{\mathbf{k}}}{\partial k_i} = \frac{\partial f(\mathbf{k})}{\partial k_i} \left[-t' \pm t \frac{1}{2\sqrt{f(\mathbf{k})+3}} \right],$$

where $\epsilon_{\mathbf{k}}$ is the single layer Graphene dispersion relation. Thus, the maximum and minimum of the dispersion is found by requiring that

$$\frac{\partial f(\mathbf{k})}{\partial k_i} = 0 \text{ for } i \in x, y.$$

Then,

$$\frac{\partial f(\mathbf{k})}{\partial k_x} = -6a \sin\left(\frac{3a}{2}k_x\right) \cos\left(\frac{\sqrt{3}a}{2}k_y\right) = 0, \tag{C.1}$$

$$\begin{aligned} \frac{\partial f(\mathbf{k})}{\partial k_y} &= -2\sqrt{3}a \left[\sin\left(\sqrt{3}ak_y\right) + \cos\left(\frac{3a}{2}k_x\right) \sin\left(\frac{\sqrt{3}a}{2}k_y\right) \right] \\ &= -2\sqrt{3}a \sin\left(\frac{\sqrt{3}a}{2}k_y\right) \left[2\cos\left(\frac{\sqrt{3}a}{2}k_y\right) + \cos\left(\frac{3a}{2}k_x\right) \right] = 0, \end{aligned} \tag{C.2}$$

where it was used that

$$\sin\left(2\frac{x}{2}\right) = 2\sin\left(\frac{x}{2}\right)\cos\left(\frac{x}{2}\right).$$

From equation (C.1), either

$$k_x = \frac{2\pi}{3a}n \ , \ n \in \mathbb{Z}, \tag{C.3}$$

or

$$k_y = \frac{2\pi}{\sqrt{3}a}\left[\frac{1}{2} + n'\right] \ \ n' \in \mathbb{Z}. \tag{C.4}$$

If equation (C.3) holds, then equation (C.2) implies that

$$\sin\left(\frac{\sqrt{3}a}{2}k_y\right) = 0 \Rightarrow k_y = \frac{2\pi}{\sqrt{3}a}n',$$

or

$$2\cos\left(\frac{\sqrt{3}a}{2}k_y\right)+\cos\left(\frac{3a}{2}k_x\right)=2\cos\left(\frac{\sqrt{3}a}{2}k_y\right)+\cos(\pi n)=0.$$

If n is odd, then $\cos(n\pi)=-1$ and

$$k_y=\frac{2\pi}{\sqrt{3}a}\left[\frac{1}{3}+2n'\right] \text{ or } k_y=\frac{2\pi}{\sqrt{3}a}\left[\frac{5}{3}+2n'\right].$$

If n is even, then $\cos(n\pi)=+1$ and

$$k_y=\frac{4\pi}{\sqrt{3}a}\left[\frac{1}{3}+n'\right] \text{ or } k_y=\frac{4\pi}{\sqrt{3}a}\left[\frac{2}{3}+n'\right].$$

If it is no longer require that equation (C.3) holds, but rather that equation (C.4) holds, then equation (C.2) implies that

$$2\cos\left(\frac{\sqrt{3}a}{2}k_y\right)+\cos\left(\frac{3a}{2}k_x\right)=\cos\left(\frac{3a}{2}k_x\right)=0$$

must be satisfied and thus,

$$k_x=\frac{2\pi}{3a}\left[\frac{1}{2}+n\right].$$

Table C.1 lists the resulting coordinates of the maximum and minimum of the dispersion.

k_x	k_y	*Constraints*
$\frac{2\pi}{3a}n$	$\frac{2\pi}{\sqrt{3}a}n'$	$n, n' \in \mathbb{Z}$
$\frac{2\pi}{3a}n$	$\frac{2\pi}{\sqrt{3}a}\left[\frac{1}{3}+2n'\right]$ or $\frac{2\pi}{\sqrt{3}a}\left[\frac{5}{3}+2n'\right]$	$n' \in \mathbb{Z}$ and n is odd
$\frac{2\pi}{3a}n$	$\frac{4\pi}{\sqrt{3}a}\left[\frac{1}{3}+n'\right]$ or $\frac{4\pi}{\sqrt{3}a}\left[\frac{2}{3}+n'\right]$	$n' \in \mathbb{Z}$ and n is even
$\frac{2\pi}{3a}\left[\frac{1}{2}+n\right]$	$\frac{2\pi}{\sqrt{3}a}\left[\frac{1}{2}+n'\right]$	$n, n' \in \mathbb{Z}$

Table C.1: Coordinates of maximum and minimum of the single layer Graphene dispersion.

Next, insert the momentum coordinates from table C.1 into the expression for $\epsilon_{\mathbf{k}}$. Thus, it is found that

$$f\left(\frac{2\pi}{3a}n,\frac{2\pi}{\sqrt{3}a}n'\right)=2\cos(2\pi n')+4\cos(\pi n)\cos(\pi n')=\begin{cases}+6\ ;\ n,n' \text{ both odd or both even}\\ -2\ ;\ n,n' \text{ one even and one odd}\end{cases},$$

and for both n, n' even or both odd it is found that $\epsilon_{\mathbf{k}}=-6t'\pm 3t$. For n even and n' odd or vice versa $\epsilon_{\mathbf{k}}=2t'\pm t$. Now look at

$$f\left(\frac{2\pi}{3a}n,\frac{2\pi}{\sqrt{3}a}\left[\frac{1}{3}+2n'\right]\right)=2\cos\left(\frac{2\pi}{3}+4\pi n'\right)+4\cos(\pi n)\cos\left(\frac{\pi}{3}+2\pi n'\right)=-3,$$

where it was used that n is odd for this momentum coordinate. This gives $\epsilon_{\mathbf{k}}=3t'$. Similarly,

$$f\left(\frac{2\pi}{3a}n,\frac{2\pi}{\sqrt{3}a}\left[\frac{5}{3}+2n'\right]\right)=2\cos\left(\frac{10\pi}{3}+4\pi n'\right)+4\cos(\pi n)\cos\left(\frac{5\pi}{3}+2\pi n'\right)=-3,$$

also leading to $\epsilon_{\mathbf{k}}=3t'$. Following this, calculate the next two energy values corresponding to the next two coordinates in table C.1:

$$f\left(\frac{2\pi}{3a}n,\frac{4\pi}{\sqrt{3}a}\left[\frac{1}{3}+n'\right]\right)=2\cos\left(\frac{4\pi}{3}+4\pi n'\right)+4\cos(\pi n)\cos\left(\frac{2\pi}{3}+2\pi n'\right)=-3,$$

$$f\left(\frac{2\pi}{3a}n, \frac{4\pi}{\sqrt{3}a}\left[\frac{2}{3}+n'\right]\right) = 2\cos\left(\frac{8\pi}{3}+4\pi n'\right) + 4\cos(\pi n)\cos\left(\frac{4\pi}{3}+2\pi n'\right) = -3,$$

where it can be noted that, in this case, n must be even. Again it is found that $\epsilon_{\mathbf{k}} = 3t'$. For the last coordinate pair in table C.1

$$f\left(\frac{2\pi}{3a}\left[\frac{1}{2}+n\right], \frac{2\pi}{\sqrt{3}a}\left[\frac{1}{2}+n'\right]\right) = 2\cos\left(\pi+2\pi n'\right) + 4\cos\left(\frac{\pi}{2}+\pi n\right)\cos\left(\frac{\pi}{2}+\pi n'\right) = -2,$$

which leads to $\epsilon_{\mathbf{k}} = 2t' \pm t$.

Observe the momentum coordinates (k_x, k_y) that lead to $\epsilon_{\mathbf{k}} = 3t'$. Here, both the branches of the dispersion $\epsilon_{\mathbf{k}}$ meet at the same energy. These are the Dirac points and can be observed to coincide with where the Dirac cones of figure 3.2 meet. Also notice that the Dirac points emerging from n, n' being in the set $n, n' \in \{0, \pm 1\}$ while $k_x, k_y \in (-4, 4)$ correspond exactly with the corners of the Brillouin zone shown in figure B.1.

APPENDIX D

Moving the singularities

Starting with the expression

$$\frac{\Delta E_{ex}}{A_c} = \sum_{i,j=1,\sigma}^{4} \sum_{a} \int_0^1 dk \int_0^1 dk' \int_0^{2\pi} d\theta F_{ij}(k,k',\theta) V_s(k,k',\theta) \times \left[n^{FM}_{\sigma,a,i}(k\Lambda) n^{FM}_{\sigma,a,j}(k'\Lambda) - n^{PM}_{\sigma,a,i}(k\Lambda) n^{PM}_{\sigma,a,j}(k'\Lambda) \right]$$

and inserting the Fermi occupation functions corresponding to two types of carriers leads to

$$\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta \Big[& \int_0^{Q_\uparrow} dk \int_0^{Q_\uparrow} dk' F_{11}(k,k',\theta) V_s(k,k',\theta) \\
& + \int_0^{Q_\uparrow} dk \int_0^1 dk' 2F_{12}(k,k',\theta) V_s(k,k',\theta) + \int_0^1 dk \int_0^1 dk' F_{22}(k,k',\theta) V_s(k,k',\theta) \\
& + \int_0^1 dk \int_0^1 dk' F_{22}(k,k',\theta) V_s(k,k',\theta) - \int_0^{Q_\downarrow} dk \int_0^1 dk' 2F_{22}(k,k',\theta) V_s(k,k',\theta) \\
& + \int_0^{Q_\downarrow} dk \int_0^{Q_\downarrow} dk' F_{22}(k,k',\theta) V_s(k,k',\theta) - \int_0^{Q_d} dk \int_0^{Q_d} dk' 2F_{11}(k,k',\theta) V_s(k,k',\theta) \\
& - \int_0^{Q_d} dk \int_0^1 dk' 4F_{12}(k,k',\theta) V_s(k,k',\theta) - \int_0^1 dk \int_0^1 dk' 2F_{22}(k,k',\theta) V_s(k,k',\theta) \\
& - \int_0^1 dk \int_0^1 dk' 2F_{44}(k,k',\theta) V_s(k,k',\theta) - \int_0^1 dk \int_0^1 dk' 4F_{24}(k,k',\theta) V_s(k,k',\theta) \\
& - \int_0^{Q_d} dk \int_0^1 dk' 4F_{14}(k,k',\theta) V_s(k,k',\theta) + \int_0^1 dk \int_0^1 dk' 2F_{44}(k,k',\theta) V_s(k,k',\theta) \\
& + \int_0^1 dk \int_0^1 dk' 2F_{24}(k,k',\theta) V_s(k,k',\theta) + \int_0^{Q_\uparrow} dk \int_0^1 dk' 2F_{14}(k,k',\theta) V_s(k,k',\theta) \\
& + \int_0^1 dk \int_0^1 dk' 2F_{24}(k,k',\theta) V_s(k,k',\theta) - \int_0^1 dk \int_0^{Q_\downarrow} dk' 2F_{24}(k,k',\theta) V_s(k,k',\theta) \Big].
\end{aligned}$$

Simplifications were made by using

$$\begin{aligned}\int_0^Q dk\int_0^1 dk' F_{ij}(k,k',\theta) &\sim \int_0^Q dk\int_0^1 dk'\chi^\alpha_{ij}(k,k',\theta)\chi^\alpha_{ji}(k',k,\theta)\\ &= \int_0^Q dk'\int_0^1 dk\chi^\alpha_{ij}(k',k,\theta)\chi^\alpha_{ji}(k,k',\theta)\\ &= \int_0^1 dk\int_0^Q dk'\chi^\alpha_{ji}(k,k',\theta)\chi^\alpha_{ij}(k',k,\theta),\end{aligned}$$

which leads to

$$\int_0^Q dk\int_0^1 dk' F_{ij}(k,k',\theta) = \int_0^1 dk\int_0^Q dk' F_{ji}(k,k',\theta).$$

The factor of two comes from summing over the two K-points (i.e. sum over a). Some of the terms in $\Delta Eex/A_c$ cancel. Thus,

$$\begin{aligned}\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta\Big[&\int_0^{Q_\uparrow} dk\int_0^{Q_\uparrow} dk' F_{11}(k,k',\theta)V_s(k,k',\theta) + \int_0^{Q_\uparrow} dk\int_0^1 dk' 2F_{12}(k,k',\theta)V_s(k,k',\theta)\\ &- \int_0^{Q_\downarrow} dk\int_0^1 dk' 2F_{22}(k,k',\theta)V_s(k,k',\theta) + \int_0^{Q_\downarrow} dk\int_0^{Q_\downarrow} dk' F_{22}(k,k',\theta)V_s(k,k',\theta)\\ &- \int_0^{Q_d} dk\int_0^{Q_d} dk' 2F_{11}(k,k',\theta)V_s(k,k',\theta) - \int_0^{Q_d} dk\int_0^1 dk' 4F_{12}(k,k',\theta)V_s(k,k',\theta)\\ &+ \int_0^{Q_\uparrow} dk\int_0^1 dk' 2F_{14}(k,k',\theta)V_s(k,k',\theta) - \int_0^{Q_d} dk\int_0^1 dk' 4F_{14}(k,k',\theta)V_s(k,k',\theta)\\ &- \int_0^{Q_\downarrow} dk\int_0^1 dk' 2F_{24}(k,k',\theta)V_s(k,k',\theta)\Big].\end{aligned}$$

In order to perform a Duffy transformation the integral boundaries must run from 0 to 1. This is achieve by a change of variables of the form $k = \tilde{k}Q$. Performing the change of variables leads to

$$\begin{aligned}\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta\int_0^1 dk\int_0^1 dk'\big[&Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow k',\theta)V_s(k,k',\theta)\\ &+ 2Q_\uparrow F_{12}(Q_\uparrow k,k',\theta)V_s(Q_\uparrow k,k',\theta) - 2Q_\downarrow F_{22}(Q_\downarrow k,k',\theta)V_s(Q_\downarrow k,k',\theta)\\ &+ Q_\downarrow F_{22}(Q_\downarrow k,Q_\downarrow k',\theta)V_s(k,k',\theta) - 2Q_d F_{11}(Q_d k,Q_d k',\theta)V_s(k,k',\theta)\\ &- 4Q_d F_{12}(Q_d k,k',\theta)V_s(Q_d k,k',\theta) + 2Q_\uparrow F_{14}(Q_\uparrow k,k',\theta)V_s(Q_\uparrow k,k',\theta)\\ &- 2Q_\downarrow F_{24}(Q_\downarrow k,k',\theta)V_s(Q_\downarrow k,k',\theta) - 4Q_d F_{14}(Q_d k,k',\theta)V_s(Q_d k,k',\theta)\big],\end{aligned}$$

where the equality

$$V_s(Qk,Qk',\theta) = \frac{1}{Q}V_s(k,k',\theta)$$

was used to simplify the expression. Applying a Duffy transformation yields

$$\begin{aligned}\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta\int_0^1 dk\int_0^1 dk'\big[&Q_\uparrow\{F_{11}(Q_\uparrow k,Q_\uparrow kk',\theta) + F_{11}(Q_\uparrow kk',Q_\uparrow k,\theta)\}V_s(k',\theta)\\ &+ 2Q_\uparrow\{F_{12}(Q_\uparrow k,kk',\theta)V_s^1(k',\theta,\uparrow) + F_{12}(Q_\uparrow kk',k,\theta)V_s^2(k',\theta,\uparrow)\}\\ &- 2Q_\downarrow\{F_{22}(Q_\downarrow k,kk',\theta)V_s^1(k',\theta,\downarrow) + F_{22}(Q_\downarrow kk',k,\theta)V_s^2(k',\theta,\downarrow)\}\\ &+ Q_\downarrow\{F_{22}(Q_\downarrow k,Q_\downarrow kk',\theta) + F_{22}(Q_\downarrow kk',Q_\downarrow k,\theta)\}V_s(k',\theta)\\ &- 2Q_d\{F_{11}(Q_d k,Q_d kk',\theta) + F_{11}(Q_d kk',Q_d k,\theta)\}V_s(k',\theta)\\ &- 4Q_d\{F_{12}(Q_d k,kk',\theta)V_s^1(k',\theta,d) + F_{12}(Q_d kk',k,\theta)V_s^2(k',\theta,d)\}\\ &+ 2Q_\uparrow\{F_{14}(Q_\uparrow k,kk',\theta)V_s^1(k',\theta,\uparrow) + F_{14}(Q_\uparrow kk',k,\theta)V_s^2(k',\theta,\uparrow)\}\\ &- 2Q_\downarrow\{F_{24}(Q_\downarrow k,kk',\theta)V_s^1(k',\theta,\downarrow) + F_{24}(Q_\downarrow kk',k,\theta)V_s^2(k',\theta,\downarrow)\}\\ &- 4Q_d\{F_{14}(Q_d k,kk',\theta)V_s^1(k',\theta,d) + F_{14}(Q_d kk',k,\theta)V_s^2(k',\theta,d)\}\big],\end{aligned}$$

where

$$
\begin{aligned}
V_s(k',\theta) &:= \frac{1}{\sqrt{k'^2 - 2k'\cos\theta + 1}}, \\
V_s^1(k',\theta,i) &:= \frac{1}{\sqrt{k'^2 - 2k'Q_i\cos\theta + Q_i^2}} = \frac{1}{Q_i\sqrt{(k'/Q_i)^2 - 2k'/Q_i\cos\theta + 1}}, \\
V_s^2(k',\theta,i) &:= \frac{1}{\sqrt{k'^2Q_i^2 - 2k'Q_i\cos\theta + 1}}.
\end{aligned}
$$

Further simplifications can be made by noting that

$$
\begin{aligned}
\int_0^1 dk \int_0^1 dk' F_{ii}(x,y,\theta) &\sim \int_0^1 dk \int_0^1 dk' \chi_{ii}^{\alpha}(x,y,\theta)\chi_{ii}^{\alpha}(y,x,\theta) \\
&= \int_0^1 dk \int_0^1 dk' \chi_{ii}^{\alpha}(y,x,\theta)\chi_{ii}^{\alpha}(x,y,\theta),
\end{aligned}
$$

such that

$$
\int_0^1 dk \int_0^1 dk' F_{ii}(x,y,\theta)g(x,y) = \int_0^1 dk \int_0^1 dk' F_{ii}(y,x,\theta)g(x,y).
$$

Then,

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \big[& 2Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk',\theta)V_s(k',\theta) \\
&+ 2Q_\uparrow\{F_{12}(Q_\uparrow k, kk',\theta)V_s^1(k',\theta,\uparrow) + F_{12}(Q_\uparrow kk', k,\theta)V_s^2(k',\theta,\uparrow)\} \\
&- 2Q_\downarrow\{F_{22}(Q_\downarrow k, kk',\theta)V_s^1(k',\theta,\downarrow) + F_{22}(Q_\downarrow kk', k,\theta)V_s^2(k',\theta,\downarrow)\} \\
&+ 2Q_\downarrow F_{22}(Q_\downarrow k, Q_\downarrow kk',\theta)V_s(k',\theta) \\
&- 4Q_d F_{11}(Q_d k, Q_d kk',\theta)V_s(k',\theta) \\
&- 4Q_d\{F_{12}(Q_d k, kk',\theta)V_s^1(k',\theta,d) + F_{12}(Q_d kk', k,\theta)V_s^2(k',\theta,d)\} \\
&+ 2Q_\uparrow\{F_{14}(Q_\uparrow k, kk',\theta)V_s^1(k',\theta,\uparrow) + F_{14}(Q_\uparrow kk', k,\theta)V_s^2(k',\theta,\uparrow)\} \\
&- 2Q_\downarrow\{F_{24}(Q_\downarrow k, kk',\theta)V_s^1(k',\theta,\downarrow) + F_{24}(Q_\downarrow kk', k,\theta)V_s^2(k',\theta,\downarrow)\} \\
&- 4Q_d\{F_{14}(Q_d k, kk',\theta)V_s^1(k',\theta,d) + F_{14}(Q_d kk', k,\theta)V_s^2(k',\theta,d)\}\big].
\end{aligned}
$$

All singularities are now independent of k, but not all are located at $k' = 1$. In order to make numerical integration easier all singularities will be moved to $k' = 1$ through two types of change of variables. For terms involving V_s^1, let $\tilde{k}' = k'/Q_i$ and let $\tilde{k}' = k'Q_i$ for terms involving V_s^2.

This leads to

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} &= 2\int_0^{2\pi} d\theta \int_0^1 dk \Bigg[2Q_\uparrow \int_0^1 dk' F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) \\
&+ 2Q_\uparrow \Bigg\{ \int_0^{1/Q_\uparrow} dk' F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\uparrow} dk' \frac{1}{Q_\uparrow} F_{12}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
&- 2Q_\downarrow \Bigg\{ \int_0^{1/Q_\downarrow} dk' F_{22}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\downarrow} dk' \frac{1}{Q_\downarrow} F_{22}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
&+ 2Q_\downarrow \int_0^1 dk' F_{22}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) - 4Q_d \int_0^1 dk' F_{11}(Q_d k, Q_d kk', \theta) V_s(k', \theta) \\
&- 4Q_d \Bigg\{ \int_0^{1/Q_d} dk' F_{12}(Q_d k, Q_d kk', \theta) V_s(k', \theta) + \int_0^{Q_d} dk' \frac{1}{Q_d} F_{12}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
&+ 2Q_\uparrow \Bigg\{ \int_0^{1/Q_\uparrow} dk' F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\uparrow} dk' \frac{1}{Q_\uparrow} F_{14}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
&- 2Q_\downarrow \Bigg\{ \int_0^{1/Q_\downarrow} dk' F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\downarrow} dk' \frac{1}{Q_\downarrow} F_{24}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
&- 4Q_d \Bigg\{ \int_0^{1/Q_d} dk' F_{14}(Q_d k, Q_d kk', \theta) V_s(k', \theta) + \int_0^{Q_d} dk' \frac{1}{Q_d} F_{14}(kk', k, \theta) V_s(k', \theta) \Bigg\} \Bigg].
\end{aligned}
$$

All singularities of the integrand are now located at $k' = 1$. However, in order to perform numerical integrations, the singularities should only be positioned along the integration boundaries of the integration. Since $Q_i \leq 1$ then $1/Q_i \geq 1$, which means it is necessary to divide the integrals with boundary $1/Q_i$ into two integrals. Dividing the integrations leads to

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = &\int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \big\{ 4Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\uparrow F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) \\
&\quad - 8Q_d F_{11}(Q_d k, Q_d kk', \theta) - 8Q_d F_{12}(Q_d k, Q_d kk', \theta) \\
&\quad + 4Q_\uparrow F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) - 4Q_\downarrow F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
&\quad - 8Q_d F_{14}(Q_d k, Q_d kk', \theta) \big\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} dk' 4Q_\uparrow \left\{ F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) + F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} dk' 4Q_\downarrow \left\{ F_{22}(Q_\downarrow k, Q_\downarrow kk', \theta) + F_{24}(Q_\downarrow k, Q_\downarrow kk', \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} dk' 8Q_d \left\{ F_{12}(Q_d k, Q_d kk', \theta) + F_{14}(Q_d k, Q_d kk', \theta) \right\} V_s(k', \theta) \\
&+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\uparrow} dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\downarrow} dk' 4 \left\{ F_{22}(kk', k, \theta) + F_{24}(kk', k, \theta) \right\} V_s(k', \theta) \\
&- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_d} dk' 8 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta).
\end{aligned}
$$

This is the expression for the exchange energy based on a ferromagnetic state of two types of charge carriers that will be evaluated numerically.

There exists an analogous expression for the exchange energy based on a ferromagnetic state with one type of carrier that needs to be calculated. As before, the corresponding Fermi occupation

functions are inserted into equation (4.20). This yields

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} &= 2\int_0^{2\pi} d\theta \Big[\int_0^{Q_\uparrow} dk \int_0^{Q_\uparrow} dk' F_{11}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{Q_\uparrow} dk \int_0^{1} dk' 2F_{12}(k,k',\theta)V_s(k,k',\theta) + \int_0^{1} dk \int_0^{1} dk' F_{22}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{1} dk \int_0^{1} dk' F_{22}(k,k',\theta)V_s(k,k',\theta) + \int_0^{Q_\downarrow} dk \int_0^{1} dk' 2F_{12}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{Q_\downarrow} dk \int_0^{Q_\downarrow} dk' F_{11}(k,k',\theta)V_s(k,k',\theta) - \int_0^{Q_d} dk \int_0^{Q_d} dk' 2F_{11}(k,k',\theta)V_s(k,k',\theta) \\
&- \int_0^{Q_d} dk \int_0^{1} dk' 4F_{12}(k,k',\theta)V_s(k,k',\theta) - \int_0^{1} dk \int_0^{1} dk' 2F_{22}(k,k',\theta)V_s(k,k',\theta) \\
&- \int_0^{1} dk \int_0^{1} dk' 2F_{44}(k,k',\theta)V_s(k,k',\theta) - \int_0^{1} dk \int_0^{1} dk' 4F_{24}(k,k',\theta)V_s(k,k',\theta) \\
&- \int_0^{Q_d} dk \int_0^{1} dk' 4F_{14}(k,k',\theta)V_s(k,k',\theta) + \int_0^{1} dk \int_0^{1} dk' 2F_{44}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{1} dk \int_0^{1} dk' 4F_{24}(k,k',\theta)V_s(k,k',\theta) + \int_0^{Q_\uparrow} dk \int_0^{1} dk' 2F_{14}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{Q_\downarrow} dk \int_0^{1} dk' 2F_{14}(k,k',\theta)V_s(k,k',\theta) \Big].
\end{aligned}
$$

As in the previous case, some terms cancel and

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} &= 2\int_0^{2\pi} d\theta \Big[\int_0^{Q_\uparrow} dk \int_0^{Q_\uparrow} dk' F_{11}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{Q_\uparrow} dk \int_0^{1} dk' 2F_{12}(k,k',\theta)V_s(k,k',\theta) + \int_0^{Q_\downarrow} dk \int_0^{1} dk' 2F_{12}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{Q_\downarrow} dk \int_0^{Q_\downarrow} dk' F_{11}(k,k',\theta)V_s(k,k',\theta) - \int_0^{Q_d} dk \int_0^{Q_d} dk' 2F_{11}(k,k',\theta)V_s(k,k',\theta) \\
&- \int_0^{Q_d} dk \int_0^{1} dk' 4F_{12}(k,k',\theta)V_s(k,k',\theta) + \int_0^{Q_\uparrow} dk \int_0^{1} dk' 2F_{14}(k,k',\theta)V_s(k,k',\theta) \\
&+ \int_0^{Q_\downarrow} dk \int_0^{1} dk' 2F_{14}(k,k',\theta)V_s(k,k',\theta) - \int_0^{Q_d} dk \int_0^{1} dk' 4F_{14}(k,k',\theta)V_s(k,k',\theta) \Big].
\end{aligned}
$$

Using change of variables to move the integration boundaries to the range $[0,1]$ leads to

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} &= 2\int_0^{2\pi} d\theta \int_0^{1} dk \int_0^{1} dk' \big[Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow k',\theta)V_s(k,k',\theta) \\
&+ 2Q_\uparrow F_{12}(Q_\uparrow k,k',\theta)V_s(Q_\uparrow k,k',\theta) + 2Q_\downarrow F_{12}(Q_\downarrow k,k',\theta)V_s(Q_\downarrow k,k',\theta) \\
&+ Q_\downarrow F_{11}(Q_\downarrow k, Q_\downarrow k',\theta)V_s(k,k',\theta) - 2Q_d F_{11}(Q_d k, Q_d k',\theta)V_s(k,k',\theta) \\
&- 4Q_d F_{12}(Q_d k,k',\theta)V_s(Q_d k,k',\theta) + 2Q_\uparrow F_{14}(Q_\uparrow k,k',\theta)V_s(Q_\uparrow k,k',\theta) \\
&+ 2Q_\downarrow F_{14}(Q_\downarrow k,k',\theta)V_s(Q_\downarrow k,k',\theta) - 4Q_d F_{14}(Q_d k,k',\theta)V_s(Q_d k,k',\theta) \big].
\end{aligned}
$$

By applying a Duffy transformation the exchange energy becomes

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \big[& 2Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) \\
& + 2Q_\uparrow \{F_{12}(Q_\uparrow k, kk', \theta) V_s^1(k', \theta, \uparrow) + F_{12}(Q_\uparrow kk', k, \theta) V_s^2(k', \theta, \uparrow)\} \\
& + 2Q_\downarrow \{F_{12}(Q_\downarrow k, kk', \theta) V_s^1(k', \theta, \downarrow) + F_{12}(Q_\downarrow kk', k, \theta) V_s^2(k', \theta, \downarrow)\} \\
& + 2Q_\downarrow F_{11}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) \\
& - 4Q_d F_{11}(Q_d k, Q_d kk', \theta) V_s(k', \theta) \\
& - 4Q_d \{F_{12}(Q_d k, kk', \theta) V_s^1(k', \theta, d) + F_{12}(Q_d kk', k, \theta) V_s^2(k', \theta, d)\} \\
& + 2Q_\uparrow \{F_{14}(Q_\uparrow k, kk', \theta) V_s^1(k', \theta, \uparrow) + F_{14}(Q_\uparrow kk', k, \theta) V_s^2(k', \theta, \uparrow)\} \\
& + 2Q_\downarrow \{F_{14}(Q_\downarrow k, kk', \theta) V_s^1(k', \theta, \downarrow) + F_{14}(Q_\downarrow kk', k, \theta) V_s^2(k', \theta, \downarrow)\} \\
& - 4Q_d \{F_{14}(Q_d k, kk', \theta) V_s^1(k', \theta, d) + F_{14}(Q_d kk', k, \theta) V_s^2(k', \theta, d)\} \big].
\end{aligned}
$$

Moving all singularities to $k' = 1$ results in

$$
\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = 2\int_0^{2\pi} d\theta \int_0^1 dk \Bigg[& 2Q_\uparrow \int_0^1 dk' F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) \\
& + 2Q_\uparrow \Bigg\{ \int_0^{1/Q_\uparrow} dk' F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\uparrow} dk' \frac{1}{Q_\uparrow} F_{12}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
& + 2Q_\downarrow \Bigg\{ \int_0^{1/Q_\downarrow} dk' F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\downarrow} dk' \frac{1}{Q_\downarrow} F_{12}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
& + 2Q_\downarrow \int_0^1 dk' F_{11}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) - 4Q_d \int_0^1 dk' F_{11}(Q_d k, Q_d kk', \theta) V_s(k', \theta) \\
& - 4Q_d \Bigg\{ \int_0^{1/Q_d} dk' F_{12}(Q_d k, Q_d kk', \theta) V_s(k', \theta) + \int_0^{Q_d} dk' \frac{1}{Q_d} F_{12}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
& + 2Q_\uparrow \Bigg\{ \int_0^{1/Q_\uparrow} dk' F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\uparrow} dk' \frac{1}{Q_\uparrow} F_{14}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
& + 2Q_\downarrow \Bigg\{ \int_0^{1/Q_\downarrow} dk' F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta) V_s(k', \theta) + \int_0^{Q_\downarrow} dk' \frac{1}{Q_\downarrow} F_{14}(kk', k, \theta) V_s(k', \theta) \Bigg\} \\
& - 4Q_d \Bigg\{ \int_0^{1/Q_d} dk' F_{14}(Q_d k, Q_d kk', \theta) V_s(k', \theta) + \int_0^{Q_d} dk' \frac{1}{Q_d} F_{14}(kk', k, \theta) V_s(k', \theta) \Bigg\} \Bigg]
\end{aligned}
$$

and after dividing the integrations at $1/Q_i$ the expression for the exchange energy for the ferromagnetic state with one carrier becomes

$$\begin{aligned}
\frac{\Delta E_{ex}}{Ac} = \int_0^{2\pi} d\theta \int_0^1 dk \int_0^1 dk' \Big\{ & 4Q_\uparrow F_{11}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\uparrow F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) \\
& + 4Q_\downarrow F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) + 4Q_\downarrow F_{11}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
& - 8Q_d F_{11}(Q_d k, Q_d kk', \theta) - 8Q_d F_{12}(Q_d k, Q_d kk', \theta) \\
& + 4Q_\uparrow F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) + 4Q_\downarrow F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta) \\
& - 8Q_d F_{14}(Q_d k, Q_d kk', \theta) \Big\} V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\uparrow} & dk' 4Q_\uparrow \left\{ F_{12}(Q_\uparrow k, Q_\uparrow kk', \theta) + F_{14}(Q_\uparrow k, Q_\uparrow kk', \theta) \right\} V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_\downarrow} & dk' 4Q_\downarrow \left\{ F_{12}(Q_\downarrow k, Q_\downarrow kk', \theta) + F_{14}(Q_\downarrow k, Q_\downarrow kk', \theta) \right\} V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_1^{1/Q_d} & dk' 8Q_d \left\{ F_{12}(Q_d k, Q_d kk', \theta) + F_{14}(Q_d k, Q_d kk', \theta) \right\} V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\uparrow} & dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta) \\
+ \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_\downarrow} & dk' 4 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta) \\
- \int_0^{2\pi} d\theta \int_0^1 dk \int_0^{Q_d} & dk' 8 \left\{ F_{12}(kk', k, \theta) + F_{14}(kk', k, \theta) \right\} V_s(k', \theta).
\end{aligned}$$

APPENDIX E

C++ utility classes and algorithms

In this section the implementation of C++ classes and algorithms used to generate phase diagrams for both bilayer and trilayer graphene are shown and explained. Figure E.1 shows a unified modeling language (UML) class diagram [62] of the trilayer code. The entry point of all the programs will be the function `main()`.

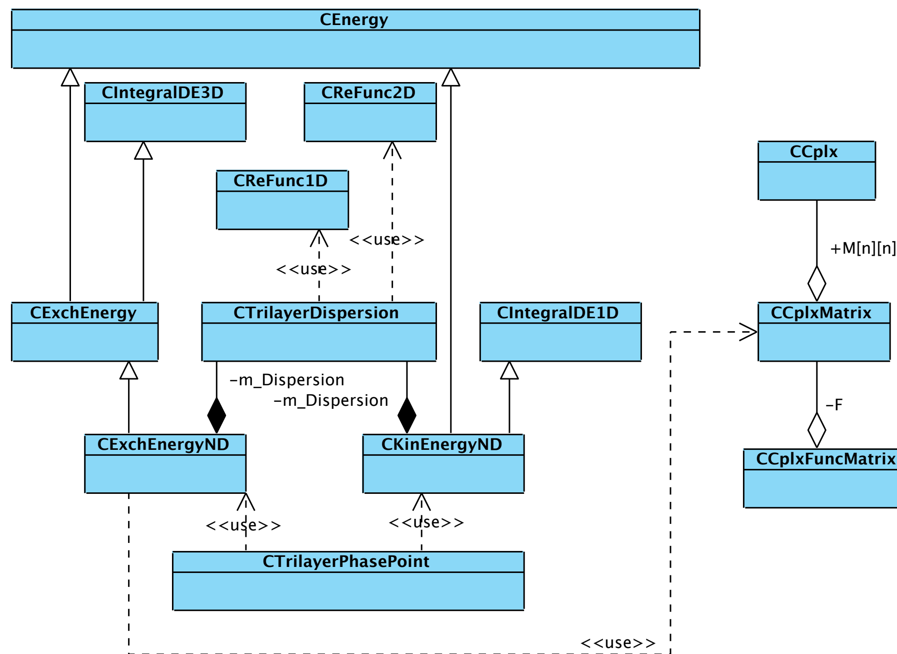


Figure E.1: UML class diagram representation of the complete ABC-trilayer code.

The idea behind the implementation of the program is to created a modular implementation where parts can be reused for other projects. The program is built as a command line program, such that it can be used as a tool to investigate the ferromagnetic properties of graphene when fully implemented. The modular implementation of the program also makes it easy to extend the program to do more advanced calculations. This is at the expense of a relatively larger framework

than what would occur if each task was programmed separately. However, this has not given a large increase in the programs time complexity.

E.1 The CCplx class

This class contains all the functionality needed to do complex arithmetic. This includes:

- adding complex numbers,
- subtracting complex numbers,
- multiplying complex numbers,
- converting complex numbers from polar to rectangular representations and vice versa.

The class is defined as

```
class CCplx
{
public:
    CCplx() { x = 0.0; y = 0.0; }
    CCplx(double dX, double dY) { x = dX; y = dY;}
    CCplx(const CCplx& src) { x = src.x; y = src.y;}


public:
    void Set(double dX, double dY) { x = dX; y = dY;}
    void SetEuler(double dR, double dTheta);
    void operator=(const CCplx& src) { x = src.x; y = src.y; }
    CCplx operator+(const CCplx& z);
    CCplx operator-(const CCplx& z);
    CCplx operator*(const CCplx& z);
    CCplx operator+=(const CCplx& z);
    double Re() { return x; }
    double Im() { return y; }
    CCplx Conj();
    double Mod();
    double Arg();


public:
    double  x;
    double  y;
};
```

The function implementations are shown below. The content of these are self explanatory and no further comments will be made on their behalf. The class is used by making an instance of it. As can be seen from the constructors, the class can be initiated, at construction time, by specifying the real part and imaginary part which are denoted by `dX` and `dY` in the code. If no arguments are given, the class is constructed with both real and imaginary part set to zero. The third constructor is the copy constructor, which allows for the constructed object to receive values of an already existing instance of the class by use of the `=` operator. Furthermore, the `=` operator has been overridden such that it can be used after construction to set the contents of one instance equal to another. Similarly, the `+` and `-` operator is overridden to enable addition and subtraction of complex numbers represented by instances of this class. The `*` operator is overridden to enable multiplication of complex numbers according to $(x_1 + iy_1)(x_2 + iy_2)$, as one should expect by

invoking this operator between two instances of the class. Calling `Re()` or `Im()` of an instance will return the real and imaginary parts, while `Mod()` and `Arg()` returns the moduli and argument of the complex number respectively. Lastly, the `Conj()` method returns an instance which is complex conjugated with respect to the instance where the method is invoked.

```
#include "Cplx.h"
#include <math.h>

void CCplx::SetEuler(double dR, double dTheta)
{
    x = dR*cos(dTheta);
    y = dR*sin(dTheta);
}

CCplx CCplx::operator+(const CCplx& z)
{
    CCplx   Res(x + z.x, y + z.y);

    return Res;
}

CCplx CCplx::operator-(const CCplx& z)
{
    CCplx   Res(x - z.x, y - z.y);

    return Res;
}

CCplx CCplx::operator*(const CCplx& z)
{
    CCplx   Res(z.x*x - z.y*y, z.x*y + z.y*x);

    return Res;
}

CCplx CCplx::operator+=(const CCplx& z)
{
    CCplx   Res(x + z.x, y + z.y);

    x = Res.x;
    y = Res.y;

    return Res;
}

CCplx CCplx::Conj()
{
    CCplx   Res(x, -y);

    return Res;
}

double CCplx::Mod()
{
```

```
    return sqrt(x*x + y*y);
}

double CCplx::Arg()
{
    return atan(y / x);
}
```

E.2 The CCplxMatrix class

Using the `CCplx` class, this class encapsulates the functionality of a complex matrix and typical operations that can be applied to such matrices. These operations include

- matrix multiplication of two complex matrices,
- transpose, complex conjugation and Hermitian conjugate of complex matrices,
- summation of the squares of real parts of elements of upper and lower triangular,
- diagonalization of real and complex matrices.

The class is defined as

```
#include "Cplx.h"

class CCplxMatrix
{
public:
    enum EMatrix { maxMatrixSize = 20 };


public:
    CCplxMatrix();
    CCplxMatrix(int iRows, int iColumns);
    CCplxMatrix(const CCplxMatrix& src);


public:
    void SetSize(int iRows, int iColumns);
    void GetSize(int& iRows, int& iColumns);
    void operator=(const CCplxMatrix& src);
    CCplxMatrix operator*(const CCplxMatrix& src);
    CCplxMatrix operator+(const CCplxMatrix& src);
    CCplxMatrix operator-(const CCplxMatrix& src);
    CCplxMatrix Transpose();
    CCplxMatrix Conj();
    CCplxMatrix ConjTranspose();
    void SetIdentity();
    double SumReUpTri2();
    double SumReLowTri2();
    bool DiagJacobiRe(CCplxMatrix& D, CCplxMatrix& P, int N, int& NRes);
    bool DiagJacobiCplx(CCplxMatrix& D, CCplxMatrix& P, int N, int& NRes);
    bool StandardizeDiagonalizer();
    void Print(bool bImaginary);
```

```
private:
    void RotEq(CCplxMatrix& A, int& r, const double& s, const double& tau,
               const int& p, const int& q, bool bSym);
    void RotD(CCplxMatrix& D, const double& t, const double& s,
              const double& tau, const int& p, const int& q, const int& n);
    void RotP(CCplxMatrix& P, const double& s, const double& tau, const int& p,
              const int& q, const int& n);
    void SortEigVals(CCplxMatrix& D, CCplxMatrix& P);


public:
    CCplx M[maxMatrixSize][maxMatrixSize];

private:
    int m_iRows;
    int m_iColumns;
};
```

The implementation of the methods of the class is shown below. The matrix class can be instantiated by specifying the number of rows and columns in the constructor. A copy constructor is also available. If no arguments are specified when instantiating an object a 2×2 matrix is created. The size of the matrix instance can be adjusted at any time by a call to the `SetSize()` method. The `+`. `-` and `*` is overridden to provide the expected summation, subtraction and complex matrix multiplication between instances of the class. The matrix class does not grow dynamically and is limited by the `maxMatrixSize` enumeration, which is set to 20 in the current implementation of the class. The methods `Transpose()`, `Conj()` and `ConjTranspose()` return instances of the class which have been transposed, conjugated or Hermitian conjugated, as one would expect. In section 2.3 the algorithms used in the implementation of `DiagJacobiRe()`, `DiagJacobiCplx()` and `StandardizeDiagonalizer()` were explained in detail.

```
#include "Matrix.h"
#include <float.h>
#include <math.h>
#include <iostream>

CCplxMatrix::CCplxMatrix()
{
    m_iColumns = 2;
    m_iRows = 2;
}

CCplxMatrix::CCplxMatrix(int iRows, int iColumns)
{
    m_iRows = iRows;
    m_iColumns = iColumns;
}

CCplxMatrix::CCplxMatrix(const CCplxMatrix& src)
{
    *this = src;
}

void CCplxMatrix::SetSize(int iRows, int iColumns)
{
    m_iRows = iRows;
```

```
    m_iColumns = iColumns;
}

void CCplxMatrix::GetSize(int& iRows, int& iColumns)
{
    iRows = m_iRows;
    iColumns = m_iColumns;
}

void CCplxMatrix::operator=(const CCplxMatrix& src)
{
    m_iRows = src.m_iRows;
    m_iColumns = src.m_iColumns;

    for(int i=0; i<m_iColumns; i++)
    {
        for(int j=0; j<m_iRows; j++)
        {
            M[j][i].x = src.M[j][i].x;
            M[j][i].y = src.M[j][i].y;
        }
    }
}

CCplxMatrix CCplxMatrix::operator+(const CCplxMatrix& src)
{
    CCplxMatrix Ret;


    Ret.m_iRows = m_iRows;
    Ret.m_iColumns = m_iColumns;

    for(int i=0; i<m_iRows; i++)
    {
        for(int j=0; j<m_iColumns; j++)
        {
            Ret.M[i][j].x = M[i][j].x + src.M[i][j].x;
            Ret.M[i][j].y = M[i][j].y + src.M[i][j].y;
        }
    }

    return Ret;
}

CCplxMatrix CCplxMatrix::operator-(const CCplxMatrix& src)
{
    CCplxMatrix Ret;


    Ret.m_iRows = m_iRows;
    Ret.m_iColumns = m_iColumns;

    for(int i=0; i<m_iRows; i++)
    {
```

```
        for(int j=0; j<m_iColumns; j++)
        {
            Ret.M[i][j].x = M[i][j].x - src.M[i][j].x;
            Ret.M[i][j].y = M[i][j].y - src.M[i][j].y;
        }
    }

    return Ret;
}

CCplxMatrix CCplxMatrix::operator*(const CCplxMatrix& src)
{
    CCplxMatrix Ret;


    Ret.m_iRows = m_iRows;
    Ret.m_iColumns = src.m_iColumns;

    for(int i=0; i<m_iRows; i++)
    {
        for(int j=0; j<m_iColumns; j++)
        {
            for(int k=0; k<m_iColumns; k++)
            {
                Ret.M[i][j].x += M[i][k].x*src.M[k][j].x - M[i][k].y*src.M[k][j].y;
                Ret.M[i][j].y += M[i][k].x*src.M[k][j].y + M[i][k].y*src.M[k][j].x;
            }
        }
    }

    return Ret;
}

CCplxMatrix CCplxMatrix::Transpose()
{
    CCplxMatrix     Ret;


    Ret.m_iRows = m_iRows;
    Ret.m_iColumns = m_iColumns;

    for(int i=0; i<m_iRows; i++)
    {
        for(int j=0; j<m_iColumns; j++)
        {
            Ret.M[i][j].x = M[j][i].x;
            Ret.M[i][j].y = M[j][i].y;
        }
    }

    return Ret;
}

CCplxMatrix CCplxMatrix::Conj()
```

```
{
    CCplxMatrix     Ret;


    Ret.m_iRows = m_iRows;
    Ret.m_iColumns = m_iColumns;

    for(int i=0; i<m_iRows; i++)
    {
        for(int j=0; j<m_iColumns; j++)
        {
            Ret.M[i][j].x = M[i][j].x;
            Ret.M[i][j].y = -M[i][j].y;
        }
    }

    return Ret;
}

CCplxMatrix CCplxMatrix::ConjTranspose()
{
    CCplxMatrix     Ret;


    Ret.m_iRows = m_iRows;
    Ret.m_iColumns = m_iColumns;

    for(int i=0; i<m_iRows; i++)
    {
        for(int j=0; j<m_iColumns; j++)
        {
            Ret.M[i][j].x = M[j][i].x;
            Ret.M[i][j].y = -M[j][i].y;
        }
    }

    return Ret;
}

void CCplxMatrix::SetIdentity()
{
    int     N = (m_iRows < m_iColumns) ? m_iRows : m_iColumns;

    for(int i=0; i<N; i++) M[i][i].x = 1.0;
}

void CCplxMatrix::RotEq(CCplxMatrix& A, int& r, const double& s, const double& tau,
                        const int& p, const int& q, bool bSym)
{
    double  alpha = -s*(A.M[r][q].x + tau*A.M[r][p].x);
    double  beta = s*(A.M[r][p].x - tau*A.M[r][q].x);


    A.M[r][p].x+= alpha;
```

```
    A.M[r][q].x+= beta;

    if(bSym)
    {
        A.M[p][r].x = A.M[r][p].x;
        A.M[q][r].x = A.M[r][q].x;
    }
}

void CCplxMatrix::RotD(CCplxMatrix& D, const double& t, const double& s,
                       const double& tau, const int& p, const int& q, const int& n)
{
    D.M[p][p].x-= t*D.M[p][q].x;
    D.M[q][q].x+= t*D.M[p][q].x;

    for(int r=0; r<p; r++)
        RotEq(D, r, s, tau, p, q, true);

    for(int r=p+1; r<q; r++)
        RotEq(D, r, s, tau, p, q, true);

    for(int r=q+1; r<n; r++)
        RotEq(D, r, s, tau, p, q, true);

    D.M[p][q].x = D.M[q][p].x = 0.0;
}

double CCplxMatrix::SumReUpTri2()
{
    double  S = 0.0;

    for(int i=0; i<m_iRows; i++)
    {
        for(int j=i+1; j<m_iColumns; j++)
        {
            S+= M[i][j].x * M[i][j].x;
        }
    }

    return S;
}

double CCplxMatrix::SumReLowTri2()
{
    double  S = 0.0;

    for(int i=0; i<m_iColumns; i++)
    {
        for(int j=i+1; j<m_iRows; j++)
        {
            S+= M[j][i].x * M[j][i].x;
        }
    }
```

```
    return S;
}

void CCplxMatrix::RotP(CCplxMatrix& P, const double& s, const double& tau,
                       const int& p, const int& q, const int& n)
{
    for(int r=0; r<n; r++)
        RotEq(P, r, s, tau, p, q, false);
}

void CCplxMatrix::SortEigVals(CCplxMatrix& D, CCplxMatrix& P)
{
    int             n = m_iRows;
    int             iT;
    double          dT;


    for(int i=0; i<(n-1); i++)
    {
        dT = D.M[i][i].x;
        iT = i;
        for(int j=i; j<n; j++)
        {
            if(D.M[j][j].x >= dT)
            {
                dT = D.M[j][j].x;
                iT = j;
            }
        }

        if(i != iT)
        {
            D.M[iT][iT].x = D.M[i][i].x;
            D.M[i][i].x = dT;

            for(int j=0; j<n; j++)
            {
                dT = P.M[j][iT].x;
                P.M[j][iT].x = P.M[j][i].x;
                P.M[j][i].x = dT;

                dT = P.M[j][iT].y;
                P.M[j][iT].y = P.M[j][i].y;
                P.M[j][i].y = dT;
            }
        }
    }
}

bool CCplxMatrix::DiagJacobiRe(CCplxMatrix& D, CCplxMatrix& P, int N, int& NRes)
{
    int     n = m_iRows;
    double  theta, theta2, t, c, s, tau;
    double  S;
```

```
    if(m_iRows != m_iColumns) return false;

    NRes = 0;
    D = *this; P.SetIdentity();
    for(int i=0; i<N; i++)
    {
        for(int p=0; p<(n-1); p++)
        {
            for(int q=p+1; q<n; q++)
            {
                if((i > 4) && (D.M[p][q].x < D.M[p][p].x*1E-20)
                    && (D.M[p][q].x < D.M[q][q].x*1E-20))
                    D.M[p][q].x = D.M[q][p].x = 0.0;

                if(D.M[p][q].x != 0.0)
                {
                    theta = 0.5 * (D.M[q][q].x - D.M[p][p].x) / D.M[p][q].x;
                    theta2 = theta*theta;
                    if(theta2 > FLT_MAX) t = 0.5 / theta;
                    else
                    {
                        t = 1.0 / (fabs(theta) + sqrt(theta2 + 1.0));
                        if(theta < 0.0) t = -t;
                    }

                    c = 1.0 / sqrt(t*t + 1.0);
                    s = t*c;
                    tau = s / (1.0 + c);

                    RotD(D, t, s, tau, p, q, n);
                    RotP(P, s, tau, p, q, n);
                }
            }
        }

        NRes++;
        S = D.SumReUpTri2();
        if(S <= 0.0)
        {
            SortEigVals(D, P);
            return true;
        }
    }

    return false;
}

bool CCplxMatrix::DiagJacobiCplx(CCplxMatrix& D, CCplxMatrix& P, int N, int& NRes)
{
    int             n = m_iRows;
    int             F[n];
    CCplxMatrix     A(2*n, 2*n);
```

```
    CCplxMatrix     DA(2*n, 2*n);
    CCplxMatrix     PA(2*n, 2*n);


    if(m_iRows != m_iColumns) return false;

    for(int i=0; i<n; i++)
    {
        for(int j=0 ; j<n; j++)
        {
            A.M[i][j].x = M[i][j].x;
            A.M[i+n][j+n].x = M[i][j].x;

            A.M[i][j+n].x = -M[i][j].y;
            A.M[i+n][j].x = M[i][j].y;
        }
    }

    if(!A.DiagJacobiRe(DA, PA, N, NRes)) return false;

    for(int i=0; i<n; i++)
    {
        if(PA.M[0][2*i].x > PA.M[0][2*i+1].x) F[i] = 2*i;
        else                                  F[i] = 2*i + 1;

        D.M[i][i].x = DA.M[F[i]][F[i]].x;
    }

    for(int j=0; j<n; j++)
    {
        for(int i=0; i<n; i++)
        {
           P.M[i][j].x = PA.M[i][F[j]].x;
        }
        for(int i=n; i<(2*n); i++)
        {
            P.M[i-n][j].y = PA.M[i][F[j]].x;
        }
    }

    return true;
}

bool CCplxMatrix::StandardizeDiagonalizer()
{
    int             n = m_iRows;
    CCplxMatrix     R(n, n);


    if(m_iRows != m_iColumns) return false;

    if(M[0][0].y != 0.0)    R.M[0][0].SetEuler(1, -M[0][0].Arg());
    else                    R.M[0][0].Set(1.0, 0.0);
```

```
    if(M[0][1].y != 0.0)    R.M[1][1].SetEuler(1, -M[0][1].Arg());
    else                    R.M[1][1].Set(1.0, 0.0);

    if(M[0][2].y != 0.0)    R.M[2][2].SetEuler(1, -M[0][2].Arg());
    else                    R.M[2][2].Set(1.0, 0.0);

    if(M[0][3].y != 0.0)    R.M[3][3].SetEuler(1, -M[0][3].Arg());
    else                    R.M[3][3].Set(1.0, 0.0);

    (*this) = (*this)*R;

    if(M[0][0].x < 0.0)     R.M[0][0].Set(-1.0, 0.0);
    else                    R.M[0][0].Set(1.0, 0.0);

    if(M[0][1].x < 0.0)     R.M[1][1].Set(-1.0, 0.0);
    else                    R.M[1][1].Set(1.0, 0.0);

    if(M[0][2].x < 0.0)     R.M[2][2].Set(-1.0, 0.0);
    else                    R.M[2][2].Set(1.0, 0.0);

    if(M[0][3].x < 0.0)     R.M[3][3].Set(-1.0, 0.0);
    else                    R.M[3][3].Set(1.0, 0.0);

    (*this) = (*this)*R;

    return true;
}

void CCplxMatrix::Print(bool bImaginary)
{
    printf("\r\n\r\n");
    for(int i=0; i<m_iRows; i++)
    {
        for(int j=0; j<m_iColumns; j++)
        {
            if(bImaginary)  printf("%f + %fi\t", M[i][j].x, M[i][j].y);
            else            printf("%f\t", M[i][j].x);
        }

        printf("\r\n");
    }

    printf("\r\n");
}
```

E.3 The CReFunc1D class

This class is responsible for storing a set of values $\{f(x_i)\}$ of cardinality N. Once the function is stored in an instance of the class, the class contains algorithms for finding the inverse. Once a function value is restored, the returned value is a linear interpolation between the stored points. The class is defined as

```
#include "Matrix.h"
#include "Cplx.h"
```

```
class CReFunc1D
{
public:
    enum ESizes { N = 131072 };


public:
    CReFunc1D();
    CReFunc1D(double dXMin, double dXMax);


public:
    int GetN();
    double GetDeltaX();
    void SetF(int i, double f);
    double GetF(double x);
    double GetFInv(double f, bool bJmpToEndp=false);
    void Print(const char* szPrefix);


private:
    double F[N];
    double a;
    double b;
};
```

The function implementations of the class are shown below. When constructing the class one can provide the lower and upper limits of the function domain `dXMin` and `dMax`. If no parameters are provided the limits of the domain are both set to zero. Once the class is instantiated, a function value can be entered by giving an index between 0 and $N-1$ and a function value $f(x)$. The value x of $f(x)$ is calculated based on the provided limits of the domain and the distance between two points in the domain Δx is provided by `GetDeltaX()`. The method `GetF()` returns the function value at the requested point x in the domain by using a linear interpolation. To find the inverse of a function which has been entered into an instance of the class the `GetFInv()` method can be called with the requested value of f. If the boolean parameter `bJmpToEndp=true`, then the upper or lower values of the domain is returned if f is outside the range of the function, if it is set to `false` the method returns "not a number" in the case that f is outside the function range. The algorithms for finding the linear interpolation and for finding the inverse are described in section 2.4.

```
#include "Function.h"
#include <iostream>
#include <math.h>

CReFunc1D::CReFunc1D()
{
    a = b = 0;
}

CReFunc1D::CReFunc1D(double dXMin, double dXMax)
{
    a = dXMin;
    b = dXMax;
}
```

```
int CReFunc1D::GetN()
{
    return N;
}

double CReFunc1D::GetDeltaX()
{
    return (b - a) / double(N);
}

void CReFunc1D::SetF(int i, double f)
{
    if((i < 0) || (i >= N)) return;
    F[i] = f;
}

double CReFunc1D::GetF(double x)
{
    double  DeltaX = GetDeltaX();
    int     i = (int)floor((x - a) / DeltaX);
    double  dx = x - (a + double(i)*DeltaX);


    if(DeltaX == 0.0) return NAN;
    if((i >= (N-1)) || (i < 0)) return NAN;

    return (1.0 / DeltaX) * (F[i+1]*dx - F[i]*(dx - DeltaX));
}

double CReFunc1D::GetFInv(double f, bool bJmpToEndp)
{
    int         i = 0;
    int         j = (N - 1) / 2;
    int         k = N - 1;
    double      D, xs, DeltaX = GetDeltaX();


    if(F[1] > F[0])
    {
        if(!bJmpToEndp)
        {
            if((f < F[0]) || (f > F[k])) return NAN;
        }
        else
        {
            if(f < F[0]) f = F[0];
            if(f > F[k]) f = F[k];
        }

        while(!((F[j-1] <= f) && (f <= F[j])))
        {
            if(F[j] < f) i = j;
            else         k = j;
```

```
            j = (i + k) / 2;
        }
    }

    else if(F[1] < F[0])
    {
        if(!bJmpToEndp)
        {
            if((f > F[0]) || (f < F[k])) return NAN;
        }
        else
        {
            if(f > F[0]) f = F[0];
            if(f < F[k]) f = F[k];
        }

        while(!((F[j-1] >= f) && (f >= F[j])))
        {
            if(F[j] > f) i = j;
            else         k = j;

            j = (i + k) / 2;
        }
    }

    else
    {
        return NAN;
    }

    xs = a + double(j-1)*DeltaX;
    D = F[j] - F[j-1];

    return (D != 0.0) ? xs + DeltaX * (f - F[j-1]) / D: NAN;
}

void CReFunc1D::Print(const char* szPrefix)
{
    double  x;
    double  DeltaX = GetDeltaX();


    printf("F%s={", szPrefix);

    for(int i=0; i<N; i+=8)
    {
        x = a + double(i) * DeltaX;

        if(i != 0) printf(",");
        printf("{%.15f,%.15f}", x, F[i]);
    }

    printf("};\r\n");
```

```
    printf("ListLinePlot[F%s]\r\n", szPrefix);
}
```

E.4 The CReFunc2D class

This class is responsible for storing a function of two variables. Only a point set $\{f(x_i, y_j)\}$ of cardinality N is stored. When the function is retrieved, the retrieved values are a linear interpolation of the original function. Note that this class cannot find the inverse as was the case with `CReFunc1D` class. The `CReFunc2D` is defined as

```
#include "Matrix.h"
#include "Cplx.h"

class CReFunc2D
{
public:
    enum ESizes { N = 256 };


public:
    CReFunc2D();
    CReFunc2D(double dXMin, double dXMax, double dYMin, double dYMax);


public:
    int GetN();
    double GetDeltaX();
    double GetDeltaY();
    void SetF(int i, int j, double f);
    double GetF(double x, double y);
    void Print(const char* szPrefix);


private:
    double F[N][N];
    double ax;
    double bx;
    double ay;
    double by;
};
```

The function implementations of the class are shown below. When the object is instantiated the limits of a rectangular function domain can be provided into `dXMin`, `dXMax`, `dYMin` and `dYMax`. If no limits of the domain are provided, these will be set to zero. The function values are entered into an object by a call to `SetF()` by providing two indices `i,j` accompanied by a function value $f(x, y)$. As for the `CReFunc1D` class, the values of x and y are calculated automatically based on the given limits of the function domain. The distances between points Δx and Δy are retrieved by `GetDeltaX()` and `GetDeltaY()`, respectively. The function value retrieved by `GetF()` is a two dimensional linear interpolation of the original function similar to the one dimensional linear interpolation described in section 2.4.

```
#include "Function.h"
#include <iostream>
#include <math.h>
```

```
CReFunc2D::CReFunc2D()
{
    ax = bx = ay = by = 0;
}

CReFunc2D::CReFunc2D(double dXMin, double dXMax, double dYMin, double dYMax)
{
    ax = dXMin;
    bx = dXMax;
    ay = dYMin;
    by = dYMax;
}

int CReFunc2D::GetN()
{
    return N;
}

double CReFunc2D::GetDeltaX()
{
    return (bx - ax) / double(N);
}

double CReFunc2D::GetDeltaY()
{
    return (by - ay) / double(N);
}

void CReFunc2D::SetF(int i, int j, double f)
{
    if((i < 0) || (i >= N)) return;
    if((j < 0) || (j >= N)) return;
    F[i][j] = f;
}

double CReFunc2D::GetF(double x, double y)
{
    double  DeltaX = GetDeltaX();
    double  DeltaY = GetDeltaY();
    double  DeltaA = DeltaX * DeltaY;
    int     i = (int)floor((x - ax) / DeltaX);
    int     j = (int)floor((y - ay) / DeltaY);
    double  dx = x - (ax + double(i)*DeltaX);
    double  dy = y - (ay + double(j)*DeltaY);
    double  DX = DeltaX - dx;
    double  DY = DeltaY - dy;


    if(DeltaA == 0.0) return NAN;
    if((i >= (N-1)) || (i < 0)) return NAN;
    if((j >= (N-1)) || (j < 0)) return NAN;

    return (F[i][j]*DX*DY + F[i][j+1]*dy*DX
```

```
        + F[i+1][j]*dx*DY + F[i+1][j+1]*dx*dy) / DeltaA;
}

void CReFunc2D::Print(const char* szPrefix)
{
    double  x, y;
    double  DeltaX = GetDeltaX();
    double  DeltaY = GetDeltaY();


    printf("F%s={", szPrefix);

    for(int i=0; i<N; i+=8)
    {
        x = ax + double(i) * DeltaX;
        for(int j=0; j<N; j+=8)
        {
            y = ay + double(j) * DeltaY;

            if((i != 0) || (j != 0)) printf(",");
            printf("{%.15f,%.15f,%.15f}", x, y, F[i][j]);
        }
    }

    printf("};\r\n");
    printf("ListPlot3D[F%s]\r\n", szPrefix);
}
```

E.5 The CCplxFuncMatrix2D class

This class inherits functionality from and uses the `CCplxMatrix` object. The principle of this class is identical to that of the `CReFunc2D` class with the exception that instead of storing real values, it stores complex matrices of type `CCplxMatrix`.

```
#include "Matrix.h"
#include "Cplx.h"

class CCplxFuncMatrix2D : public CCplxMatrix
{
public:
    enum EMode { printRe=0, printIm=1, printMod=2, printReAbs=3 };

public:
    CCplxFuncMatrix2D();
    CCplxFuncMatrix2D(double dXMin, double dXMax,
                      double dYMin, double dYMax, int iN);
    virtual ~CCplxFuncMatrix2D();


public:
    int GetN();
    double GetDeltaX();
    double GetDeltaY();
    void SetF(int i, int j, CCplxMatrix& f);
```

```
    bool GetF(double x, double y, bool bInterpolate, int iDeg=1);
    void Print(const char* szPrefix, int iRow, int iColumn, EMode Mode);


private:
    CCplxMatrix* F;
    double ax;
    double bx;
    double ay;
    double by;
    int N;
};
```

The function implementations of the class are shown below. The usage is identical to `CCplxMatrix` except that the method `GetF()` now sets the matrix values of `CCplxFuncMatrix2D` itself. This can be done since `CCplxFuncMatrix2D` inherits from `CCplxMatrix` and hence is an object of this type. This object can use first and second order interpolation on each of the cells in the matrix with a call to `GetF()`. The interpolation order is given by `iDeg`. It is also possible to skip the interpolation altogether and return the closest point in the stored set by setting `bInterpolate=false`.

```
#include "Function.h"
#include <iostream>
#include <math.h>

CCplxFuncMatrix2D::CCplxFuncMatrix2D() : CCplxMatrix()
{
    ax = ay = 0.0;
    bx = by = 1.0;
    N = 30;
    F = new CCplxMatrix[N * N];
}

CCplxFuncMatrix2D::CCplxFuncMatrix2D(double dXMin, double dXMax,
                                     double dYMin, double dYMax,
                                     int iN)
                                     : CCplxMatrix(maxMatrixSize, maxMatrixSize)
{
    ax = dXMin;
    bx = dXMax;
    ay = dYMin;
    by = dYMax;

    N = iN;
    F = new CCplxMatrix[N * N];
}

CCplxFuncMatrix2D::~CCplxFuncMatrix2D()
{
    if(F) delete [] F;
}

int CCplxFuncMatrix2D::GetN()
{
    return N;
}
```

```
double CCplxFuncMatrix2D::GetDeltaX()
{
    return (bx - ax) / double(N);
}

double CCplxFuncMatrix2D::GetDeltaY()
{
    return (by - ay) / double(N);
}

void CCplxFuncMatrix2D::SetF(int i, int j, CCplxMatrix& f)
{
    int     iRows, iCols;
    int     iRowsMin, iColsMin;
    int     l1;


    if((i < 0) || (i >= N)) return;
    if((j < 0) || (j >= N)) return;

    l1 = i + j*N;

    GetSize(iRowsMin, iColsMin);
    f.GetSize(iRows, iCols);
    if(iCols < iColsMin) iColsMin = iCols;
    if(iRows < iRowsMin) iRowsMin = iRows;
    SetSize(iRowsMin, iColsMin);

    F[l1] = f;
    F[l1].SetSize(iRowsMin, iColsMin);
}

bool CCplxFuncMatrix2D::GetF(double x, double y, bool bInterpolate, int iDeg)
{
    double  DeltaX = GetDeltaX();
    double  DeltaY = GetDeltaY();
    int     l1, l2;


    if(bInterpolate)
    {
        if(iDeg == 1)
        {
            int     i = (int)floor((x - ax) / DeltaX);
            int     j = (int)floor((y - ay) / DeltaY);
            double  DeltaA = DeltaX * DeltaY;
            double  dx = x - (ax + double(i)*DeltaX);
            double  dy = y - (ay + double(j)*DeltaY);
            double  DX = DeltaX - dx;
            double  DY = DeltaY - dy;
            int     iRows, iCols;

            if((i >= (N-1)) || (i < 0)) return false;
```

```
    if((j >= (N-1)) || (j < 0)) return false;
    if(DeltaA == 0.0) return false;

    l1 = i + j*N;
    l2 = l1 + N;

    GetSize(iRows, iCols);
    for(int m=0; m<iRows; m++)
    {
        for(int n=0; n<iCols; n++)
        {
            M[m][n].x = (F[l1].M[m][n].x*DX*DY + F[l1+1].M[m][n].x*dx*DY
             + F[l2].M[m][n].x*DX*dy + F[l2+1].M[m][n].x*dx*dy) / DeltaA;
            M[m][n].y = (F[l1].M[m][n].y*DX*DY + F[l1+1].M[m][n].y*dx*DY
             + F[l2].M[m][n].y*DX*dy + F[l2+1].M[m][n].y*dx*dy) / DeltaA;
        }
    }
}
else if(iDeg == 2)
{
    int     i = (int)floor((x - ax) / DeltaX);
    int     j = (int)floor((y - ay) / DeltaY);
    double  dx = x - (ax + double(i)*DeltaX);
    double  dy = y - (ay + double(j)*DeltaY);
    double  phiX[3], phiY[3];
    double  etaX = 2.0*DeltaX;
    double  etaY = 2.0*DeltaY;
    double  etaX2 = etaX * etaX;
    double  etaY2 = etaY * etaY;
    CCplx   Res;
    int     iRows, iCols;

    if((i >= (N-2)) || (i < 0)) return false;
    if((j >= (N-2)) || (j < 0)) return false;
    if((etaX2 == 0.0) || (etaY2 == 0.0)) return false;

    phiX[0] = (2.0 / etaX2) * (dx - etaX) * (dx - etaX / 2.0);
    phiX[1] = (-4.0 / etaX2) * dx * (dx - etaX);
    phiX[2] = (2.0 / etaX2) * dx * (dx - etaX / 2.0);

    phiY[0] = (2.0 / etaY2) * (dy - etaY) * (dy - etaY / 2.0);
    phiY[1] = (-4.0 / etaY2) * dy * (dy - etaY);
    phiY[2] = (2.0 / etaY2) * dy * (dy - etaY / 2.0);

    GetSize(iRows, iCols);
    for(int m=0; m<iRows; m++)
    {
        for(int n=0; n<iCols; n++)
        {
            Res.Set(0.0, 0.0);

            for(int il=0; il<3; il++)
            {
                for(int jl=0; jl<3; jl++)
```

```
                {
                    Res.x+= F[(i+il) + (j+jl)*N].M[m][n].x
                          * phiX[il] * phiY[jl];
                    Res.y+= F[(i+il) + (j+jl)*N].M[m][n].y
                          * phiX[il] * phiY[jl];
                }
            }

            M[m][n] = Res;
        }
    }
}
else
{
    int     i = (int)floor((x - ax) / DeltaX);
    int     j = (int)floor((y - ay) / DeltaY);
    double  dx = x - (ax + double(i)*DeltaX);
    double  dy = y - (ay + double(j)*DeltaY);
    double  phiX[4], phiY[4];
    double  etaX = 3.0*DeltaX;
    double  etaY = 3.0*DeltaY;
    double  etaX3 = etaX * etaX * etaX;
    double  etaY3 = etaY * etaY * etaY;
    CCplx   Res;
    int     iRows, iCols;

    if((i >= (N-3)) || (i < 0)) return false;
    if((j >= (N-3)) || (j < 0)) return false;
    if((etaX3 == 0.0) || (etaY3 == 0.0)) return false;

    phiX[0] = (-9.0 / (2.0*etaX3))
         * (dx - etaX / 3.0) * (dx - 2.0 * etaX / 3.0) * (dx - etaX);
    phiX[1] = (27.0 / (2.0*etaX3))
         * dx * (dx - 2.0 * etaX / 3.0) * (dx - etaX);
    phiX[2] = (-27.0 / (2.0*etaX3))
         * dx * (dx - etaX / 3.0) * (dx - etaX);
    phiX[3] = (9.0 / (2.0*etaX3))
         * dx * (dx - etaX / 3.0) * (x - 2.0 * etaX / 3.0);

    phiY[0] = (-9.0 / (2.0*etaY3))
         * (dy - etaY / 3.0) * (dy - 2.0 * etaY / 3.0) * (dy - etaY);
    phiY[1] = (27.0 / (2.0*etaY3))
         * dy * (dy - 2.0 * etaY / 3.0) * (dy - etaY);
    phiY[2] = (-27.0 / (2.0*etaY3))
         * dy * (dy - etaY / 3.0) * (dy - etaY);
    phiY[3] = (9.0 / (2.0*etaY3))
         * dy * (dy - etaY / 3.0) * (y - 2.0 * etaY / 3.0);

    GetSize(iRows, iCols);
    for(int m=0; m<iRows; m++)
    {
        for(int n=0; n<iCols; n++)
        {
            Res.Set(0.0, 0.0);
```

```
                    for(int il=0; il<4; il++)
                    {
                        for(int jl=0; jl<4; jl++)
                        {
                            Res.x+= F[(i+il) + (j+jl)*N].M[m][n].x
                                 * phiX[il] * phiY[jl];
                            Res.y+= F[(i+il) + (j+jl)*N].M[m][n].y
                                 * phiX[il] * phiY[jl];
                        }
                    }

                    M[m][n] = Res;
                }
            }
        }
    }

    else
    {
        int     i = (int)round((x - ax) / DeltaX);
        int     j = (int)round((y - ay) / DeltaY);

        if((i >= N) || (i < 0)) return false;
        if((j >= N) || (j < 0)) return false;

        l1 = i + j*N;
        *((CCplxMatrix*)this) = F[l1];
    }

    return true;
}

void CCplxFuncMatrix2D::Print(const char* szPrefix, int iRow,
                              int iColumn, EMode Mode)
{
    double  x, y, v;
    double  DeltaX = GetDeltaX();
    double  DeltaY = GetDeltaY();
    int     l1;
    int     D = N / 30.0;
    int     m = iRow, n=iColumn;


    if(D == 0) D = 1;
    printf("F%s={", szPrefix);

    for(int i=0; i<N; i+=D)
    {
        x = ax + double(i) * DeltaX;
        for(int j=0; j<N; j+=D)
        {
            l1 = i + j*N;
            y = ay + double(j) * DeltaY;
```

```
            if((i != 0) || (j != 0)) printf(",");

            if(Mode == printRe)
                   v = F[l1].M[m][n].x;
            else if(Mode == printIm)
                   v = F[l1].M[m][n].y;
            else if(Mode == printMod)
                   v = sqrt(F[l1].M[m][n].x*F[l1].M[m][n].x
                     + F[l1].M[m][n].y*F[l1].M[m][n].y);
            else if(Mode == printReAbs)
                   v = fabs(F[l1].M[m][n].x);
            else
                   v = F[l1].M[m][n].x;

            printf("{%.15f,%.15f,%.15f}", x, y, v);
        }
    }

    printf("};\r\n");
    printf("ListPlot3D[F%s]\r\n", szPrefix);
}
```

E.6 The CIntegralDE1D class

This class implements a single variable version of the double exponential algorithm described in section 2.1. The class is defined as

```
class CIntegralDE1D
{
public:
    CIntegralDE1D();
    CIntegralDE1D(int N);
    virtual ~CIntegralDE1D();


public:
    void Init(int N);
    virtual double Integrate(void);

protected:
    virtual double f(double x) = 0;


private:
    int         m_N;
    double*     pDT;
    double*     pT;
};
```

The function implementations of the class are shown below. This class is used by creating a class that inherits from `CIntegralDE1D` and implements the pure virtual function `f()`. Polymorphism will ensure that the correct function `f()` is invoked. The function `f()` must return a single variable function $f(x)$ to be integrated. Once an instance of the new class has been created, a call

to `Integrate()` will return

$$\int_{-1}^{1} f(x)dx.$$

The accuracy is defined by N, which gives the number of partitions of the domain.

```
#include "IntegralDE.h"
#include <math.h>
#include <float.h>
#include <iostream>

CIntegralDE1D::CIntegralDE1D()
{
    pT = NULL;
    pDT = NULL;

    Init(30);
}

CIntegralDE1D::CIntegralDE1D(int N)
{
    pT = NULL;
    pDT = NULL;

    Init(N);
}

CIntegralDE1D::~CIntegralDE1D()
{
    if(pT) delete [] pT;
    if(pDT) delete [] pDT;
}

void CIntegralDE1D::Init(int N)
{
    double     x, dx;
    double     c;


    m_N = N;

    ////////////////////////////////////////////////////
    // Prepare double int. transformations
    ////////////////////////////////////////////////////

    dx = log(4.0*double(N)) / double(N);

    if(pDT) delete [] pDT;
    pDT = new double[2*N+1];

    if(pT) delete [] pT;
    pT = new double[2*N+1];

    for(int k=-N; k<N; k++)
    {
```

```
        x = double(k) * dx;

        pT[k + N] = tanh(M_PI / 2.0 * sinh(x));

        c = cosh(M_PI / 2.0 * sinh(x));
        pDT[k + N] = (M_PI / 2.0 * cosh(x)) / (c * c);
    }
}

double CIntegralDE1D::Integrate(void)
{
    double     dx, I=0.0;


    ///////////////////////////////////////////////////
    // Algorithm for intgrating f(x) over [-1,1]
    // using double exponential transf. and a cutoff
    // Riemann's sum
    ///////////////////////////////////////////////////

    dx = log(4.0*double(m_N)) / double(m_N);

    for(int i=-m_N; i<m_N; i++)
    {
        I += f(pT[i + m_N])*dx*pDT[i + m_N];
    }

    return I;
}
```

E.7 The CIntegralDE3D class

This class implements a triple variable version of the double exponential algorithm described in section 2.1. The class is defined as

```
class CIntegralDE3D
{
public:
    CIntegralDE3D();
    CIntegralDE3D(int N);
    virtual ~CIntegralDE3D();


public:
    void Init(int N);
    virtual double Integrate();
    virtual double Integrate(double xs, double xe,
                            double ys, double ye, double zs, double ze);
    virtual double TestAccuracy(double& dAccuracy);

protected:
    virtual double f(double x, double y, double z) = 0;
```

```
private:
    int         m_N;
    double*     pDT;
    double*     pT;
};
```

The function implementations are shown below. The class operates in an identical manner as the `CIntegralDE1D` class except that $f(x) \to f(x,y,z)$. Thus, a call to `Integrate()` with no arguments returns

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(x,y,z)dxdydz.$$

If arguments are entered into `Integrate()` it will return

$$\int_{xs}^{xe}\int_{ys}^{ye}\int_{zs}^{ze} f(x,y,z)dxdydz.$$

In this class, N indicates the number of partitions of the domain in each direction x, y and z (i.e. the x-axis is divided into N sub intervals as well as the y-axis).

```
#include "IntegralDE.h"
#include <math.h>
#include <float.h>
#include <iostream>

CIntegralDE3D::CIntegralDE3D()
{
    pT = NULL;
    pDT = NULL;

    Init(30);
}

CIntegralDE3D::CIntegralDE3D(int N)
{
    pT = NULL;
    pDT = NULL;

    Init(N);
}

CIntegralDE3D::~CIntegralDE3D()
{
    if(pT) delete [] pT;
    if(pDT) delete [] pDT;
}

void CIntegralDE3D::Init(int N)
{
    double     x, dx;
    double     c;


    m_N = N;

    //////////////////////////////////////////////////
```

```
    // Prepare double int. transformations
    ////////////////////////////////////////////////////

    dx = log(4.0*double(N)) / double(N);

    if(pDT) delete [] pDT;
    pDT = new double[2*N+1];

    if(pT) delete [] pT;
    pT = new double[2*N+1];

    for(int k=-N; k<N; k++)
    {
        x = double(k) * dx;

        pT[k + N] = tanh(M_PI / 2.0 * sinh(x));

        c = cosh(M_PI / 2.0 * sinh(x));
        pDT[k + N] = (M_PI / 2.0 * cosh(x)) / (c * c);
    }
}

double CIntegralDE3D::Integrate(void)
{
    double     dx, I=0.0;


    ////////////////////////////////////////////////////
    // Algorithm for intgrating f(x,y,z) over [-1,1]^3
    // using double exponential transf. and a cutoff
    // Riemann's sum
    ////////////////////////////////////////////////////

    dx = log(4.0*double(m_N)) / double(m_N);

    for(int i=-m_N; i<m_N; i++)
    {
        for(int j=-m_N; j<m_N; j++)
        {
            for(int k=-m_N; k<m_N; k++)
            {
                I += f(pT[i + m_N], pT[k + m_N], pT[j + m_N])
                  *dx*dx*dx*pDT[i + m_N]*pDT[k + m_N]*pDT[j + m_N];
            }
        }
    }

    return I;
}

double CIntegralDE3D::Integrate(double xs, double xe,
                                double ys, double ye, double zs, double ze)
{
    double     dx, I=0.0;
```

```
    double     ax = (xe - xs) / 2.0;
    double     ay = (ye - ys) / 2.0;
    double     az = (ze - zs) / 2.0;
    double     bx = (xe + xs) / (xe - xs);
    double     by = (ye + ys) / (ye - ys);
    double     bz = (ze + zs) / (ze - zs);


    ///////////////////////////////////////////////////
    // Algorithm for intgrating f(x,y,z) over [-1,1]^3
    // using double exponential transf. and a cutoff
    // Riemann's sum
    ///////////////////////////////////////////////////

    dx = log(4.0*double(m_N)) / double(m_N);

    for(int i=-m_N; i<m_N; i++)
    {
        for(int j=-m_N; j<m_N; j++)
        {
            for(int k=-m_N; k<m_N; k++)
            {
                I += f(ax*(pT[i + m_N] + bx),
                       ay*(pT[k + m_N] + by), az*(pT[j + m_N] + bz))
                   *dx*dx*dx*pDT[i + m_N]*pDT[k + m_N]*pDT[j + m_N];
            }
        }
    }

    return I*ax*ay*az;
}

double CIntegralDE3D::TestAccuracy(double& dAccuracy)
{
    int     iOldN = m_N;
    double  dRes1, dRes2;


    Init(iOldN+1);
    dRes1 = Integrate();

    Init(iOldN);
    dRes2 = Integrate();

    if(dRes1 == 0.0)
    {
        dAccuracy = 0.0;
        return 0.0;
    }

    dAccuracy = fabs(dRes1 - dRes2) / fabs(dRes1);

    return dRes2;
}
```

E.8 The CCmdLineParser class

The responsibility of this class is related to the control of the program. It takes input parameters that originates from `main()`. These parameters in turn comes from the command line that is entered to start the program. The class is defined as

```
class CCmdLineParser
{
public:
    enum ELimits { maxSwitchLen = 10, maxArgs = 20, maxNumLen = 30 };


public:
    class CArg
    {
    public:
        double m_dVal;
        char m_cSwitch[maxSwitchLen+1];
    };


public:
    CCmdLineParser() {}


public:
    bool Parse(int argc, const char * argv[]);
    int GetNumArgs(){ return m_iNumArgs; }
    CArg GetArg(int iIndex) { return m_Arguments[iIndex]; }


private:
    int m_iNumArgs;
    CArg m_Arguments[maxArgs];
};
```

The function implementations are shown below. The decoding (or parsing) of the command line parameters is executed by a call to `Parse()`. Once this has been done, the number of command line parameters that was collected is found by a call to `GetNumArgs()` and the name plus numeric value for one of these parameters can be retrieved by a call to `GetArg()`. According to the implementation of this class, a command line parameter should be formatted as `<Program name> -<argument name>=<numeric value> -<argument name>=<numeric value>....`

```
#include "CmdLineParser.h"
#include <math.h>
#include <iostream>

bool CCmdLineParser::Parse(int argc, const char * argv[])
{
    int     i, j, k;
    char    szNum[maxNumLen];


    m_iNumArgs = 0;
```

```
    i = 0;
    for(int n=1; n<argc; n++)
    {
        if(i >= maxArgs) break;

        j = 0;
        while(argv[n][j] != 0)
        {
            if(argv[n][j] == '-')
            {
                memset(m_Arguments[i].m_cSwitch, 0, maxSwitchLen+1);

                j++; k = 0;
                while(argv[n][j] != 0)
                {
                    if(k >= maxSwitchLen) break;
                    m_Arguments[i].m_cSwitch[k] = argv[n][j];

                    j++; k++;
                    if(argv[n][j] == '=')
                    {
                        memset(szNum, 0, maxNumLen);

                        j++; k = 0;
                        while(argv[n][j] != 0)
                        {
                            if(k >= maxNumLen) break;
                            szNum[k] = argv[n][j];

                            j++; k++;
                        }

                        m_Arguments[i].m_dVal = atof(szNum);
                        if(isnan(m_Arguments[i].m_dVal))
                        {
                            printf("Error! Could not parse command line!\r\n");
                            return false;
                        }

                        i++;
                        m_iNumArgs++;
                    }
                }
            }
            else
            {
                j++;
            }
        }
    }

    return true;
}
```

E.9 The CTimer class

This class has as responsibility to time features of the program and report that time to the command line interface. The class is defined as

```
class CTimer
{
public:
    CTimer() { m_lStart = 0; m_lStop = 0; }


public:
    void Start();
    void Stop();
    void PrintElapsed();


private:
    long m_lStart;
    long m_lStop;
};
```

The function implementations are shown below. A call to `Start()` of a timer instance will note the time of the call internally in the instance. A call to `Stop()` will note the time of the stop call, while a call to `PrintElapsed()` calculate the elapsed time and writes it to the command line.

```
#include "Timer.h"
#include <iostream>
#include <time.h>

void CTimer::Start()
{
    m_lStart = (long)time(NULL);
}

void CTimer::Stop()
{
    m_lStop = (long)time(NULL);
}

void CTimer::PrintElapsed()
{
    printf("\r\n\r\nElapsed time: %ld s\r\n\r\n", m_lStop-m_lStart);
}
```

APPENDIX F

Bilayer C++ code

F.1 The CBiLayerDispersion class

This class is responsible for all calculations related to the bilayer single particle dispersion relation. It can calculate

- the full dispersion relation,
- the low energy dispersion (based on the linearized matrix Hamiltonian),
- the 1D low energy dispersion emerging from an intersection at one of the K-points of the Brillouin zone,
- the diagonalizing matrix of the low energy matrix Hamiltonian,
- the density of states $\mathcal{D}(\epsilon)$ and $\mathcal{D}(p)$.

The resulting functions are stored in the various function classes `CReFunc1D`, `CReFunc2D` and `CCplxFuncMatrix2D` described in appendix E. The class is defined as

```
#include "Function.h"

class CBiLayerDispersion
{
public:
    CBiLayerDispersion();
    virtual ~CBiLayerDispersion();


public:
    void InitFullDispersion(double dKMin, double dKMax);
    void InitLin2DDispersion(double dKMin, double dKMax);
    void InitLin1DDispersion(double dKMin, double dKMax);
    void InitLinDiagMatrix(double dKMin, double dKMax);
    void InitLin1DDensityOfStates(double dKMin, double dKMax);
    CReFunc2D* GetFullDispersion(int iBand);
    CReFunc2D* GetLin2DDispersion(int iBand);
    CReFunc1D* GetLin1DDispersion(int iBand);
```

```
    CCplxFuncMatrix2D* GetLinDiagMatrix();
    void GetLinDiagMatrix(double p, double phi, CCplxMatrix& Md);
    double Get1DDensityOfStates(int iBand, double x, bool bXIsP=true);
    void PrintFullDispersion();
    void PrintLin2DDispersion();
    void PrintLin1DDispersion();
    void PrintLinDiagMatrix();
    void Print1DDensityOfStates(int iBand, double dEMin, double dEMax,
                                bool bEIsP=false);


private:
    CReFunc2D*          m_pFullDispersion[4];
    CReFunc2D*          m_pLin2DDispersion[4];
    CReFunc1D*          m_pLin1DDispersion[4];
    CCplxFuncMatrix2D*  m_pLinDiagMatrix;

    CCplxMatrix         H;
    CCplxMatrix         M;
    CCplxMatrix         E;
};
```

The function implementations are shown below. In order to calculate one of the dispersions described above, or the density of states, the class must fist be instantiated. Calculation of a dispersion or density of states are done with a call to one of the corresponding `Init...()` methods. These methods take as argument the momentum range (k_min, k_max) that the functions are to be calculated for. To get access to the various functions after a call to `Init...()`, a call to the corresponding `Get...()` method can be made. Arguments to some of these methods include the index of the band to return `iBand`. The method `GetLinDiagMatrix()` does not have a corresponding `Init...()` method that needs to be called. Instead it will immediately calculate the matrix at the specified momentum p and angle ϕ in momentum space and return it as a complex matrix `CCplxMatrix` in `Md`. The method `Get1DDensityOfStates()` takes as argument the band `iBand` that the density should be calculated for and `x`. If `bXIsP=true` then `x` is a momentum p such that $\mathcal{D} = \mathcal{D}(p)$ and if `bXIsP=false` then `x` is an energy and $\mathcal{D} = \mathcal{D}(\epsilon)$. All the dispersions are calculated by using the diagonalization features of the `CCplxMatrix` class by entering a matrix version of the Hamiltonian into it. The `Print...()` methods will output the corresponding function values to the command line and display this information in a Mathematica format.

```
#include "Dispersion.h"
#include <math.h>
#include <stdio.h>
#include <iostream>

CBiLayerDispersion::CBiLayerDispersion()
{
    for(int i=0; i<4; i++)
    {
        m_pFullDispersion[i] = NULL;
        m_pLin2DDispersion[i] = NULL;
        m_pLin1DDispersion[i] = NULL;
    }

    m_pLinDiagMatrix = NULL;
```

```
    H.SetSize(4, 4);
    M.SetSize(4, 4);
    E.SetSize(4, 4);
}

CBiLayerDispersion::~CBiLayerDispersion()
{
    for(int i=0; i<4; i++)
    {
        if(m_pFullDispersion[i]) delete m_pFullDispersion[i];
        if(m_pLin2DDispersion[i]) delete m_pLin2DDispersion[i];
        if(m_pLin1DDispersion[i]) delete m_pLin1DDispersion[i];
    }

    if(m_pLinDiagMatrix) delete m_pLinDiagMatrix;
}

void CBiLayerDispersion::PrintFullDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<4; i++)
    {
        if(m_pFullDispersion[i])
        {
            sprintf(szPrefix, "E%i", i);
            m_pFullDispersion[i]->Print(szPrefix);
        }
    }
}

void CBiLayerDispersion::PrintLin2DDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<4; i++)
    {
        if(m_pLin2DDispersion[i])
        {
            sprintf(szPrefix, "EL%i", i);
            m_pLin2DDispersion[i]->Print(szPrefix);
        }
    }
}

void CBiLayerDispersion::PrintLin1DDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<4; i++)
    {
        if(m_pLin1DDispersion[i])
        {
            sprintf(szPrefix, "EL%i", i);
```

```
            m_pLin1DDispersion[i]->Print(szPrefix);
        }
    }
}

void CBiLayerDispersion::PrintLinDiagMatrix()
{
    int     iColumns, iRows;
    char    szPrefix[10];


    if(!m_pLinDiagMatrix) return;

    m_pLinDiagMatrix->GetSize(iRows, iColumns);
    for(int m=0; m<iRows; m++)
    {
        for(int n=0; n<iColumns; n++)
        {
            sprintf(szPrefix, "Re%i%i", m, n);
            m_pLinDiagMatrix->Print(szPrefix, m, n, CCplxFuncMatrix2D::printRe);
            sprintf(szPrefix, "Im%i%i", m, n);
            m_pLinDiagMatrix->Print(szPrefix, m, n, CCplxFuncMatrix2D::printIm);
        }
    }
}

void CBiLayerDispersion::Print1DDensityOfStates(int iBand, double dEMin,
                                                double dEMax, bool bEIsP)
{
    double  D;

    if(m_pLin1DDispersion[iBand])
    {
        printf("FDL%i={", iBand);
        for(double E=dEMin; E<dEMax; E+=(dEMax-dEMin)/50.0)
        {
            D = Get1DDensityOfStates(iBand, E, bEIsP);
            if(E != dEMin) printf(",");
            printf("{%.15f,%.15f}", E, D);
        }
        printf("};\r\n");
        printf("ListLinePlot[FDL%i]\r\n", iBand);
    }
}

void CBiLayerDispersion::InitFullDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    CCplx           C[2];
    double          x, y, DeltaX, DeltaY;
    double          v;
    const double    a = 1.5551;
    const double    sqrt3_2 = 0.8660254037844386;
    double          delta[3][2] = {{0.5*a, sqrt3_2*a},{0.5*a, -sqrt3_2*a},{-a, 0}};
```

```
    CReFunc2D*      F[4];
    const double    t_perp = 0.05;
    const double    t = 8.57*t_perp;


    for(int i=0; i<4; i++)
    {
        if(m_pFullDispersion[i]) delete m_pFullDispersion[i];
        F[i] = m_pFullDispersion[i] = new CReFunc2D(dKMin, dKMax, dKMin, dKMax);
    }

    DeltaX = F[0]->GetDeltaX();
    DeltaY = F[0]->GetDeltaY();
    N = F[0]->GetN();

    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            x = dKMin + double(i)*DeltaX;
            y = dKMin + double(j)*DeltaY;

            H.M[0][2].x = -t_perp;
            H.M[2][0].x = -t_perp;

            H.M[0][1].Set(0.0, 0.0);
            H.M[1][0].Set(0.0, 0.0);

            H.M[2][3].Set(0.0, 0.0);
            H.M[3][2].Set(0.0, 0.0);

            for(int k=0; k<3; k++)
            {
                v = x*delta[k][0] + y*delta[k][1];
                C[0].SetEuler(-t, v);
                C[1].SetEuler(-t, -v);

                H.M[0][1] += C[1];
                H.M[1][0] += C[0];

                H.M[2][3] += C[0];
                H.M[3][2] += C[1];
            }

            H.DiagJacobiCplx(E, M, 30, NRes);

            F[0]->SetF(i, j, E.M[0][0].x);
            F[1]->SetF(i, j, E.M[1][1].x);
            F[2]->SetF(i, j, E.M[2][2].x);
            F[3]->SetF(i, j, E.M[3][3].x);
        }
    }
}
```

```
void CBiLayerDispersion::InitLin2DDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    double          x, y, DeltaX, DeltaY;
    CReFunc2D*      F[4];
    const double    t_perp = 0.05;


    for(int i=0; i<4; i++)
    {
        if(m_pLin2DDispersion[i]) delete m_pLin2DDispersion[i];
        F[i] = m_pLin2DDispersion[i] = new CReFunc2D(dKMin, dKMax, dKMin, dKMax);
    }

    DeltaX = F[0]->GetDeltaX();
    DeltaY = F[0]->GetDeltaY();
    N = F[0]->GetN();

    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            x = dKMin + double(i)*DeltaX;
            y = dKMin + double(j)*DeltaY;

            H.M[0][2].x = -t_perp;
            H.M[2][0].x = -t_perp;

            H.M[0][1].Set(y, -x);
            H.M[1][0].Set(y, x);

            H.M[2][3].Set(y, x);
            H.M[3][2].Set(y, -x);

            H.DiagJacobiCplx(E, M, 30, NRes);

            F[0]->SetF(i, j, E.M[0][0].x);
            F[1]->SetF(i, j, E.M[1][1].x);
            F[2]->SetF(i, j, E.M[2][2].x);
            F[3]->SetF(i, j, E.M[3][3].x);
        }
    }
}

void CBiLayerDispersion::InitLin1DDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    double          x, DeltaX;
    CReFunc1D*      F[4];
    const double    t_perp = 0.05;


    for(int i=0; i<4; i++)
    {
```

```
        if(m_pLin1DDispersion[i]) delete m_pLin1DDispersion[i];
        F[i] = m_pLin1DDispersion[i] = new CReFunc1D(dKMin, dKMax);
    }

    DeltaX = F[0]->GetDeltaX();
    N = F[0]->GetN();

    for(int i=0; i<N; i++)
    {
        x = dKMin + double(i)*DeltaX;

        H.M[0][2].x = -t_perp;
        H.M[2][0].x = -t_perp;

        H.M[0][1].Set(0.0, -x);
        H.M[1][0].Set(0.0, x);

        H.M[2][3].Set(0.0, x);
        H.M[3][2].Set(0.0, -x);

        H.DiagJacobiCplx(E, M, 30, NRes);

        F[0]->SetF(i, E.M[0][0].x);
        F[1]->SetF(i, E.M[1][1].x);
        F[2]->SetF(i, E.M[2][2].x);
        F[3]->SetF(i, E.M[3][3].x);
    }
}

void CBiLayerDispersion::InitLinDiagMatrix(double dKMin, double dKMax)
{
    int                 N, NRes;
    double              x, y, DeltaX, DeltaY;
    CCplxMatrix         Md(4, 4);
    CCplxFuncMatrix2D*  F;
    const double        t_perp = 0.05;
    const int           C[4] = {2, 1, 3, 0};


    if(m_pLinDiagMatrix) delete m_pLinDiagMatrix;
    F = m_pLinDiagMatrix = new CCplxFuncMatrix2D(dKMin, dKMax, 0.0, 2.0*M_PI, 30);

    DeltaX = F->GetDeltaX();
    DeltaY = F->GetDeltaY();
    N = F->GetN();

    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            x = dKMin + double(i)*DeltaX;
            y = double(j)*DeltaY;

            H.M[0][2].x = -t_perp;
```

```
            H.M[2][0].x = -t_perp;

            H.M[0][1].SetEuler(x, -y);
            H.M[1][0].SetEuler(x, y);

            H.M[2][3].SetEuler(x, y);
            H.M[3][2].SetEuler(x, -y);

            if(!H.DiagJacobiCplx(E, M, 5000, NRes))
                printf("Error Diagonalizing\r\n");

            M.StandardizeDiagonalizer();

            for(int i=0; i<4; i++)
            {
                for(int j=0; j<4; j++)
                {
                    Md.M[i][C[j]].x = M.M[i][j].x;
                    Md.M[i][C[j]].y = M.M[i][j].y;
                }
            }

            F->SetF(i, j, Md);
        }
    }
}

void CBiLayerDispersion::InitLin1DDensityOfStates(double dKMin, double dKMax)
{
    InitLin1DDispersion(dKMin, dKMax);
}

CReFunc2D* CBiLayerDispersion::GetFullDispersion(int iBand)
{
    return m_pFullDispersion[iBand];
}

CReFunc2D* CBiLayerDispersion::GetLin2DDispersion(int iBand)
{
    return m_pLin2DDispersion[iBand];
}

CReFunc1D* CBiLayerDispersion::GetLin1DDispersion(int iBand)
{
    return m_pLin1DDispersion[iBand];
}

CCplxFuncMatrix2D* CBiLayerDispersion::GetLinDiagMatrix()
{
    return m_pLinDiagMatrix;
}

void CBiLayerDispersion::GetLinDiagMatrix(double p, double phi, CCplxMatrix& Md)
{
```

```
    const double        t_perp = 0.05;
    int                 NRes;


    H.M[0][2].x = -t_perp;
    H.M[2][0].x = -t_perp;

    H.M[0][1].SetEuler(p, -phi);
    H.M[1][0].SetEuler(p, phi);

    H.M[2][3].SetEuler(p, phi);
    H.M[3][2].SetEuler(p, -phi);

    if(!H.DiagJacobiCplx(E, Md, 5000, NRes)) printf("Error Diagonalizing!\r\n");
}

double CBiLayerDispersion::Get1DDensityOfStates(int iBand, double x, bool bXIsP)
{
    CReFunc1D*  E = GetLin1DDispersion(iBand);
    double      depsilon = 1E-7;
    double      epsilon = bXIsP ? E->GetF(x) : x;
    double      k1 = E->GetFInv(epsilon + depsilon);
    double      k2 = E->GetFInv(epsilon);

    return  (k1*k1 - k2*k2) / depsilon;
}
```

F.2 The CEnergy class

This class is a pure virtual class. Its only purposes is to provide the common value of $t_\perp = 0.05$, such that this value is stored only one place, and to introduce polymorphic behavior to the `Calculate()` method. The class is defined as

```
class CEnergy
{
public:
    enum ECarriers { carrierOneType=1, carrierTwoTypes };

public:
    CEnergy() { t_perp = 0.05; }

public:
    virtual double Calculate(double Qd, double Qsu, double Qsd,
                             double g, ECarriers Carriers, double* pAccuracy=0) = 0;

protected:
    double t_perp;
};
```

F.3 The CExchEnergy class

This class inherits from the `CEnergy` and must therefore implement a `Calculate()` method. Since the class also inherits from `CIntegralDE3D` the class is an integral and must therefore implement

a `f(x)` method. The integral to be calculated here is the exchange integral. The bilayer exchange integral can be calculated in two ways; by using analytic diagonalization and by using numerical diagonalization. The two methods will need a different implementation of the $F_{ij}(k, k', \theta)$ function of equation (4.22). Therefore, the method that corresponds to this function is implemented as a pure virtual function such that a class that inherits from this can decide upon behavior of F_{ij} through use of polymorphism. The class is defined as

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CExchEnergy : public CEnergy, public CIntegralDE3D
{
public:
    CExchEnergy();
    CExchEnergy(int N);


public:
    double Calculate(double Qd, double Qsu, double Qsd, double g,
                     ECarriers Carriers, double* pAccuracy=0);
    double EvalIntegrand(double k, double p, double phi, double Qd,
                         double Qsu, double Qsd, double g, int iPart);

protected:
    double f(double k, double p, double phi);

protected:
    virtual double F(int i, int j, double k, double p, double phi, double g) = 0;
    virtual int Band(int iNumber) = 0;
    double V(double k, double p, double phi);
    void Init(void);


protected:
    int                 m_iPart;
    double              m_Qd;
    double              m_Qsu;
    double              m_Qsd;
    double              m_g;
};
```

The function implementations are shown below. The method `Calculate()` will return the bilayer exchange energy $\Delta E_{ex}/A_c$ by calculating equation (4.21). As already mentioned, F_{ij} in $\Delta E_{ex}/A_c$ is calculated in a class that inherits from `CExchEnergy`. The argument `Carriers` can be either one type of charge carrier `carrierOneType` or two types of charge carriers `carrierTwoTypes`. Furthermore, `Qd` represents Q_d, `Qsu` represents $Q_\uparrow$ and `Qsd` represents $Q_\downarrow$. It is also possible to evaluate the integrand of the $\Delta E_{ex}/A_c$ integration by a call to `EvalIntegrand` which can be used to plot the integrand, which has been useful for investigating the singularities. Note that the actual formula that is used for the calculations is shown in equation (4.23) for two carriers and equation (4.24) for one carrier.

```
#include "Energies.h"
#include <math.h>
#include <float.h>
```

```
#include <iostream>

CExchEnergy::CExchEnergy() : CIntegralDE3D(30), CEnergy()
{
    CExchEnergy::Init();
}

CExchEnergy::CExchEnergy(int N) : CIntegralDE3D(N), CEnergy()
{
    CExchEnergy::Init();
}

void CExchEnergy::Init()
{
    m_Qd = 0.0;
    m_Qsu = 0.5;
    m_Qsd = 0.5;
    m_g = 6.0;

    m_iPart = 0;
}

double CExchEnergy::Calculate(double Qd, double Qsu, double Qsd, double g,
                              ECarriers Carriers, double* pAccuracy)
{
    double              dRes = 0.0;


    m_Qd = Qd;
    m_Qsu = Qsu;
    m_Qsd = Qsd;
    m_g = g;

    if(pAccuracy)
    {
        m_iPart = 1;
        TestAccuracy(*pAccuracy);
    }

    if(Carriers == carrierTwoTypes)
    {
        m_iPart = 1;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 2;
        dRes += (Qsu != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 3;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 4;
        dRes += (Qsd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 5;
```

```
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 6;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 7;
        dRes += CIntegralDE3D::Integrate();
    }

    if(Carriers == carrierOneType)
    {
        m_iPart = 8;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 2;
        dRes += (Qsu != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 3;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 9;
        dRes += (Qsd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 5;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 6;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 10;
        dRes += CIntegralDE3D::Integrate();
    }

    return dRes;
}

double CExchEnergy::f(double k, double p, double phi)
{
    double  dRes = 0.0;
    double  e;
    int     a, b, d;


    a = Band(1);
    b = Band(2);
    d = Band(4);

    k = 0.5*(k + 1.0);

    e = M_PI;
    phi = e*(phi + 1.0);

    switch(m_iPart)
    {
```

```
case 1:
    p = 0.5*(p + 1.0);
    dRes = (2.0 * m_Qsu * (F(a, a, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                        +F(a, b, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                        +F(a, d, m_Qsu*k, m_Qsu*k*p, phi, m_g))
            - 2.0 * m_Qsd * F(b, d, m_Qsd*k, m_Qsd*k*p, phi, m_g)
            - 4.0 * m_Qd *  (F(a, a, m_Qd*k, m_Qd*k*p, phi, m_g)
                          +F(a, b, m_Qd*k, m_Qd*k*p, phi, m_g)
                          +F(a, d, m_Qd*k, m_Qd*k*p, phi, m_g)))
                          *V(k, p, phi);
    dRes*= 0.5;
    break;

case 2:
    e = 1.0 / m_Qsu;
    p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
    dRes = 2.0 * m_Qsu * (F(a, b, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                       +F(a, d, m_Qsu*k, m_Qsu*k*p, phi, m_g))
                       *V(k, p, phi);
    dRes*= (e - 1.0) / 2.0;
    break;

case 3:
    e = 1.0 / m_Qd;
    p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
    dRes = -4.0 * m_Qd * (F(a, b, m_Qd*k, m_Qd*k*p, phi, m_g)
                       +F(a, d, m_Qd*k, m_Qd*k*p, phi, m_g))
                       *V(k, p, phi);
    dRes*= (e - 1.0) / 2.0;
    break;

case 4:
    e = 1.0 / m_Qsd;
    p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
    dRes = -2.0 * m_Qsd * (F(b, b, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                        +F(b, d, m_Qsd*k, m_Qsd*k*p, phi, m_g))
                        *V(k, p, phi);
    dRes*= (e - 1.0) / 2.0;
    break;

case 5:
    e = m_Qsu / 2.0;
    p = e*(p + 1.0);
    dRes = 2.0 * (F(a, b, k*p, k, phi, m_g)
                 +F(a, d, k*p, k, phi, m_g))
                 *V(k, p, phi);
    dRes*= e;
    break;

case 6:
    e = m_Qd / 2.0;
    p = e*(p + 1.0);
    dRes = -4.0 * (F(a, b, k*p, k, phi, m_g)
                  +F(a, d, k*p, k, phi, m_g))
```

```
                        *V(k, p, phi);
            dRes*= e;
            break;

        case 7:
            e = m_Qsd / 2.0;
            p = e*(p + 1.0);
            dRes = -2.0 * (F(b, b, k*p, k, phi, m_g)
                          +F(b, d, k*p, k, phi, m_g))
                          *V(k, p, phi);
            dRes*= e;
            break;

        case 8:
            p = 0.5*(p + 1.0);
            dRes = (2.0 * m_Qsu * (F(a, a, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                  +F(a, b, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                  +F(a, d, m_Qsu*k, m_Qsu*k*p, phi, m_g))
                    + 2.0 * m_Qsd * (F(a, a, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                    +F(a, b, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                    +F(a, d, m_Qsd*k, m_Qsd*k*p, phi, m_g))
                    - 4.0 * m_Qd *  (F(a, a, m_Qd*k, m_Qd*k*p, phi, m_g)
                                    +F(a, b, m_Qd*k, m_Qd*k*p, phi, m_g)
                                    +F(a, d, m_Qd*k, m_Qd*k*p, phi, m_g)))
                                    *V(k, p, phi);
            dRes*= 0.5;
            break;

        case 9:
            e = 1.0 / m_Qsd;
            p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
            dRes = 2.0 * m_Qsd * (F(a, b, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                 +F(a, d, m_Qsd*k, m_Qsd*k*p, phi, m_g))
                                 *V(k, p, phi);
            dRes*= (e - 1.0) / 2.0;
            break;

        case 10:
            e = m_Qsd / 2.0;
            p = e*(p + 1.0);
            dRes = 2.0 * (F(a, b, k*p, k, phi, m_g)
                         +F(a, d, k*p, k, phi, m_g))
                         *V(k, p, phi);
            dRes*= e;
            break;
    }

    dRes*= 0.5;
    dRes*= M_PI;

    return dRes;
}

double CExchEnergy::V(double k, double p, double phi)
```

```
{
    double  A = p*p - 2*p*cos(phi) + 1.0;
    double  B;


    B = sqrt((A < 0.0) ? 0.0 : A);
    if(B <= 0.0) return 0.0;

    return 1.0 / B;
}

double CExchEnergy::EvalIntegrand(double k, double p, double phi,
                                  double Qd, double Qsu, double Qsd, double g, int iPart)
{
    m_Qd = Qd;
    m_Qsu = Qsu;
    m_Qsd = Qsd;
    m_g = g;

    m_iPart = iPart;

    return f(k, p, phi);
}
```

F.4 The CExchEnergyAD class

This class implements the $F_{ij}(k, k', \theta)$ function of equation (4.22) by using analytic diagonalization to find the diagonalizing matrices of the Hamiltonian. The matrix multiplications are carried out using the `CCplxMatrix` class. Also, since different methods of implementing F_{ij} results in different indexing of the band, the virtual function `Band()` is used to provide a consistent band assignment in the `CExchEnergy` class. The `CExchEnergyAD` class is defined as

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CExchEnergyAD : public CExchEnergy
{
public:
    CExchEnergyAD();
    CExchEnergyAD(int N);


private:
    double F(int i, int j, double k, double p, double phi, double g);
    int Band(int iNumber);
    void Init();


private:
    CCplxMatrix         M1_p;
    CCplxMatrix         M1_k;
    CCplxMatrix         M2;
    CCplxMatrix         M3_p;
```

```
    CCplxMatrix         M3_k;
    CCplxMatrix         M_p;
    CCplxMatrix         M_k;
    CCplxMatrix         IM;
    CCplxMatrix         Chi_pk[2];
    CCplxMatrix         Chi_kp[2];
};
```

The function implementations of this class are shown below. When constructing the object, the parameter N represents the number of partitions to use in the integration. It has been found through experimentation that this value should range between $N = 23$ to $N = 30$ for varying accuracy of the resulting integration, where $N = 23$ is poor but representative and $N = 30$ yields an accuracy such that the error bars are not visible in a plot. Note that N is transferred directly to the `CIntegralDE3D` class and the time complexity of the integration algorithm is of $\mathcal{O}(N^3)$. Since `CExchEnergyAD` does no numerical diagonalization, it will be faster than the `CExchEnergyND` class.

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>

CExchEnergyAD::CExchEnergyAD() : CExchEnergy()
{
    CExchEnergyAD::Init();
}

CExchEnergyAD::CExchEnergyAD(int N) : CExchEnergy(N)
{
    CExchEnergyAD::Init();
}

void CExchEnergyAD::Init()
{
    M1_p.SetSize(4, 4);
    M1_k.SetSize(4, 4);
    M2.SetSize(4, 4);
    M3_p.SetSize(4, 4);
    M3_k.SetSize(4, 4);
    M_p.SetSize(4, 4);
    M_k.SetSize(4, 4);
    IM.SetSize(4, 4);

    Chi_pk[0].SetSize(4, 4);
    Chi_pk[1].SetSize(4, 4);
    Chi_kp[0].SetSize(4, 4);
    Chi_kp[1].SetSize(4, 4);

    M2.M[0][0].x = M_SQRT1_2;
    M2.M[0][2].x = M_SQRT1_2;
    M2.M[1][1].x = M_SQRT1_2;
    M2.M[1][3].x = M_SQRT1_2;
    M2.M[2][0].x = M_SQRT1_2;
    M2.M[2][2].x = -M_SQRT1_2;
    M2.M[3][1].x = M_SQRT1_2;
```

```
    M2.M[3][3].x = -M_SQRT1_2;

    IM.M[0][0].x = 1.0;
    IM.M[1][1].x = 1.0;
    IM.M[2][2].x = -1.0;
    IM.M[3][3].x = -1.0;
}

double CExchEnergyAD::F(int i, int j, double k, double p, double phi, double g)
{
    const double        d16PiSquareI = 0.006332573977646111;
    const double        d = 3.7;
    const double        vF = 1.0;
    double              Cos_k, Sin_k, Cos_p, Sin_p;
    double              t_perp2 = t_perp*t_perp;
    double              k2;
    double              p2;
    double              A_k, A_p;
    double              A, B, C;
    double              V_NS[2];
    double              dRet;


    k2 = k*k;
    p2 = p*p;

    A_k = sqrt(t_perp2 + 4.0*k2);
    Cos_k = M_SQRT1_2*sqrt((A_k + t_perp) / A_k);
    Sin_k = M_SQRT1_2*sqrt((A_k - t_perp) / A_k);

    A_p = sqrt(t_perp2 + 4.0*p2);
    Cos_p = M_SQRT1_2*sqrt((A_p + t_perp) / A_p);
    Sin_p = M_SQRT1_2*sqrt((A_p - t_perp) / A_p);

    M1_k.M[0][0].x = 1.0;
    M1_k.M[1][1].SetEuler(1.0, phi);
    M1_k.M[2][2].x = 1.0;
    M1_k.M[3][3].SetEuler(1.0, -phi);

    M3_k.M[0][0].x = Cos_k;
    M3_k.M[0][1].x = Sin_k;
    M3_k.M[1][0].x = -Sin_k;
    M3_k.M[1][1].x = Cos_k;
    M3_k.M[2][2].x = Cos_k;
    M3_k.M[2][3].x = -Sin_k;
    M3_k.M[3][2].x = Sin_k;
    M3_k.M[3][3].x = Cos_k;

    M3_p.M[0][0].x = Cos_p;
    M3_p.M[0][1].x = Sin_p;
    M3_p.M[1][0].x = -Sin_p;
    M3_p.M[1][1].x = Cos_p;
    M3_p.M[2][2].x = Cos_p;
    M3_p.M[2][3].x = -Sin_p;
```

```
    M3_p.M[3][2].x = Sin_p;
    M3_p.M[3][3].x = Cos_p;


    M_k = M1_k*M2*M3_k;
    M_p = M2*M3_p;


    Chi_kp[0] = M_k.ConjTranspose()*M_p;
    Chi_kp[1] = M_k.ConjTranspose()*IM*M_p;
    Chi_pk[0] = Chi_kp[0].ConjTranspose();
    Chi_pk[1] = Chi_kp[1].ConjTranspose();

    C = p2 + k2 - 2*p*k*cos(phi);
    A = sqrt((C < 0.0) ? 0.0 : C);
    B = exp(-d*A);
    V_NS[0] = 1.0 + B;
    V_NS[1] = 1.0 - B;


    dRet = 0.0;
    for(int iAlpha=0; iAlpha<2; iAlpha++)
    {
        dRet += (Chi_kp[iAlpha].M[i][j]*Chi_pk[iAlpha].M[j][i]).Re()*V_NS[iAlpha];
    }

    return -d16PiSquareI * g * vF * p * k * dRet;
}

int CExchEnergyAD::Band(int iNumber)
{
    if(iNumber == 1) return 1;
    if(iNumber == 2) return 3;
    if(iNumber == 3) return 2;
    if(iNumber == 4) return 0;

    return -1;
}
```

F.5 The CExchEnergyND class

This class implements the $F_{ij}(k, k', \theta)$ function of equation (4.22) by using numerical diagonalization to find the diagonalizing matrices of the Hamiltonian. The matrix multiplications and diagonalizations are carried out using the `CCplxMatrix` class. Also, since different methods of implementing F_{ij} results in different indexing of the band, the virtual function `Band()` is used to provide a consistent band assignment in the `CExchEnergy` class. The `CExchEnergyND` class is defined as

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CExchEnergyND : public CExchEnergy
{
```

```
public:
    CExchEnergyND();
    CExchEnergyND(int N);


private:
    double F(int i, int j, double k, double p, double phi, double g);
    int Band(int iNumber);
    double ThomasFermiWaveVec2();
    void Init();


private:
    CBiLayerDispersion  m_Dispersion;
    CCplxMatrix         M_p;
    CCplxMatrix         M_k;
    CCplxMatrix         IM;
    CCplxMatrix         Chi_pk[2];
    CCplxMatrix         Chi_kp[2];
};
```

The function implementations of this class are shown below. When constructing the object, the parameter N represents the number of partitions to use in the integration. It has been found through experimentation that this value should range between $N = 23$ to $N = 30$ for varying accuracy of the resulting integration, where $N = 23$ is poor but representative and $N = 30$ yields an accuracy such that the error bars are not visible in a plot. Note that N is transferred directly to the `CIntegralDE3D` class and the time complexity of the integration algorithm is of $\mathcal{O}(N^3)$. Since `CExchEnergyAD` does no numerical diagonalization, it will be faster than the `CExchEnergyND` class.

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>

CExchEnergyND::CExchEnergyND() : CExchEnergy()
{
    CExchEnergyND::Init();
}

CExchEnergyND::CExchEnergyND(int N) : CExchEnergy(N)
{
    CExchEnergyND::Init();
}

void CExchEnergyND::Init()
{
    M_p.SetSize(4, 4);
    M_k.SetSize(4, 4);
    IM.SetSize(4, 4);

    Chi_pk[0].SetSize(4, 4);
    Chi_pk[1].SetSize(4, 4);
    Chi_kp[0].SetSize(4, 4);
    Chi_kp[1].SetSize(4, 4);
```

```
    IM.M[0][0].x = 1.0;
    IM.M[1][1].x = 1.0;
    IM.M[2][2].x = -1.0;
    IM.M[3][3].x = -1.0;
}

double CExchEnergyND::F(int i, int j, double k, double p, double phi, double g)
{
    const double        d16PiSquareI = 0.006332573977646111;
    const double        d = 3.7;
    const double        vF = 1.0;
    double              k2;
    double              p2;
    double              A, B, C;
    double              V_NS[2];
    double              dRet;


    k2 = k*k;
    p2 = p*p;

    m_Dispersion.GetLinDiagMatrix(k, phi, M_k);
    m_Dispersion.GetLinDiagMatrix(p, 0.0, M_p);

    Chi_kp[0] = M_k.ConjTranspose()*M_p;
    Chi_kp[1] = M_k.ConjTranspose()*IM*M_p;
    Chi_pk[0] = Chi_kp[0].ConjTranspose();
    Chi_pk[1] = Chi_kp[1].ConjTranspose();

    C = p2 + k2 - 2*p*k*cos(phi);
    A = sqrt((C < 0.0) ? 0.0 : C);
    B = exp(-d*A);
    V_NS[0] = 1.0 + B;
    V_NS[1] = 1.0 - B;


    dRet = 0.0;
    for(int iAlpha=0; iAlpha<2; iAlpha++)
    {
        dRet += (Chi_kp[iAlpha].M[i][j]*Chi_pk[iAlpha].M[j][i]).Re()*V_NS[iAlpha];
    }

    return -d16PiSquareI * g * vF * p * k * dRet;
}

int CExchEnergyND::Band(int iNumber)
{
    if(iNumber == 1) return 1;
    if(iNumber == 2) return 2;
    if(iNumber == 3) return 0;
    if(iNumber == 4) return 3;

    return -1;
```

```
}
```

F.6 The CKinEnergyAD class

This class calculates the kinetic energy $\Delta E_k/A_c$ of equation (4.25) by using analytic diagonalization to find the dispersion. Since it inherits from `CEnergy` it must provide the `Calculate()` method. The class is defined as

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CKinEnergyAD : public CEnergy
{
public:
    CKinEnergyAD();


public:
    double Calculate(double Qd, double Qsu, double Qsd, double g,
                     ECarriers Carriers, double* pAccuracy=0);

private:
    double E1(const double& k);
    double F(double E);
};
```

The function implementations of this class is shown below.

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>

CKinEnergyAD::CKinEnergyAD() : CEnergy()
{
}

double CKinEnergyAD::Calculate(double Qd, double Qsu, double Qsd, double g,
                               ECarriers Carriers, double* pAccuracy)
{
    return F(E1(Qsu)) + F(E1(Qsd)) - 2.0*F(E1(Qd));
}

double CKinEnergyAD::F(double E)
{
    return 0.5*M_1_PI * (1.0 / 3.0 * E*E*E + 1.0 / 4.0 * E*E * t_perp);
}

double CKinEnergyAD::E1(const double& k)
{
    return -t_perp / 2.0 + sqrt(t_perp*t_perp / 4.0 + k*k);
}
```

F.7 The CKinEnergyND class

This class calculates the kinetic energy $\Delta E_k/A_c$ of equation (4.25) by using numerical diagonalization to find the dispersion. Since it inherits from `CEnergy` it must provide the `Calculate()` method. The class is defined as

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CKinEnergyND : public CEnergy, public CIntegralDE1D
{
public:
    CKinEnergyND();


public:
    double Calculate(double Qd, double Qsu, double Qsd, double g,
                     ECarriers Carriers, double* pAccuracy=0);

protected:
    double f(double x);

private:
    double F(double E);
    double N1(double epsilon);


private:
    CBiLayerDispersion  m_Dispersion;
    double              m_dE;
};
```

The function implementations are shown below. Since this calss calculates dispersions by numerical diagonalization it uses the `CBilayerDispersion` class to provide the single particle dispersion. The class is also an integral through inheriting from the `CIntegralDE1D` class. This is necessary to perform the integration needed to find the kinetic energy from the dispersion relation. The inverse of the dispersion is provided by the `CReFunc1D` class.

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>

CKinEnergyND::CKinEnergyND() : CEnergy()
{
    m_Dispersion.InitLin1DDispersion(0.0, 1.5);
    m_dE = 0.0;
}

double CKinEnergyND::Calculate(double Qd, double Qsu, double Qsd, double g,
                               ECarriers Carriers, double* pAccuracy)
{
    CReFunc1D*      E1 = m_Dispersion.GetLin1DDispersion(1);
    double          E1_Qsu = E1->GetF(Qsu);
```

```
    double          E1_Qsd = E1->GetF(Qsd);
    double          E1_Qd = E1->GetF(Qd);

    return F(E1_Qsu) + F(E1_Qsd) - 2.0*F(E1_Qd);
}

double CKinEnergyND::N1(double epsilon)
{
    double          k;
    CReFunc1D*      E1 = m_Dispersion.GetLin1DDispersion(1);
    const double    d4PiInv = 0.07957747154594767;

    k = E1->GetFInv(epsilon);
    return d4PiInv * k*k;
}

double CKinEnergyND::f(double x)
{
    double          epsilon = 0.5*m_dE*(x + 1.0);

    return 0.5 * m_dE * N1(epsilon);
}

double CKinEnergyND::F(double E)
{
    m_dE = E;
    return E*N1(E) - Integrate();
}
```

F.8 The CBiLayerPhasePoint class

This class is responsible for calculating

- $\Delta E/A_c$ as a function of x,
- single points in the electron-electron coupling versus doping phase diagram,
- multiple points in the electron-electron coupling versus doping phase diagram.

The mode of diagonalization used by the class is defined on construction of the class. The class is defined as

```
#include "Energies.h"

class CBiLayerPhasePoint
{
public:
    enum EMode { modeAnalyticalDiag = 0, modeNumericalDiag = 1 };

public:
    CBiLayerPhasePoint() { m_DiagnMode = modeAnalyticalDiag; }


public:
    double PlotSingleDEvsQ(int iIndex, const char* szPrefix, double Qd,
```

```
                          double g, int iPts, double Xmax, bool bCenterAtQd,
                          int iMeshPts=30, bool bMeasureAccuracy=false);
    void PlotMultipleDEvsQ(const char* szPrefix, double QdL, double QdH, int NQd,
                          double g, double xH, bool bCenterAtQd, int iMesh);
    void PlotBinSearchDEvsQ(const char* szPrefix, double QdL, double QdH, int NQd,
                           double g, double xH, bool bCenterAtQd, int iMesh);
    void PlotAllDEvsQ(bool bCenterAtQd, int iMesh, int iSequence);

private:
    double FindMinimum(double* aE, double& dx, int iSize);
    double PlotSingleDEvsQ(CEnergy& KinE, CEnergy& ExchE, int iIndex,
                          const char* szPrefix, double Qd, double g,
                          int iPts, double Xmax, bool bCenterAtQd,
                          int iMeshPts, bool bMeasureAccuracy);


public:
    EMode m_DiagnMode;
};
```

The function implementations are shown below. `PlotSingleDEvsQ()` will calculate a single function $\Delta E/A_c(X)$ where $X = x$ if `bCenterAtQd=false` or $X = -x'$ if `PlotSingleDEvsQ=true` (see section 4.2.6 for definitions of x and x'). The function will be calculated with a given doping `Qd`, electron-electron coupling `g` and the number of points that are calculated for the graph are given by `iPts` while the range is $(-x_max, x_max)$ if `bCenterAtQd=false` and $(-x_max, 0)$ if `bCenterAtQd=true`. The `iIndex` and `szPrefix` will provide a number and a text string that will identify the output graph, which will appear in Mathematica format on the command line. Note that the integration accuracy is given by `iMesh`, which corresponds to the parameter N of the integration class.

The method `PlotMultipleDEvsQ()` will make multiple calls to `PlotSingleDEvsQ()` in order to create a batch of functions in the domain $(-x_h, x_h)$ or $(-x_h, 0)$ depending on the `bCenterAtQd` parameter. There will be `NQd` plots in the batch ranging from doping level `QdL` to `QdH`.

The `PlotBinSearchDEvsQ()` method is equivalent to the `PlotMultipleDEvsQ()` method, but will instead use the binary search algorithm of section 2.5 to successively find a doping closer to the critical of the curves $\Delta E/A_c(x)$.

The method `PlotAllDEvsQ()` contains a series of predefined calls to the above methods based on the `iSequence` parameter in a script like fashion. This can be used to preprogram a full phase diagram, that will appear as all the collective graphs as Mathematica code. Graphs such as the ones in figure 4.10 and figure 4.11 are automatically generated and is imported directly into Mathematica for analysis. From these graphs one can determine a single point in the phase diagram.

```
#include "BiLayerPhasePoint.h"
#include <math.h>
#include <float.h>
#include <iostream>

void CBiLayerPhasePoint::PlotAllDEvsQ(bool bCenterAtQd, int iMesh, int iSequence)
{
    if(iSequence == 1)
    {
        PlotMultipleDEvsQ("010", 0.0, 0.001, 6, 0.1, -0.00002,
```

```
                        bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("025", 0.0, 0.001, 6, 0.25, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("050", 0.0, 0.002, 6, 0.5, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("1", 0.0, 0.002, 6, 1.0, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("2", 0.0, 0.002, 6, 2.0, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("3", 0.0, 0.002, 6, 3.0, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("4", 0.0, 0.002, 6, 4.0, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("5", 0.0, 0.002, 6, 5.0, -0.00002,
                          bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("6", 0.0, 0.002, 6, 6.0, -0.00002,
                          bCenterAtQd, iMesh);
    }
    else if(iSequence == 3)
    {
        PlotBinSearchDEvsQ("010", 0.0, 0.001, 8, 0.1, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("025", 0.0, 0.001, 8, 0.25, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("050", 0.0, 0.002, 8, 0.5, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("1", 0.0, 0.002, 8, 1.0, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("2", 0.0, 0.002, 8, 2.0, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("3", 0.0, 0.002, 8, 3.0, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("4", 0.0, 0.002, 8, 4.0, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("5", 0.0, 0.002, 8, 5.0, -0.00002,
                           bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("6", 0.0, 0.002, 8, 6.0, -0.00002,
                           bCenterAtQd, iMesh);
    }
    else
    {
        printf("Sequence does not exist!\r\n");
    }
}

void CBiLayerPhasePoint::PlotMultipleDEvsQ(const char* szPrefix,
     double QdL, double QdH, int NQd, double g, double xH,
     bool bCenterAtQd, int iMesh)
{
    double          dQd, Qd;
    double*         aMin;


    printf("Running with\r\n");
```

```
    printf("------------\r\n");
    printf("QdL \t= %.15f\r\n", QdL);
    printf("QdH \t= %.15f\r\n", QdH);
    printf("NQd \t= %i\r\n", NQd);
    printf("g \t= %.15f\r\n", g);
    printf("xH \t= %.15f\r\n", xH);
    printf("Mesh \t= %i\r\n", iMesh);
    printf("DMode \t= %s\r\n", (m_DiagnMode == modeAnalyticalDiag)
                              ? "Analytical" : "Numerical");
    printf("------------\r\n\r\n");


    aMin = new double[NQd];

    dQd = (QdH - QdL) / double(NQd - 1);
    for(int i=0; i<NQd; i++)
    {
        Qd = QdL + dQd * double(i);

        aMin[i] = PlotSingleDEvsQ(i+1, szPrefix, Qd, g, 25, xH,
                                  bCenterAtQd, iMesh);
    }

    printf("\r\nxmins%s={", szPrefix);
    for(int i=0; i<NQd; i++)
    {
        Qd = QdL + dQd * double(i);
        if(i!=0) printf(",");
        printf("{%.15f,%.15f}", Qd, aMin[i]);
    }
    printf("}\r\n\r\n\r\n");

    delete [] aMin;
}

void CBiLayerPhasePoint::PlotBinSearchDEvsQ(const char* szPrefix, double QdL,
        double QdH, int NQd, double g, double xH, bool bCenterAtQd, int iMesh)
{
    double          Q, Q1, Q2;
    double          DE, DE1, DE2;
    double*         aMin;
    double*         aMinQ;


    printf("Running with\r\n");
    printf("------------\r\n");
    printf("QdL \t= %.15f\r\n", QdL);
    printf("QdH \t= %.15f\r\n", QdH);
    printf("NQd \t= %i\r\n", NQd);
    printf("g \t= %.15f\r\n", g);
    printf("xH \t= %.15f\r\n", xH);
    printf("Mesh \t= %i\r\n", iMesh);
    printf("DMode \t= %s\r\n", (m_DiagnMode == modeAnalyticalDiag)
                               ? "Analytical" : "Numerical");
```

```
    printf("Using binary search\r\n");
    printf("------------\r\n\r\n");


    aMin = new double[NQd];
    aMinQ = new double[NQd];

    Q1 = aMinQ[0] = QdL;
    Q2 = aMinQ[1] = QdH;
    DE1 = aMin[0] = PlotSingleDEvsQ(1, szPrefix, QdL, g, 25, xH,
                                    bCenterAtQd, iMesh);
    DE2 = aMin[1] = PlotSingleDEvsQ(2, szPrefix, QdH, g, 25, xH,
                                    bCenterAtQd, iMesh);

    for(int i=0; i<(NQd-2); i++)
    {
        Q = aMinQ[i+2] = (Q1 + Q2) / 2.0;
        DE = aMin[i+2] = PlotSingleDEvsQ(3+i, szPrefix, Q, g, 25, xH,
                                         bCenterAtQd, iMesh);

        if(DE < 0.0)
        {
            Q1 = Q; DE1 = DE;
        }
        else
        {
            Q2 = Q; DE2 = DE;
        }
    }

    printf("\r\nxmins%s={", szPrefix);
    for(int i=0; i<NQd; i++)
    {
        if(i!=0) printf(",");
        printf("{%.15f,%.15f}", aMinQ[i], aMin[i]);
    }
    printf("}\r\n\r\n\r\n");

    delete [] aMinQ;
    delete [] aMin;
}

double CBiLayerPhasePoint::PlotSingleDEvsQ(int iIndex, const char* szPrefix,
          double Qd, double g, int iPts, double Xmax, bool bCenterAtQd,
          int iMeshPts, bool bMeasureAccuracy)
{
    if(m_DiagnMode == modeNumericalDiag)
    {
        CExchEnergyND   ExchE(iMeshPts);
        CKinEnergyND    KinE;

        return PlotSingleDEvsQ(KinE, ExchE, iIndex, szPrefix, Qd, g, iPts,
                               Xmax, bCenterAtQd, iMeshPts, bMeasureAccuracy);
    }
```

```
    else if(m_DiagnMode == modeAnalyticalDiag)
    {
        CExchEnergyAD   ExchE(iMeshPts);
        CKinEnergyAD    KinE;

        return PlotSingleDEvsQ(KinE, ExchE, iIndex, szPrefix, Qd, g, iPts,
                               Xmax, bCenterAtQd, iMeshPts, bMeasureAccuracy);
    }

    return 0.0;
}

double CBiLayerPhasePoint::PlotSingleDEvsQ(CEnergy& KinE, CEnergy& ExchE,
            int iIndex, const char* szPrefix, double Qd, double g, int iPts,
            double Xmax, bool bCenterAtQd, int iMeshPts, bool bMeasureAccuracy)
{
    double*         pAccuracy = NULL;
    double          Qsu, Qsd, Qsu2;
    double          dRes, dx, x, xp;
    double          aE[2*iPts + 1];
    double          dMinE;
    int             c = 0;
    int             iPtsE = iPts;


    if(bMeasureAccuracy) pAccuracy = new double;
    if(bCenterAtQd) iPtsE = 0;

    printf("data%s%i = {", szPrefix, iIndex);

    dx = fabs(Xmax) / double(iPts);
    for(int j=-iPts; j<=iPtsE; j++)
    {
        xp = x = double(j) * dx;
        if(bCenterAtQd) x += Qd*Qd;

        if(x <= 0.0)
        {
            if(Qd > M_SQRT1_2) continue;

            Qsu = sqrt(2.0*Qd*Qd + fabs(x));
            Qsd = sqrt(fabs(x));

            dRes = ExchE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierTwoTypes,
                                   pAccuracy);
            dRes+= KinE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierTwoTypes);
        }
        else
        {
            Qsu2 = 2.0*Qd*Qd - x;
            if(Qsu2 > 0.0)
            {
                Qsu = sqrt(Qsu2);
                Qsd = sqrt(x);
```

```
                dRes = ExchE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierOneType,
                                       pAccuracy);
                dRes+= KinE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierOneType);

            } else continue;
        }

        if(j != -iPts) printf(", ");
        if(pAccuracy) printf("{%.15f, %.15f, %.15f}", xp, dRes, *pAccuracy);
        else          printf("{%.15f, %.15f}", xp, dRes);
        aE[c++] = dRes;
    }

    printf("};\r\n");
    printf("ListLinePlot[data%s%i]\r\n", szPrefix, iIndex);

    dMinE = FindMinimum(aE, dx, c);
    printf("MinE%s%i:=%.15f", szPrefix, iIndex, dMinE);
    printf("\r\n\r\n");

    if(pAccuracy) delete pAccuracy;

    return dMinE;
}

double CBiLayerPhasePoint::FindMinimum(double* aE, double& dx, int iSize)
{
    double  dMinE = DBL_MAX;
    double  a, b, c, Xm;
    int     iMin = 0;


    for(int i=0; i<iSize; i++)
    {
        if(aE[i] < dMinE)
        {
            dMinE = aE[i];
            iMin = i;
        }
    }

    if((iMin > 0) && (iMin < (iSize-1)))
    {
        a = 1.0 / (2.0*dx*dx) * (aE[iMin-1] - 2.0*aE[iMin] + aE[iMin+1]);
        b = 1.0 / (2.0*dx) * (aE[iMin+1] - aE[iMin-1]);
        c = aE[iMin];

        if(a == 0.0) dMinE = aE[iMin];
        else
        {
            Xm = -b / (2.0*a);
            dMinE = a*Xm*Xm + b*Xm + c;
        }
```

```
    }
    else
    {
        dMinE = aE[iMin];
    }

    return dMinE;
}
```

F.9 The main() entry point

Below is the main entry point of the program. This routine has as responsibility to interpret the command line parameters that come from the command line and use the classes of the program to calculate the requested parameters. To interpret the command line parameters, the `CCmdLineParser` class is used. Instances of the `CBiLayerPhasePoint` and `CBiLayerDispersion` classes provide the calculations that can be initiated from the command line.

```
#include "CmdLineParser.h"
#include "BiLayerPhasePoint.h"
#include "Timer.h"
#include "Dispersion.h"
#include <iostream>

int main(int argc, const char * argv[])
{
    CTimer               Timer;
    CCmdLineParser       CmdLine;
    CBiLayerPhasePoint   BiLayerPhasePt;
    CBiLayerDispersion   Dispersion;
    double               QdL = 0.0, QdH = 0.002;
    double               g = 6.0;
    double               xH = 0.005*0.005;
    bool                 bCenterAtQd = 0;
    bool                 bBinSearch = 0;
    int                  NQd = 5, iMesh = 30;
    int                  iAll = -1;
    int                  iSingle = -1;
    int                  iMode = 0;
    int                  iDMode = 0;


    if(!CmdLine.Parse(argc, argv)) return 0;
    for(int i=0; i<CmdLine.GetNumArgs(); i++)
    {
        // Mode
        if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Mode")==0)
        {
            iMode = (int)CmdLine.GetArg(i).m_dVal;
        }

        // Diagonalization mode
        if(strcmp(CmdLine.GetArg(i).m_cSwitch, "DMode")==0)
        {
            iDMode = (int)CmdLine.GetArg(i).m_dVal;
```

```
}

// Low Qd
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "QdL")==0)
{
    QdL = CmdLine.GetArg(i).m_dVal;
}

// High Qd
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "QdH")==0)
{
    QdH = CmdLine.GetArg(i).m_dVal;
}

// Step Qd
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "NQd")==0)
{
    NQd = (int)CmdLine.GetArg(i).m_dVal;
}

// Coupling g
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "g")==0)
{
    g = CmdLine.GetArg(i).m_dVal;
}

// High x
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "xH")==0)
{
    xH = CmdLine.GetArg(i).m_dVal;
}

// Center at Qd^2
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "CAQd")==0)
{
    bCenterAtQd = CmdLine.GetArg(i).m_dVal;
}

// Use binary search algorithm for a multiple DE vs Q plots
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "BinS")==0)
{
    bBinSearch = CmdLine.GetArg(i).m_dVal;
}

// Number of mesh points used in all directions
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Mesh")==0)
{
    iMesh = CmdLine.GetArg(i).m_dVal;
}

// Sequence through all points in phase diagram
if(strcmp(CmdLine.GetArg(i).m_cSwitch, "All")==0)
{
    iAll = CmdLine.GetArg(i).m_dVal;
```

```
    }

    // Plot single delta E versus x
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Single")==0)
    {
        iSingle = CmdLine.GetArg(i).m_dVal;
    }
}

BiLayerPhasePt.m_DiagnMode = (CBiLayerPhasePoint::EMode)iDMode;

// Mode : Calculate phase diagram
if(iMode == 0)
{
    Timer.Start();

    if(iAll > 0)
    {
        BiLayerPhasePt.PlotAllDEvsQ(bCenterAtQd, iMesh, iAll);
    }
    else if(iSingle == 1)
    {
        BiLayerPhasePt.PlotSingleDEvsQ(1, "S", QdL, g, 25, xH,
                                       bCenterAtQd, iMesh);
    }
    else
    {
        if(!bBinSearch)
            BiLayerPhasePt.PlotMultipleDEvsQ("A", QdL, QdH, NQd,
                                        g, xH, bCenterAtQd, iMesh);
        else
            BiLayerPhasePt.PlotBinSearchDEvsQ("A", QdL, QdH, NQd,
                                        g, xH, bCenterAtQd, iMesh);
    }

    Timer.Stop();
    Timer.PrintElapsed();
}

// Mode : Calculate full dispersion
if(iMode == 1)
{
    Dispersion.InitFullDispersion(-xH, xH);
    Dispersion.PrintFullDispersion();
}

// Mode : Calculate low energy 2D dispersion
if(iMode == 2)
{
    Dispersion.InitLin2DDispersion(-xH, xH);
    Dispersion.PrintLin2DDispersion();
}

// Mode : Calculate low energy 1D dispersion
```

```
    if(iMode == 3)
    {
        Dispersion.InitLin1DDispersion(-xH, xH);
        Dispersion.PrintLin1DDispersion();
    }

    // Mode : Calculate low energy diagonalization matrix
    if(iMode == 4)
    {
        Dispersion.InitLinDiagMatrix(-xH, xH);
        Dispersion.PrintLinDiagMatrix();
    }

    // Mode : Test accuracy of integrations
    if(iMode == 5)
    {
        Timer.Start();
        BiLayerPhasePt.PlotSingleDEvsQ(1, "S", QdL, g, 25, xH, iMesh, true);
        Timer.Stop();
        Timer.PrintElapsed();
    }

    // Mode : Calculate 1D density of states, band 1
    if(iMode == 7)
    {
        Dispersion.InitLin1DDensityOfStates(0.0, 1.5);
        Dispersion.Print1DDensityOfStates(1, 0.0, xH);
    }

    // Mode : Calculate 1D density of states (as function of p), band 1
    if(iMode == 8)
    {
        Dispersion.InitLin1DDensityOfStates(0.0, 1.5);
        Dispersion.Print1DDensityOfStates(1, 0.0, xH, true);
    }

    return 0;
}
```

APPENDIX G

ABC-Trilayer C++ code

In the following the code that is specific for ABC-stracked trilayer graphene is listed. This code is very similar to the code of appendix F for bilayer graphene i.e. except for different calculations within the functions, the function calls are the same and the differences have been explained in earlier chapters. Thus, if an explanation is needed for any of the functions then consult appendix F.

G.1 The CTriLayerDispersion class

Class header:

```
#include "Function.h"

class CTriLayerDispersion
{
public:
    CTriLayerDispersion();
    virtual ~CTriLayerDispersion();


public:
    void InitFullDispersion(double dKMin, double dKMax);
    void InitFull1DDispersion(double dKMin, double dKMax);
    void InitLin2DDispersion(double dKMin, double dKMax);
    void InitLin1DDispersion(double dKMin, double dKMax);
    void InitLinDiagMatrix(double dKMin, double dKMax);
    void InitLin1DDensityOfStates(double dKMin, double dKMax);
    CReFunc2D* GetFullDispersion(int iBand);
    CReFunc1D* GetFull1DDispersion(int iBand);
    CReFunc2D* GetLin2DDispersion(int iBand);
    CReFunc1D* GetLin1DDispersion(int iBand);
    CCplxFuncMatrix2D* GetLinDiagMatrix();
    void GetLinDiagMatrix(double p, double phi, CCplxMatrix& Md);
    double Get1DDensityOfStates(int iBand, double x, bool bXIsP=true);
    void PrintFullDispersion();
```

```
    void PrintFull1DDispersion();
    void PrintLin2DDispersion();
    void PrintLin1DDispersion();
    void PrintLinDiagMatrix();
    void Print1DDensityOfStates(int iBand, double dEMin, double dEMax,
                                bool bEIsP=false);


private:
    CReFunc2D*          m_pFullDispersion[6];
    CReFunc2D*          m_pLin2DDispersion[6];
    CReFunc1D*          m_pLin1DDispersion[6];
    CReFunc1D*          m_pFull1DDispersion[6];
    CCplxFuncMatrix2D*  m_pLinDiagMatrix;

    CCplxMatrix         H;
    CCplxMatrix         M;
    CCplxMatrix         E;
};
```

Class body:

```
#include "Dispersion.h"
#include <math.h>
#include <stdio.h>
#include <iostream>

CTriLayerDispersion::CTriLayerDispersion()
{
    for(int i=0; i<6; i++)
    {
        m_pFullDispersion[i] = NULL;
        m_pLin2DDispersion[i] = NULL;
        m_pLin1DDispersion[i] = NULL;
    }

    m_pLinDiagMatrix = NULL;

    H.SetSize(6, 6);
    M.SetSize(6, 6);
    E.SetSize(6, 6);
}

CTriLayerDispersion::~CTriLayerDispersion()
{
    for(int i=0; i<6; i++)
    {
        if(m_pFullDispersion[i]) delete m_pFullDispersion[i];
        if(m_pLin2DDispersion[i]) delete m_pLin2DDispersion[i];
        if(m_pLin1DDispersion[i]) delete m_pLin1DDispersion[i];
    }

    if(m_pLinDiagMatrix) delete m_pLinDiagMatrix;
}
```

```
void CTriLayerDispersion::PrintFullDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<6; i++)
    {
        if(m_pFullDispersion[i])
        {
            sprintf(szPrefix, "E%i", i);
            m_pFullDispersion[i]->Print(szPrefix);
        }
    }
}

void CTriLayerDispersion::PrintLin2DDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<6; i++)
    {
        if(m_pLin2DDispersion[i])
        {
            sprintf(szPrefix, "EL%i", i);
            m_pLin2DDispersion[i]->Print(szPrefix);
        }
    }
}

void CTriLayerDispersion::PrintLin1DDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<6; i++)
    {
        if(m_pLin1DDispersion[i])
        {
            sprintf(szPrefix, "EL%i", i);
            m_pLin1DDispersion[i]->Print(szPrefix);
        }
    }
}

void CTriLayerDispersion::PrintFull1DDispersion()
{
    char    szPrefix[5];

    for(int i=0; i<6; i++)
    {
        if(m_pFull1DDispersion[i])
        {
            sprintf(szPrefix, "EF%i", i);
            m_pFull1DDispersion[i]->Print(szPrefix);
        }
    }
```

```
}

void CTriLayerDispersion::PrintLinDiagMatrix()
{
    int     iColumns, iRows;
    char    szPrefix[10];


    if(!m_pLinDiagMatrix) return;

    m_pLinDiagMatrix->GetSize(iRows, iColumns);
    for(int m=0; m<iRows; m++)
    {
        for(int n=0; n<iColumns; n++)
        {
            sprintf(szPrefix, "Re%i%i", m, n);
            m_pLinDiagMatrix->Print(szPrefix, m, n,
                               CCplxFuncMatrix2D::printRe);
            sprintf(szPrefix, "Im%i%i", m, n);
            m_pLinDiagMatrix->Print(szPrefix, m, n,
                               CCplxFuncMatrix2D::printIm);
        }
    }
}

void CTriLayerDispersion::Print1DDensityOfStates(int iBand,
                          double dEMin, double dEMax, bool bEIsP)
{
    double  D;

    if(m_pLin1DDispersion[iBand])
    {
        printf("FDL%i={", iBand);
        for(double E=dEMin; E<dEMax; E+=(dEMax-dEMin)/50.0)
        {
            D = Get1DDensityOfStates(iBand, E, bEIsP);
            if(E != dEMin) printf(",");
            printf("{%.15f,%.15f}", E, D);
        }
        printf("};\r\n");
        printf("ListLinePlot[FDL%i]\r\n", iBand);
    }
}

void CTriLayerDispersion::InitFullDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    CCplx           C[2];
    double          x, y, DeltaX, DeltaY;
    double          v;
    const double    a = 1.5551;
    const double    sqrt3_2 = 0.8660254037844386;
    double          delta[3][2] = {{0.5*a, sqrt3_2*a},
                               {0.5*a, -sqrt3_2*a},{-a, 0}};
```

```
CReFunc2D*      F[6];
const double    t_perp = 0.05;
const double    t = 8.57*t_perp;


for(int i=0; i<6; i++)
{
    if(m_pFullDispersion[i]) delete m_pFullDispersion[i];
    F[i] = m_pFullDispersion[i] = new CReFunc2D(dKMin, dKMax, dKMin, dKMax);
}

DeltaX = F[0]->GetDeltaX();
DeltaY = F[0]->GetDeltaY();
N = F[0]->GetN();

for(int i=0; i<N; i++)
{
    for(int j=0; j<N; j++)
    {
        x = dKMin + double(i)*DeltaX;
        y = dKMin + double(j)*DeltaY;

        H.M[0][2].x = -t_perp;
        H.M[2][0].x = -t_perp;
        H.M[3][5].x = -t_perp;
        H.M[5][3].x = -t_perp;

        H.M[0][1].Set(0.0, 0.0);
        H.M[1][0].Set(0.0, 0.0);

        H.M[2][3].Set(0.0, 0.0);
        H.M[3][2].Set(0.0, 0.0);

        H.M[4][5].Set(0.0, 0.0);
        H.M[5][4].Set(0.0, 0.0);

        for(int k=0; k<3; k++)
        {
            v = x*delta[k][0] + y*delta[k][1];
            C[0].SetEuler(-t, v);
            C[1].SetEuler(-t, -v);

            H.M[0][1] += C[1];
            H.M[1][0] += C[0];

            H.M[2][3] += C[0];
            H.M[3][2] += C[1];

            H.M[4][5] += C[1];
            H.M[5][4] += C[0];
        }

        H.DiagJacobiCplx(E, M, 30, NRes);
```

```
            F[0]->SetF(i, j, E.M[0][0].x);
            F[1]->SetF(i, j, E.M[1][1].x);
            F[2]->SetF(i, j, E.M[2][2].x);
            F[3]->SetF(i, j, E.M[3][3].x);
            F[4]->SetF(i, j, E.M[4][4].x);
            F[5]->SetF(i, j, E.M[5][5].x);
        }
    }
}

void CTriLayerDispersion::InitFull1DDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    CCplx           C[2];
    double          x, DeltaX;
    double          v;
    const double    a = 1.5551;
    const double    sqrt3_2 = 0.8660254037844386;
    double          delta[3][2] = {{0.5*a, sqrt3_2*a},
                                   {0.5*a, -sqrt3_2*a},{-a, 0}};
    CReFunc1D*      F[6];
    const double    Ky = 1.5551203015562138;
    const double    t_perp = 0.05;
    const double    t = 8.57*t_perp;


    for(int i=0; i<6; i++)
    {
        if(m_pFull1DDispersion[i]) delete m_pFull1DDispersion[i];
        F[i] = m_pFull1DDispersion[i] = new CReFunc1D(dKMin, dKMax);
    }

    DeltaX = F[0]->GetDeltaX();
    N = F[0]->GetN();

    for(int i=0; i<N; i++)
    {
        x = dKMin + double(i)*DeltaX;

        H.M[0][2].x = -t_perp;
        H.M[2][0].x = -t_perp;
        H.M[3][5].x = -t_perp;
        H.M[5][3].x = -t_perp;

        H.M[0][1].Set(0.0, 0.0);
        H.M[1][0].Set(0.0, 0.0);

        H.M[2][3].Set(0.0, 0.0);
        H.M[3][2].Set(0.0, 0.0);

        H.M[4][5].Set(0.0, 0.0);
        H.M[5][4].Set(0.0, 0.0);

        for(int k=0; k<3; k++)
```

```
        {
            v = x*delta[k][0] + Ky*delta[k][1];
            C[0].SetEuler(-t, v);
            C[1].SetEuler(-t, -v);

            H.M[0][1] += C[1];
            H.M[1][0] += C[0];

            H.M[2][3] += C[0];
            H.M[3][2] += C[1];

            H.M[4][5] += C[1];
            H.M[5][4] += C[0];
        }

        H.DiagJacobiCplx(E, M, 30, NRes);

        F[0]->SetF(i, E.M[0][0].x);
        F[1]->SetF(i, E.M[1][1].x);
        F[2]->SetF(i, E.M[2][2].x);
        F[3]->SetF(i, E.M[3][3].x);
        F[4]->SetF(i, E.M[4][4].x);
        F[5]->SetF(i, E.M[5][5].x);
    }
}

void CTriLayerDispersion::InitLin2DDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    double          x, y, DeltaX, DeltaY;
    CReFunc2D*      F[6];
    const double    t_perp = 0.05;


    for(int i=0; i<6; i++)
    {
        if(m_pLin2DDispersion[i]) delete m_pLin2DDispersion[i];
        F[i] = m_pLin2DDispersion[i] = new CReFunc2D(dKMin, dKMax, dKMin, dKMax);
    }

    DeltaX = F[0]->GetDeltaX();
    DeltaY = F[0]->GetDeltaY();
    N = F[0]->GetN();

    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            x = dKMin + double(i)*DeltaX;
            y = dKMin + double(j)*DeltaY;

            H.M[0][2].x = -t_perp;
            H.M[2][0].x = -t_perp;
            H.M[3][5].x = -t_perp;
```

```
            H.M[5][3].x = -t_perp;

            H.M[0][1].Set(y, -x);
            H.M[1][0].Set(y, x);

            H.M[2][3].Set(y, x);
            H.M[3][2].Set(y, -x);

            H.M[4][5].Set(y, -x);
            H.M[5][4].Set(y, x);

            H.DiagJacobiCplx(E, M, 30, NRes);

            F[0]->SetF(i, j, E.M[0][0].x);
            F[1]->SetF(i, j, E.M[1][1].x);
            F[2]->SetF(i, j, E.M[2][2].x);
            F[3]->SetF(i, j, E.M[3][3].x);
            F[4]->SetF(i, j, E.M[4][4].x);
            F[5]->SetF(i, j, E.M[5][5].x);
        }
    }
}

void CTriLayerDispersion::InitLin1DDispersion(double dKMin, double dKMax)
{
    int             N, NRes;
    double          x, DeltaX;
    CReFunc1D*      F[6];
    const double    t_perp = 0.05;


    for(int i=0; i<6; i++)
    {
        if(m_pLin1DDispersion[i]) delete m_pLin1DDispersion[i];
        F[i] = m_pLin1DDispersion[i] = new CReFunc1D(dKMin, dKMax);
    }

    DeltaX = F[0]->GetDeltaX();
    N = F[0]->GetN();

    for(int i=0; i<N; i++)
    {
        x = dKMin + double(i)*DeltaX;

        H.M[0][2].x = -t_perp;
        H.M[2][0].x = -t_perp;
        H.M[3][5].x = -t_perp;
        H.M[5][3].x = -t_perp;

        H.M[0][1].Set(0.0, -x);
        H.M[1][0].Set(0.0, x);

        H.M[2][3].Set(0.0, x);
        H.M[3][2].Set(0.0, -x);
```

```
        H.M[4][5].Set(0.0, -x);
        H.M[5][4].Set(0.0, x);

        H.DiagJacobiCplx(E, M, 30, NRes);

        F[0]->SetF(i, E.M[0][0].x);
        F[1]->SetF(i, E.M[1][1].x);
        F[2]->SetF(i, E.M[2][2].x);
        F[3]->SetF(i, E.M[3][3].x);
        F[4]->SetF(i, E.M[4][4].x);
        F[5]->SetF(i, E.M[5][5].x);
    }
}

void CTriLayerDispersion::InitLinDiagMatrix(double dKMin, double dKMax)
{
    int                 N, NRes;
    double              x, y, DeltaX, DeltaY;
    CCplxMatrix         Md(6, 6);
    CCplxFuncMatrix2D*  F;
    const double        t_perp = 0.05;
    const int           C[6] = {2, 1, 3, 0};


    if(m_pLinDiagMatrix) delete m_pLinDiagMatrix;
    F = m_pLinDiagMatrix = new CCplxFuncMatrix2D(dKMin, dKMax, 0.0, 2.0*M_PI, 30);

    DeltaX = F->GetDeltaX();
    DeltaY = F->GetDeltaY();
    N = F->GetN();

    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            x = dKMin + double(i)*DeltaX;
            y = double(j)*DeltaY;

            H.M[0][2].x = -t_perp;
            H.M[2][0].x = -t_perp;
            H.M[3][5].x = -t_perp;
            H.M[5][3].x = -t_perp;

            H.M[0][1].SetEuler(x, -y);
            H.M[1][0].SetEuler(x, y);

            H.M[2][3].SetEuler(x, y);
            H.M[3][2].SetEuler(x, -y);

            H.M[4][5].SetEuler(x, -y);
            H.M[5][4].SetEuler(x, y);

            if(!H.DiagJacobiCplx(E, M, 5000, NRes))
```

```
                printf("Error Diagonalizing\r\n");

            M.StandardizeDiagonalizer();

            for(int i=0; i<6; i++)
            {
                for(int j=0; j<6; j++)
                {
                    Md.M[i][C[j]].x = M.M[i][j].x;
                    Md.M[i][C[j]].y = M.M[i][j].y;
                }
            }

            F->SetF(i, j, Md);
        }
    }
}

void CTriLayerDispersion::InitLin1DDensityOfStates(double dKMin, double dKMax)
{
    InitLin1DDispersion(dKMin, dKMax);
}

CReFunc2D* CTriLayerDispersion::GetFullDispersion(int iBand)
{
    return m_pFullDispersion[iBand];
}

CReFunc1D* CTriLayerDispersion::GetFull1DDispersion(int iBand)
{
    return m_pFull1DDispersion[iBand];
}

CReFunc2D* CTriLayerDispersion::GetLin2DDispersion(int iBand)
{
    return m_pLin2DDispersion[iBand];
}

CReFunc1D* CTriLayerDispersion::GetLin1DDispersion(int iBand)
{
    return m_pLin1DDispersion[iBand];
}

CCplxFuncMatrix2D* CTriLayerDispersion::GetLinDiagMatrix()
{
    return m_pLinDiagMatrix;
}

void CTriLayerDispersion::GetLinDiagMatrix(double p, double phi, CCplxMatrix& Md)
{
    const double        t_perp = 0.05;
    int                 NRes;
```

```
    H.M[0][2].x = -t_perp;
    H.M[2][0].x = -t_perp;
    H.M[3][5].x = -t_perp;
    H.M[5][3].x = -t_perp;

    H.M[0][1].SetEuler(p, -phi);
    H.M[1][0].SetEuler(p, phi);

    H.M[2][3].SetEuler(p, phi);
    H.M[3][2].SetEuler(p, -phi);

    H.M[4][5].SetEuler(p, -phi);
    H.M[5][4].SetEuler(p, phi);

    if(!H.DiagJacobiCplx(E, Md, 5000, NRes)) printf("Error Diagonalizing!\r\n");
}

double CTriLayerDispersion::Get1DDensityOfStates(int iBand, double x, bool bXIsP)
{
    CReFunc1D*  E = GetLin1DDispersion(iBand);
    double      depsilon = 1E-10;
    double      epsilon = bXIsP ? E->GetF(x) : x;
    double      k1 = E->GetFInv(epsilon + depsilon);
    double      k2 = E->GetFInv(epsilon);

    return  (k1*k1 - k2*k2) / depsilon;
}
```

G.2 The CEnergy class

Class header:

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CEnergy
{
public:
    enum ECarriers { carrierOneType=1, carrierTwoTypes };

public:
    CEnergy() { t_perp = 0.05; }

public:
    virtual double Calculate(double Qd, double Qsu, double Qsd,
                  double g, ECarriers Carriers, double* pAccuracy=0) = 0;

protected:
    double t_perp;
};
```

G.3 The CExchEnergy class

Class header:

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CExchEnergy : public CEnergy, public CIntegralDE3D
{
public:
    CExchEnergy();
    CExchEnergy(int N);


public:
    double Calculate(double Qd, double Qsu, double Qsd, double g,
                     ECarriers Carriers, double* pAccuracy=0);
    double EvalIntegrand(double k, double p, double phi, double Qd,
                         double Qsu, double Qsd, double g, int iPart);

protected:
    double f(double k, double p, double phi);

protected:
    virtual double F(int i, int j, double k, double p, double phi, double g) = 0;
    virtual int Band(int iNumber) = 0;
    double V(double k, double p, double phi);
    void Init(void);


protected:
    int                m_iPart;
    double             m_Qd;
    double             m_Qsu;
    double             m_Qsd;
    double             m_g;
};
```

Class body:

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>


CExchEnergy::CExchEnergy() : CIntegralDE3D(30), CEnergy()
{
    CExchEnergy::Init();
}

CExchEnergy::CExchEnergy(int N) : CIntegralDE3D(N), CEnergy()
{
    CExchEnergy::Init();
```

```
}

void CExchEnergy::Init()
{
    m_Qd = 0.0;
    m_Qsu = 0.5;
    m_Qsd = 0.5;
    m_g = 6.0;

    m_iPart = 0;
}

double CExchEnergy::Calculate(double Qd, double Qsu, double Qsd, double g,
                              ECarriers Carriers, double* pAccuracy)
{
    double              dRes = 0.0;


    m_Qd = Qd;
    m_Qsu = Qsu;
    m_Qsd = Qsd;
    m_g = g;

    if(pAccuracy)
    {
        m_iPart = 1;
        TestAccuracy(*pAccuracy);
    }

    if(Carriers == carrierTwoTypes)
    {
        m_iPart = 1;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 2;
        dRes += (Qsu != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 3;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 4;
        dRes += (Qsd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 5;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 6;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 7;
        dRes += CIntegralDE3D::Integrate();
    }

    if(Carriers == carrierOneType)
```

```
    {
        m_iPart = 8;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 2;
        dRes += (Qsu != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 3;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 9;
        dRes += (Qsd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 5;
        dRes += CIntegralDE3D::Integrate();

        m_iPart = 6;
        dRes += (Qd != 0.0) ? CIntegralDE3D::Integrate() : 0.0;

        m_iPart = 10;
        dRes += CIntegralDE3D::Integrate();
    }

    return dRes;
}

double CExchEnergy::f(double k, double p, double phi)
{
    double  dRes = 0.0;
    double  e;
    int     a, b, c, d;


    a = Band(1);
    b = Band(2);
    c = Band(4);
    d = Band(6);

    k = 0.5*(k + 1.0);

    e = M_PI;
    phi = e*(phi + 1.0);

    switch(m_iPart)
    {
        case 1:
            p = 0.5*(p + 1.0);
            dRes = (2.0 * m_Qsu * (F(a, a, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                 + F(a, b, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                 + F(a, c, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                 + F(a, d, m_Qsu*k, m_Qsu*k*p, phi, m_g))
                    - 2.0 * m_Qsd * F(b, c, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                    - 2.0 * m_Qsd * F(b, d, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                    - 4.0 * m_Qd *  (F(a, a, m_Qd*k, m_Qd*k*p, phi, m_g)
```

```
                         + F(a, b, m_Qd*k, m_Qd*k*p, phi, m_g)
                         + F(a, c, m_Qd*k, m_Qd*k*p, phi, m_g)
                         + F(a, d, m_Qd*k, m_Qd*k*p, phi, m_g)))
                         *V(k, p, phi);

    dRes*= 0.5;
    break;

case 2:
    e = 1.0 / m_Qsu;
    p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
    dRes = 2.0 * m_Qsu * (F(a, b, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                        +F(a, c, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                        +F(a, d, m_Qsu*k, m_Qsu*k*p, phi, m_g))
                        *V(k, p, phi);
    dRes*= (e - 1.0) / 2.0;
    break;

case 3:
    e = 1.0 / m_Qd;
    p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
    dRes = -4.0 * m_Qd * (F(a, b, m_Qd*k, m_Qd*k*p, phi, m_g)
                        +F(a, c, m_Qd*k, m_Qd*k*p, phi, m_g)
                        +F(a, d, m_Qd*k, m_Qd*k*p, phi, m_g))
                        *V(k, p, phi);
    dRes*= (e - 1.0) / 2.0;
    break;

case 4:
    e = 1.0 / m_Qsd;
    p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
    dRes = -2.0 * m_Qsd * (F(b, b, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                         +F(b, c, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                         +F(b, d, m_Qsd*k, m_Qsd*k*p, phi, m_g))
                         *V(k, p, phi);
    dRes*= (e - 1.0) / 2.0;
    break;

case 5:
    e = m_Qsu / 2.0;
    p = e*(p + 1.0);
    dRes = 2.0 * (F(a, b, k*p, k, phi, m_g)
                +F(a, c, k*p, k, phi, m_g)
                +F(a, d, k*p, k, phi, m_g))*V(k, p, phi);
    dRes*= e;
    break;

case 6:
    e = m_Qd / 2.0;
    p = e*(p + 1.0);
    dRes = -4.0 * (F(a, b, k*p, k, phi, m_g)
                 +F(a, c, k*p, k, phi, m_g)
                 +F(a, d, k*p, k, phi, m_g))*V(k, p, phi);
    dRes*= e;
```

```
            break;

        case 7:
            e = m_Qsd / 2.0;
            p = e*(p + 1.0);
            dRes = -2.0 * (F(b, b, k*p, k, phi, m_g)
                          +F(b, c, k*p, k, phi, m_g)
                          +F(b, d, k*p, k, phi, m_g))*V(k, p, phi);
            dRes*= e;
            break;

        case 8:
            p = 0.5*(p + 1.0);
            dRes = (2.0 * m_Qsu * (F(a, a, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                  +F(a, b, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                  +F(a, c, m_Qsu*k, m_Qsu*k*p, phi, m_g)
                                  +F(a, d, m_Qsu*k, m_Qsu*k*p, phi, m_g))
                    + 2.0 * m_Qsd * (F(a, a, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                    +F(a, b, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                    +F(a, c, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                    +F(a, d, m_Qsd*k, m_Qsd*k*p, phi, m_g))
                    - 4.0 * m_Qd *  (F(a, a, m_Qd*k, m_Qd*k*p, phi, m_g)
                                    +F(a, b, m_Qd*k, m_Qd*k*p, phi, m_g)
                                    +F(a, c, m_Qd*k, m_Qd*k*p, phi, m_g)
                                    +F(a, d, m_Qd*k, m_Qd*k*p, phi, m_g)))
                                    *V(k, p, phi);
            dRes*= 0.5;
            break;

        case 9:
            e = 1.0 / m_Qsd;
            p = (e - 1.0) / 2.0 * (p + (e + 1.0)/(e - 1.0));
            dRes = 2.0 * m_Qsd * (F(a, b, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                 +F(a, c, m_Qsd*k, m_Qsd*k*p, phi, m_g)
                                 +F(a, d, m_Qsd*k, m_Qsd*k*p, phi, m_g))
                                 *V(k, p, phi);
            dRes*= (e - 1.0) / 2.0;
            break;

        case 10:
            e = m_Qsd / 2.0;
            p = e*(p + 1.0);
            dRes = 2.0 * (F(a, b, k*p, k, phi, m_g)
                         +F(a, c, k*p, k, phi, m_g)
                         +F(a, d, k*p, k, phi, m_g))*V(k, p, phi);
            dRes*= e;
            break;
    }

    dRes*= 0.5;
    dRes*= M_PI;

    return dRes;
}
```

```
double CExchEnergy::V(double k, double p, double phi)
{
    double  A = p*p - 2*p*cos(phi) + 1.0;
    double  B;


    B = sqrt((A < 0.0) ? 0.0 : A);
    if(B <= 0.0) return 0.0;

    return 1.0 / B;
}

double CExchEnergy::EvalIntegrand(double k, double p, double phi, double Qd,
                                  double Qsu, double Qsd, double g, int iPart)
{
    m_Qd = Qd;
    m_Qsu = Qsu;
    m_Qsd = Qsd;
    m_g = g;

    m_iPart = iPart;

    return f(k, p, phi);
}
```

G.4 The CExchEnergyND class

Class header:

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CExchEnergyND : public CExchEnergy
{
public:
    CExchEnergyND();
    CExchEnergyND(int N);


private:
    double F(int i, int j, double k, double p, double phi, double g);
    int Band(int iNumber);
    void Init();


private:
    CTriLayerDispersion m_Dispersion;
    CCplxMatrix         M_p;
    CCplxMatrix         M_k;
    CCplxMatrix         M_pT;
    CCplxMatrix         M_kT;
    CCplxMatrix         IM[5];
```

```
    CCplxMatrix         Chi_pk[6];
    CCplxMatrix         Chi_kp[6];
};
```

Class body:

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>

CExchEnergyND::CExchEnergyND() : CExchEnergy()
{
    CExchEnergyND::Init();
}

CExchEnergyND::CExchEnergyND(int N) : CExchEnergy(N)
{
    CExchEnergyND::Init();
}

void CExchEnergyND::Init()
{
    M_p.SetSize(6, 6);
    M_k.SetSize(6, 6);
    M_pT.SetSize(6, 6);
    M_kT.SetSize(6, 6);
    for(int i=0; i<5; i++)
        IM[i].SetSize(6, 6);

    Chi_pk[0].SetSize(6, 6);
    Chi_pk[1].SetSize(6, 6);
    Chi_pk[2].SetSize(6, 6);
    Chi_pk[3].SetSize(6, 6);
    Chi_pk[4].SetSize(6, 6);
    Chi_pk[5].SetSize(6, 6);
    Chi_kp[0].SetSize(6, 6);
    Chi_kp[1].SetSize(6, 6);
    Chi_kp[2].SetSize(6, 6);
    Chi_kp[3].SetSize(6, 6);
    Chi_kp[4].SetSize(6, 6);
    Chi_kp[5].SetSize(6, 6);

    IM[0].M[0][0].x = -1.0;
    IM[0].M[1][1].x = -1.0;
    IM[0].M[4][4].x = 1.0;
    IM[0].M[5][5].x = 1.0;

    IM[1].M[0][0].x = 1.0;
    IM[1].M[1][1].x = 1.0;
    IM[1].M[4][4].x = 1.0;
    IM[1].M[5][5].x = 1.0;

    IM[2].M[0][0].x = M_SQRT1_2;
    IM[2].M[1][1].x = M_SQRT1_2;
```

```
    IM[2].M[2][2].x = -1.0;
    IM[2].M[3][3].x = -1.0;
    IM[2].M[4][4].x = M_SQRT1_2;
    IM[2].M[5][5].x = M_SQRT1_2;

    IM[3].M[0][0].x = M_SQRT1_2;
    IM[3].M[1][1].x = M_SQRT1_2;
    IM[3].M[2][2].x = 1.0;
    IM[3].M[3][3].x = 1.0;
    IM[3].M[4][4].x = M_SQRT1_2;
    IM[3].M[5][5].x = M_SQRT1_2;

    IM[4].M[2][2].x = M_SQRT2;
    IM[4].M[3][3].x = M_SQRT2;
}

double CExchEnergyND::F(int i, int j, double k, double p, double phi, double g)
{
    const double        d16PiSquareI = 0.006332573977646111;
    const double        d = 3.7;
    const double        vF = 1.0;
    double              k2;
    double              p2;
    double              A, B1, B2, C;
    double              V_NS[6];
    double              dRet;


    k2 = k*k;
    p2 = p*p;

    m_Dispersion.GetLinDiagMatrix(k, phi, M_k);
    m_Dispersion.GetLinDiagMatrix(p, 0.0, M_p);

    M_kT = M_k.ConjTranspose();
    Chi_kp[0] = M_kT*IM[0]*M_p;
    Chi_kp[1] = M_kT*IM[2]*M_p;
    Chi_kp[2] = M_kT*IM[3]*M_p;
    Chi_kp[4] = M_kT*IM[0]*M_p;
    Chi_kp[5] = M_kT*IM[1]*M_p;
    Chi_pk[0] = Chi_kp[0].ConjTranspose();
    Chi_pk[1] = Chi_kp[1].ConjTranspose();
    Chi_pk[2] = Chi_kp[2].ConjTranspose();
    Chi_pk[4] = Chi_kp[4].ConjTranspose();
    Chi_pk[5] = Chi_kp[5].ConjTranspose();

    C = p2 + k2 - 2*p*k*cos(phi);
    A = sqrt((C < 0.0) ? 0.0 : C);
    B1 = exp(-d*A);
    B2 = exp(-2.0*d*A);
    V_NS[0] = 1.0;
    V_NS[1] = 1.0 - M_SQRT2*B1;
    V_NS[2] = 1.0 + M_SQRT2*B1;
    V_NS[3] = 0.0;
```

```
    V_NS[4] = -B2;
    V_NS[5] = B2;


    dRet = 0.0;
    for(int iAlpha=0; iAlpha<6; iAlpha++)
    {
        dRet += (Chi_kp[iAlpha].M[i][j]*Chi_pk[iAlpha].M[j][i]).Re()*V_NS[iAlpha];
    }

    return -d16PiSquareI * g * vF * p * k * dRet;
}

int CExchEnergyND::Band(int iNumber)
{
    if(iNumber == 1) return 2;
    if(iNumber == 2) return 3;
    if(iNumber == 3) return 1;
    if(iNumber == 4) return 4;
    if(iNumber == 5) return 0;
    if(iNumber == 6) return 5;

    return -1;
}
```

G.5 The CKinEnergyND class

Class header:

```
#include "IntegralDE.h"
#include "Matrix.h"
#include "Dispersion.h"

class CKinEnergyND : public CEnergy, public CIntegralDE1D
{
public:
    CKinEnergyND();


public:
    double Calculate(double Qd, double Qsu, double Qsd, double g,
                     ECarriers Carriers, double* pAccuracy=0);

protected:
    double f(double x);

private:
    double F(double E);
    double N1(double epsilon);


private:
    CTriLayerDispersion m_Dispersion;
    double              m_dE;
```

```
};
```

Class body:

```
#include "Energies.h"
#include <math.h>
#include <float.h>
#include <iostream>

CKinEnergyND::CKinEnergyND() : CEnergy()
{
    m_Dispersion.InitLin1DDispersion(0.0, 1.5);
    m_dE = 0.0;
}

double CKinEnergyND::Calculate(double Qd, double Qsu, double Qsd, double g,
                               ECarriers Carriers, double* pAccuracy)
{
    CReFunc1D*      E1 = m_Dispersion.GetLin1DDispersion(2);
    double          E1_Qsu = E1->GetF(Qsu);
    double          E1_Qsd = E1->GetF(Qsd);
    double          E1_Qd = E1->GetF(Qd);

    return F(E1_Qsu) + F(E1_Qsd) - 2.0*F(E1_Qd);
}

double CKinEnergyND::N1(double epsilon)
{
    double          k;
    CReFunc1D*      E1 = m_Dispersion.GetLin1DDispersion(2);
    const double    d4PiInv = 0.07957747154594767;

    k = E1->GetFInv(epsilon, true);
    return d4PiInv * k*k;
}

double CKinEnergyND::f(double x)
{
    double          epsilon = 0.5*m_dE*(x + 1.0);

    return 0.5 * m_dE * N1(epsilon);
}

double CKinEnergyND::F(double E)
{
    m_dE = E;
    return E*N1(E) - Integrate();
}
```

G.6 The CTriLayerPhasePoint class

Class header:

```
#include "Energies.h"
```

```
class CTriLayerPhasePoint
{
public:
    enum EMode { modeNumericalDiag = 1 };

public:
    CTriLayerPhasePoint() { m_DiagnMode = modeNumericalDiag; m_iPtsDEvsQ = 25; }


public:
    double PlotSingleDEvsQ(int iIndex, const char* szPrefix,
              double Qd, double g, int iPts, double Xmax,
              bool bCenterAtQd, int iMeshPts=30, bool bMeasureAccuracy=false);
    void PlotMultipleDEvsQ(const char* szPrefix, double QdL, double QdH,
              int NQd, double g, double xH, bool bCenterAtQd, int iMesh);
    void PlotBinSearchDEvsQ(const char* szPrefix, double QdL, double QdH,
              int NQd, double g, double xH, bool bCenterAtQd, int iMesh);
    void PlotAllDEvsQ(bool bCenterAtQd, int iMesh, int iSequence);

private:
    double FindMinimum(double* aE, double& dx, int iSize);
    double PlotSingleDEvsQ(CEnergy& KinE, CEnergy& ExchE, int iIndex,
             const char* szPrefix, double Qd, double g, int iPts,
             double Xmax, bool bCenterAtQd, int iMeshPts, bool bMeasureAccuracy);


public:
    EMode m_DiagnMode;
    int m_iPtsDEvsQ;
};
```

Class body:

```
#include "TriLayerPhasePoint.h"
#include <math.h>
#include <float.h>
#include <iostream>

void CTriLayerPhasePoint::PlotAllDEvsQ(bool bCenterAtQd, int iMesh, int iSequence)
{
    if(iSequence == 1)
    {
        PlotMultipleDEvsQ("1", 0.0, 0.1, 6, 1.0, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("2", 0.0, 0.1, 6, 2.0, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("3", 0.0, 0.1, 6, 3.0, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("4", 0.0, 0.1, 6, 4.0, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("5", 0.0, 0.1, 6, 5.0, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("6", 0.0, 0.1, 6, 6.0, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("010", 0.0, 0.1, 6, 0.1, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("025", 0.0, 0.1, 6, 0.25, 0.01, bCenterAtQd, iMesh);
        PlotMultipleDEvsQ("050", 0.0, 0.1, 6, 0.5, 0.01, bCenterAtQd, iMesh);
    }
    else if(iSequence == 2)
    {
```

```
    PlotMultipleDEvsQ("1", 0.0, 0.1, 6, 1.0, 0.01, bCenterAtQd, iMesh);
    PlotMultipleDEvsQ("3", 0.0, 0.1, 6, 3.0, 0.01, bCenterAtQd, iMesh);
    PlotMultipleDEvsQ("5", 0.0, 0.1, 6, 5.0, 0.01, bCenterAtQd, iMesh);
}
else if(iSequence == 3)
{
    PlotMultipleDEvsQ("2", 0.0, 0.1, 6, 2.0, 0.01, bCenterAtQd, iMesh);
    PlotMultipleDEvsQ("4", 0.0, 0.1, 6, 4.0, 0.01, bCenterAtQd, iMesh);
    PlotMultipleDEvsQ("6", 0.0, 0.1, 6, 6.0, 0.01, bCenterAtQd, iMesh);
}
else if(iSequence == 4)
{
    PlotMultipleDEvsQ("010", 0.0, 0.1, 6, 0.1, 0.01, bCenterAtQd, iMesh);
    PlotMultipleDEvsQ("025", 0.0, 0.1, 6, 0.25, 0.01, bCenterAtQd, iMesh);
    PlotMultipleDEvsQ("050", 0.0, 0.1, 6, 0.5, 0.01, bCenterAtQd, iMesh);
}
else if(iSequence == 5)
{
    PlotMultipleDEvsQ("1", 0.0, 0.06, 20, 1.0, 0.002, bCenterAtQd, iMesh);
}
else if(iSequence == 6)
{
    PlotMultipleDEvsQ("6", 0.0, 0.06, 20, 6.0, 0.002, bCenterAtQd, iMesh);
}
else if(iSequence == 7)
{
    PlotMultipleDEvsQ("6b", 0.00325, 0.06325, 20, 6.0, 0.002, bCenterAtQd,
                      iMesh);
}
else if(iSequence == 8)
{
    PlotBinSearchDEvsQ("1", 0.0, 0.02, 8, 1.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("2", 0.0, 0.02, 8, 2.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("3", 0.0, 0.02, 8, 3.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("4", 0.0, 0.02, 8, 4.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("5", 0.0, 0.02, 8, 5.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("6", 0.0, 0.02, 8, 6.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("010", 0.0, 0.02, 8, 0.1, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("025", 0.0, 0.02, 8, 0.25, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("050", 0.0, 0.02, 8, 0.5, 0.0005, bCenterAtQd, iMesh);
}
else if(iSequence == 9)
{
    PlotBinSearchDEvsQ("1", 0.0, 0.02, 8, 1.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("3", 0.0, 0.02, 8, 3.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("5", 0.0, 0.02, 8, 5.0, 0.0005, bCenterAtQd, iMesh);
}
else if(iSequence == 10)
{
    PlotBinSearchDEvsQ("2", 0.0, 0.02, 8, 2.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("4", 0.0, 0.02, 8, 4.0, 0.0005, bCenterAtQd, iMesh);
    PlotBinSearchDEvsQ("6", 0.0, 0.02, 8, 6.0, 0.0005, bCenterAtQd, iMesh);
}
else if(iSequence == 11)
```

```
    {
        PlotBinSearchDEvsQ("010", 0.0, 0.02, 8, 0.1, 0.0005, bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("025", 0.0, 0.02, 8, 0.25, 0.0005, bCenterAtQd, iMesh);
        PlotBinSearchDEvsQ("050", 0.0, 0.02, 8, 0.5, 0.0005, bCenterAtQd, iMesh);
    }
    else
    {
        printf("Sequence does not exist!\r\n");
    }
}

void CTriLayerPhasePoint::PlotMultipleDEvsQ(const char* szPrefix,
           double QdL, double QdH, int NQd, double g, double xH,
           bool bCenterAtQd, int iMesh)
{
    double          dQd, Qd;
    double*         aMin;


    printf("Running with\r\n");
    printf("------------\r\n");
    printf("QdL \t= %.15f\r\n", QdL);
    printf("QdH \t= %.15f\r\n", QdH);
    printf("NQd \t= %i\r\n", NQd);
    printf("g \t= %.15f\r\n", g);
    printf("xH \t= %.15f\r\n", xH);
    printf("Mesh \t= %i\r\n", iMesh);
    printf("DMode \t= %s\r\n", (m_DiagnMode != modeNumericalDiag)
                            ? "Analytical" : "Numerical");
    printf("------------\r\n\r\n");


    aMin = new double[NQd];

    dQd = (QdH - QdL) / double(NQd - 1);
    for(int i=0; i<NQd; i++)
    {
        Qd = QdL + dQd * double(i);

        aMin[i] = PlotSingleDEvsQ(i+1, szPrefix, Qd, g, m_iPtsDEvsQ, xH,
                                 bCenterAtQd, iMesh);
    }

    printf("\r\nxmins%s={", szPrefix);
    for(int i=0; i<NQd; i++)
    {
        Qd = QdL + dQd * double(i);
        if(i!=0) printf(",");
        printf("{%.15f,%.15f}", Qd, aMin[i]);
    }
    printf("}\r\n\r\n\r\n");

    delete [] aMin;
}
```

```
void CTriLayerPhasePoint::PlotBinSearchDEvsQ(const char* szPrefix,
           double QdL, double QdH, int NQd, double g, double xH,
           bool bCenterAtQd, int iMesh)
{
    double          Q, Q1, Q2;
    double          DE, DE1, DE2;
    double*         aMin;
    double*         aMinQ;


    printf("Running with\r\n");
    printf("------------\r\n");
    printf("QdL \t= %.15f\r\n", QdL);
    printf("QdH \t= %.15f\r\n", QdH);
    printf("NQd \t= %i\r\n", NQd);
    printf("g \t= %.15f\r\n", g);
    printf("xH \t= %.15f\r\n", xH);
    printf("Mesh \t= %i\r\n", iMesh);
    printf("DMode \t= %s\r\n", (m_DiagnMode != modeNumericalDiag)
                            ? "Analytical" : "Numerical");
    printf("Using binary search\r\n");
    printf("------------\r\n\r\n");


    aMin = new double[NQd];
    aMinQ = new double[NQd];

    Q1 = aMinQ[0] = QdL;
    Q2 = aMinQ[1] = QdH;
    DE1 = aMin[0] = PlotSingleDEvsQ(1, szPrefix, QdL, g, m_iPtsDEvsQ, xH,
                                   bCenterAtQd, iMesh);
    DE2 = aMin[1] = PlotSingleDEvsQ(2, szPrefix, QdH, g, m_iPtsDEvsQ, xH,
                                   bCenterAtQd, iMesh);

    for(int i=0; i<(NQd-2); i++)
    {
        Q = aMinQ[i+2] = (Q1 + Q2) / 2.0;
        DE = aMin[i+2] = PlotSingleDEvsQ(3+i, szPrefix, Q, g, m_iPtsDEvsQ, xH,
                                          bCenterAtQd, iMesh);

        if(DE < 0.0)
        {
            Q1 = Q; DE1 = DE;
        }
        else
        {
            Q2 = Q; DE2 = DE;
        }
    }

    printf("\r\nxmins%s={", szPrefix);
    for(int i=0; i<NQd; i++)
    {
```

```
        if(i!=0) printf(",");
        printf("{%.15f,%.15f}", aMinQ[i], aMin[i]);
    }
    printf("}\r\n\r\n\r\n");

    delete [] aMinQ;
    delete [] aMin;
}

double CTriLayerPhasePoint::PlotSingleDEvsQ(int iIndex, const char* szPrefix,
           double Qd, double g, int iPts, double Xmax,
           bool bCenterAtQd, int iMeshPts, bool bMeasureAccuracy)
{
    if(m_DiagnMode == modeNumericalDiag)
    {
        CExchEnergyND   ExchE(iMeshPts);
        CKinEnergyND    KinE;

        return PlotSingleDEvsQ(KinE, ExchE, iIndex, szPrefix, Qd, g, iPts,
                               Xmax, bCenterAtQd, iMeshPts, bMeasureAccuracy);
    }

    return 0.0;
}

double CTriLayerPhasePoint::PlotSingleDEvsQ(CEnergy& KinE, CEnergy& ExchE,
           int iIndex, const char* szPrefix, double Qd, double g,
           int iPts, double Xmax, bool bCenterAtQd, int iMeshPts,
           bool bMeasureAccuracy)
{
    double*         pAccuracy = NULL;
    double          Qsu, Qsd, Qsu2;
    double          dRes, dx, x, xp;
    double          aE[2*iPts + 1];
    double          dMinE;
    int             c = 0;
    int             iPtsE = iPts;


    if(bMeasureAccuracy) pAccuracy = new double;
    if(bCenterAtQd) iPtsE = 0;

    printf("data%s%i = {", szPrefix, iIndex);

    dx = fabs(Xmax) / double(iPts);
    for(int j=-iPts; j<=iPtsE; j++)
    {
        xp = x = double(j) * dx;
        if(bCenterAtQd) x += Qd*Qd;

        if(x <= 0.0)
        {
            if(Qd > M_SQRT1_2) continue;
```

```
            Qsu = sqrt(2.0*Qd*Qd + fabs(x));
            Qsd = sqrt(fabs(x));

            dRes = ExchE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierTwoTypes,
                                pAccuracy);
            dRes+= KinE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierTwoTypes);
        }
        else
        {
            Qsu2 = 2.0*Qd*Qd - x;
            if(Qsu2 > 0.0)
            {
                Qsu = sqrt(Qsu2);
                Qsd = sqrt(x);

                dRes = ExchE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierOneType,
                                    pAccuracy);
                dRes+= KinE.Calculate(Qd, Qsu, Qsd, g, CEnergy::carrierOneType);

            } else continue;
        }

        if(j != -iPts) printf(", ");
        if(pAccuracy) printf("{%.15f, %.15f, %.15f}", xp, dRes, *pAccuracy);
        else          printf("{%.15f, %.15f}", xp, dRes);
        aE[c++] = dRes;
    }

    printf("};\r\n");
    printf("ListLinePlot[data%s%i]\r\n", szPrefix, iIndex);

    dMinE = FindMinimum(aE, dx, c);
    printf("MinE%s%i:=%.15f", szPrefix, iIndex, dMinE);
    printf("\r\n\r\n");

    if(pAccuracy) delete pAccuracy;

    return dMinE;
}

double CTriLayerPhasePoint::FindMinimum(double* aE, double& dx, int iSize)
{
    double  dMinE = DBL_MAX;
    double  a, b, c, Xm;
    int     iMin = 0;


    for(int i=0; i<iSize; i++)
    {
        if(aE[i] < dMinE)
        {
            dMinE = aE[i];
            iMin = i;
        }
```

```
    }

    if((iMin > 0) && (iMin < (iSize-1)))
    {
        a = 1.0 / (2.0*dx*dx) * (aE[iMin-1] - 2.0*aE[iMin] + aE[iMin+1]);
        b = 1.0 / (2.0*dx) * (aE[iMin+1] - aE[iMin-1]);
        c = aE[iMin];

        if(a == 0.0) dMinE = aE[iMin];
        else
        {
            Xm = -b / (2.0*a);
            dMinE = a*Xm*Xm + b*Xm + c;
        }
    }
    else
    {
        dMinE = aE[iMin];
    }

    return dMinE;
}
```

G.7 The `main()` entry point

Function body:

```
#include "CmdLineParser.h"
#include "TriLayerPhasePoint.h"
#include "Timer.h"
#include "Dispersion.h"
#include <iostream>

int main(int argc, const char * argv[])
{
    CTimer               Timer;
    CCmdLineParser       CmdLine;
    CTriLayerPhasePoint  TriLayerPhasePt;
    CTriLayerDispersion  Dispersion;
    double               QdL = 0.0, QdH = 0.1;
    double               g = 6.0;
    double               xH = 0.1*0.1;
    bool                 bCenterAtQd = 0;
    bool                 bBinSearch = 0;
    int                  NQd = 5, iMesh = 16;
    int                  iAll = -1;
    int                  iSingle = -1;
    int                  iMode = 0;
    int                  iDMode = 1;
    int                  iPtsDEvsQ = 25;


    if(!CmdLine.Parse(argc, argv)) return 0;
    for(int i=0; i<CmdLine.GetNumArgs(); i++)
```

```
{
    // Points per curve for one DE versus Q plot
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Pts")==0)
    {
        iPtsDEvsQ = (int)CmdLine.GetArg(i).m_dVal;
    }

    // Mode
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Mode")==0)
    {
        iMode = (int)CmdLine.GetArg(i).m_dVal;
    }

    // Diagonalization mode
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "DMode")==0)
    {
        iDMode = (int)CmdLine.GetArg(i).m_dVal;
    }

    // Low Qd
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "QdL")==0)
    {
        QdL = CmdLine.GetArg(i).m_dVal;
    }

    // High Qd
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "QdH")==0)
    {
        QdH = CmdLine.GetArg(i).m_dVal;
    }

    // Step Qd
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "NQd")==0)
    {
        NQd = (int)CmdLine.GetArg(i).m_dVal;
    }

    // Coupling g
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "g")==0)
    {
        g = CmdLine.GetArg(i).m_dVal;
    }

    // High x
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "xH")==0)
    {
        xH = CmdLine.GetArg(i).m_dVal;
    }

    // Center at Qd^2
    if(strcmp(CmdLine.GetArg(i).m_cSwitch, "CAQd")==0)
    {
        bCenterAtQd = CmdLine.GetArg(i).m_dVal;
    }
```

```
        // Use binary search algorithm for a multiple DE vs Q plots
        if(strcmp(CmdLine.GetArg(i).m_cSwitch, "BinS")==0)
        {
            bBinSearch = CmdLine.GetArg(i).m_dVal;
        }

        // Number of mesh points used in all directions
        if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Mesh")==0)
        {
            iMesh = CmdLine.GetArg(i).m_dVal;
        }

        // Sequence through all points in phase diagram
        if(strcmp(CmdLine.GetArg(i).m_cSwitch, "All")==0)
        {
            iAll = CmdLine.GetArg(i).m_dVal;
        }

        // Plot single delta E versus x
        if(strcmp(CmdLine.GetArg(i).m_cSwitch, "Single")==0)
        {
            iSingle = CmdLine.GetArg(i).m_dVal;
        }
    }

    TriLayerPhasePt.m_iPtsDEvsQ = iPtsDEvsQ;
    TriLayerPhasePt.m_DiagnMode = (CTriLayerPhasePoint::EMode)iDMode;

    // Mode : Calculate phase diagram
    if(iMode == 0)
    {
        Timer.Start();

        if(iAll > 0)
        {
            TriLayerPhasePt.PlotAllDEvsQ(bCenterAtQd, iMesh, iAll);
        }
        else if(iSingle == 1)
        {
            TriLayerPhasePt.PlotSingleDEvsQ(1, "S", QdL, g, iPtsDEvsQ, xH,
                                            bCenterAtQd, iMesh);
        }
        else
        {
            if(!bBinSearch)
                TriLayerPhasePt.PlotMultipleDEvsQ("A", QdL, QdH, NQd, g, xH,
                                                  bCenterAtQd, iMesh);
            else
                TriLayerPhasePt.PlotBinSearchDEvsQ("A", QdL, QdH, NQd, g, xH,
                                                   bCenterAtQd, iMesh);
        }

        Timer.Stop();
```

```
    Timer.PrintElapsed();
}

// Mode : Calculate full dispersion
if(iMode == 1)
{
    Dispersion.InitFullDispersion(-xH, xH);
    Dispersion.PrintFullDispersion();
}

// Mode : Calculate low energy 2D dispersion
if(iMode == 2)
{
    Dispersion.InitLin2DDispersion(-xH, xH);
    Dispersion.PrintLin2DDispersion();
}

// Mode : Calculate low energy 1D dispersion
if(iMode == 3)
{
    Dispersion.InitLin1DDispersion(-xH, xH);
    Dispersion.PrintLin1DDispersion();
}

// Mode : Calculate full 1D dispersion
if(iMode == 4)
{
    Dispersion.InitFull1DDispersion(-xH, xH);
    Dispersion.PrintFull1DDispersion();
}

// Mode : Calculate low energy diagonalization matrix
if(iMode == 5)
{
    Dispersion.InitLinDiagMatrix(-xH, xH);
    Dispersion.PrintLinDiagMatrix();
}

// Mode : Test accuracy of integrations
if(iMode == 6)
{
    Timer.Start();
    TriLayerPhasePt.PlotSingleDEvsQ(1, "S", QdL, g, 25, xH, iMesh, true);
    Timer.Stop();
    Timer.PrintElapsed();
}

// Mode : Calculate 1D density of states, band 1
if(iMode == 7)
{
    Dispersion.InitLin1DDensityOfStates(0.0, 1.5);
    Dispersion.Print1DDensityOfStates(2, 0.0, xH);
}
```

```
    // Mode : Calculate 1D density of states (as function of p), band 1
    if(iMode == 8)
    {
        Dispersion.InitLin1DDensityOfStates(0.0, 1.5);
        Dispersion.Print1DDensityOfStates(2, 0.0, xH, true);
    }

    return 0;
}
```

Bibliography

[1] A. H. C. Neto, F. Guinea, N. M. R. Peres, K. S. Novoselov, and A. K. Geim, Rev. Mod. Phys. **81**, 109 (2009).

[2] K. S. Novoselov et al., Science **306**, 666 (2004).

[3] D. S. L. Abergel, A. Russell, and V. I. Fal'ko, Appl. Phys. Lett. **91**, 063125 (2007).

[4] P. Blake et al., Appl. Phys. Lett. **91**, 063124 (2007).

[5] C. Casiraghi et al., Nano Lett. **7**, 2711 (2007).

[6] A. C. Ferrari et al., Phys. Rev. Lett. **97**, 187401 (2006).

[7] P. R. Wallace, Phys. Rev. **71**, 622 (1947).

[8] K. S. Kim et al., Nature **457**, 706 (2009).

[9] H. Cao et al., J. Appl. Phys. **107**, 044310 (2010).

[10] S. Bae et al., Nature Nano. Phys. **5**, 574 (2010).

[11] K. K. Gomes, W. Mar, W. Ko, F. Guinea, and H. C. Manoharan, Nature **483**, 306 (2012).

[12] N. M. R. Peres, F. Guinea, and A. H. C. Neto, Phys. Rev. B **72**, 174406 (2005).

[13] J. Nilsson, A. H. C. Neto, N. M. R. Peres, and F. Guinea, Phys. Rev. B **73**, 214418 (2006).

[14] Y.-M. Lin et al., Science **327**, 662 (2010).

[15] G. Fiori and G. Iannaccone, IEEE Electr. Dev. Lett. **30**, 261 (2009).

[16] F. Schedin et al., Nature Mat. **6**, 652 (2007).

[17] N. Mohanty et al., Nature Comm. **3**, 844 (2012).

[18] J. Cai et al., Nature **466**, 470 (2010).

[19] C. A. Leatherdale, W.-K. Woo, F. V. Mikulec, and M. G. Bawendi, J. Phys. Chem. B **106**, 7619 (2002).

[20] M. A. Walling, J. A. Novak, and J. R. E. Shepard, Int. J. Mol. Sci. **10**, 441 (2009).

[21] K. S. Novoselov et al., Nature **438**, 197 (2005).

[22] Y. Zhang, Y.-W. Tan, H. L. Stormer, and P. Kim, Nature **438**, 201 (2005).

[23] R. van Gelderen, L.-K. Lim, and C. M. Smith, Phys. Rev. B **84**, 155446 (2011).

[24] F. Zhang, B. Sahu, H. Min, and A. H. MacDonald, Phys. Rev. B **82**, 035409 (2010).

[25] N. M. R. Peres, F. Guinea, and A. H. C. Neto, Phys. Rev. B **73**, 125411 (2006).

[26] S. Yuan, R. Rolda'n, and M. I. Katsnelson, Phys. Rev. B **84**, 035439 (2011).

[27] N. D. Mermin and H. Wagner, Phys. Rev. Lett. **17**, 1133 (1966).

[28] T. Georgiou et al., Nature **99**, 093103 (2011).

[29] N. Levy et al., Nature **329**, 544 (2011).

[30] S. H. Jhang et al., Phys. Rev. B **84**, 161408 (2011).

[31] C. H. Lui, Z. Li, K. F. Mak, E. Cappelluti, and T. F. Heinz, Nature Phys. **7**, 944 (2011).

[32] A. A. Avetisyan, B. Partoens, and F. M. Peeters, Phys. Rev. B **81**, 115432 (2010).

[33] Y. M. Zuev, W. Chang, and P. Kim, Phys. Rev. Lett. **102**, 096807 (2009).

[34] C.-R. Wang, W.-S. Lu, and W.-L. Lee, Phys. Rev. B **82**, 121406 (2010).

[35] R. Ma, L. Sheng, M. Liu, and D. N. Sheng, arXiv , 1206.4387 (2012).

[36] S. B. Kumar and J. Guo, Appl. Phys. Lett. **100**, 163102 (2012).

[37] M. Koshino and E. McCann, Phys. Rev. B **80**, 165409 (2009).

[38] M. V. Berry, Proc. R. Soc. Lond. **392**, 1802 (1984).

[39] J.-A. Yan, W. Y. Ruan, and M. Y. Chou, Phys. Rev. B **77**, 125401 (2008).

[40] Z. Li et al., Phys. Rev. Lett. **108**, 156801 (2012).

[41] A. Kumar et al., Phys. Rev. Lett. **107**, 126806 (2011).

[42] F. Zhang, D. Tilahun, and A. H. MacDonald, Phys. Rev. B **85**, 165139 (2012).

[43] K. F. Mak, J. Shan, and T. F. Heinz, Phys. Rev. Lett. **104**, 176404 (2010).

[44] J.-A. Yan, W. Y. Ruan, and M. Y. Chou, Phys. Rev. B **83**, 245418 (2011).

[45] J. Jung, F. Zhang, Z. Qiao, and A. H. MacDonald, Phys. Rev. B **84**, 075418 (2011).

[46] R. Xiao et al., Phys. Rev. B **84**, 165404 (2011).

[47] M. Taut and R. Xiao, Phys. Rev. B **84**, 233404 (2011).

[48] J. Borysiuk, J. Soltys, and J. Piechota, Nature **109**, 093523 (2011).

[49] S. C. Brenner and L. Scott, *The Mathematical Theory of Finite Element Methods*, Springer, 2010.

[50] M. Mori and M. Sugihara, Journal of Computational and Applied Mathematics **127**, 287 (2001).

[51] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes*, Cambridge, 2007.

[52] M. G. Duffy, SIAM Journal on Numerical Analysis **19**, 1260 (1984).

[53] D. J. Griffiths, *Introduction to Quantum Mechanics*, Prentice Hall, 2nd edition, 2003.

[54] L. D. Landau and E. M. Lifshitz, *Quantum Mechanics*, Elsevier, 3rd edition, 1981.

[55] J. J. Sakurai, *Modern Quantum Mechanics*, Addison Wesley, 1994.

[56] H. T. C. Stoof, K. B. Gubbels, and D. B. M. Dickerscheid, *Ultracold Quantum Fields*, Springer, 2009.

[57] C. Kittel, *Introduction to Solid State Physics*, Wiley, 1986.

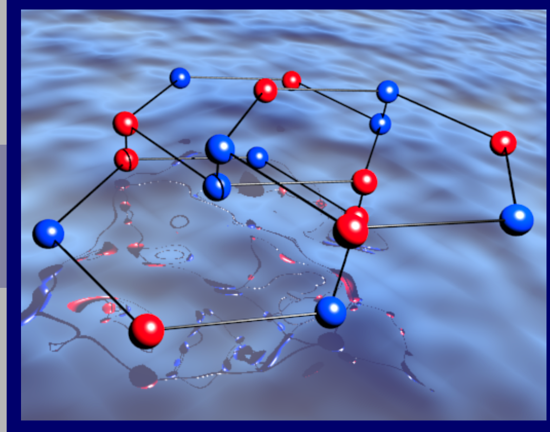
[58] H. Bruus and K. Flensberg, *Many-body quantum theory in condensed matter physics*, Oxford, 2004.

[59] J. C. Slonczewski and P. R. Weiss, Phys. Rev. **109**, 272 (1958).

[60] J. W. McClure, Phys. Rev. **108**, 612 (1957).

[61] C. Kittel, *Introduction to Solid State Physics*, Wiley, 2005.

[62] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 6th edition, 2000.



Graphene was first realized in its free form in 2004 by the physicists Andre Geim and Konstantin Novoselov. First believed to only exist as a theoretical system, this material has some very unique properties. It is a two dimensional material with an extremely high charge mobility. It turns out that stacking several layers of graphene on top of each other causes the properties of this material to change. This thesis focuses on the ferromagnetic properties of graphene. In the first few chapters, the derivations of the ferromagnetic properties of both monolayer and bilayer graphene will be treated extensively based on existing literature. The results that are presented in the literature will be reproduced by the use of numerical methods. Finally, based on this, a similar set of numerical calculations will be done for ABC-stacked trilayer graphene, which will result in a phase diagram in the electron-electron coupling versus doping plane. Finally, the effects of Coloumb screening will be investigated. Due to the "flatter" low energy bands of ABC-trilayer graphene, it will be shown that this material exhibits stronger ferromagnetism than that seen in monolayer, bilayer and ABA-trilayer graphene. This is experimentally very exciting due to the difficulty of producing graphene samples with very low doping, which makes it a big challenge to detect ferromagnetism in monolayer graphene, bilayer graphene and ABA-trilayer graphene which undergoes a phase transition to paramagnetism at relatively low levels of doping. The increased level of doping needed to make ABC-trilayer graphene paramagnetic makes its realization in the ferromagnetic phase more manageable.