C++による数値計算

志村昂輝

2026年1月22日

一級品として表れる諸々の技術や芸当は、その自由自在さで我々を驚かせますが、その背後には素人の知りえない膨大なルールと型があります。研究も例外ではなく、よりよい研究で発揮される個性はまずある一定の型やルールを身に着けた後に生まれています。研究における型を身に着ける最初の手段は、例えば学部生のうちには標準的な教科書を読んで知識を増やすことでしょうが、これは全くもって序の口にすぎず、一人前となるためには配属された研究室で師弟関係を結び、その研究室が売りとするスタイルを吸収していかなければなりません。

問題となるのが、この研究室の売りは一朝一夕に掴めるものではないということです。まず師弟関係の下で、弟子として師匠の謦咳に接しているかどうかが、素人とそうでない人とを分ける最初の分水嶺となります。将来自分の同業者になると見込んだ人間に対して、師匠はその分野の共通言語や常識を叩き込みますが、一人前とみなされるにはその量と厳しさをもってしても数年を要します。修士と博士合わせて5年かかるのは、この修行が並大抵のものではなく、素人から業界人へと決定的に変化させる重大なプロセスであることを示唆しているでしょう。

本稿は以上2つの困難を克服するために作成されています。つまり、加藤研内部で共有されている重要だがインアクセシブルな技術をまとめ、研究を始める際に典型的に出くわす困難とその対処法を示すこと、及び内容を極力端的に整理し、一学生から研究者になるための経路を効率よく整備することです。ただし、私の特殊なバックグラウンドに起因する限界について読者にお断りしなくてはなりません。まず加藤雄介研究室で扱われるテーマと技術は広く、私がここで示す内容はそのほんの一部にすぎないことです。もとより本研究室の強みは数値計算よりも微分方程式の求解や特殊関数の操作といった解析計算に存在しており、本稿の内容はむしろ傍流といえるかもしれません。主流といえるそちらの技術については、読者の皆様が教員との長い議論や共同研究の中で少しずつ身に着けたり、あるいは今後誰かが同様にドキュメント化してくれることを期待いたします。

第1章では、数値計算や解析の作業に適したプログラミング言語の紹介と簡単な文法の解説、及び環境構築の仕方について述べます。

強相関物質の研究スタイルにはかなり決まった型が存在していますが、出発点として死活的に重要なのがハミルトニアンの構築とその対角化です。第2章では、強束縛模型について軽く説明した後、ハミルトニアンを実装する方法、及びその対角化の方法について述べます。ハミルトニアンの対角化により得られた固有値がエネルギー分散ですが、実際に研究したり論文を書いたりするうえで必要なバンド図やフェルミ面の書き方についてもここで解説します。

第3章ではグリーン関数の実装方法について述べます。グリーン関数の虚部から得られる状態密度の描画方法もここで述べます。

最後に、本書はあくまでも研究室内ドキュメントであり必要に応じて参照すればよいと思いますが、内容をしっかり理解したい方は掲載されているプログラムを自分の手で実装してみるとよいと思います。最初はコードの写経でもよいですが、書き換えや順序の入れ替えで挙動がどうなるかをチェックするとより理解が深まるかと思います。

第1章　プログラミング言語の選択と環境構築

1.1　数値計算向きの言語 (C++, Fortran)

数値計算という目的に絞ってもプログラミング言語の選択肢は多岐にわたりますが、物性物理で格子模型や多体問題を扱う場合、計算量は系の大きさや自由度の増加とともに急激に増大します。したがって、実行速度やメモリ配置を意識する必要があり、C++やFortranはそれにふさわしい選択肢の一つとして挙げられます。明示的に型が指定されるために、数値誤差のふるまいを把握しやすくなるのも重要です。

1.1.1　C++について

C++はC言語が基盤となっていますが、クラスやテンプレートといった機能を導入することで大規模なプログラムの構築や保守に向いています。本稿で主に用いる言語です。コンパイラもフリーで手に入り、特にLinuxでは標準装備されています。Windowsでもコンパイラがフリーで入手できるようです。

数値計算上一番大きなメリットはなんといっても、実行速度の速さです。C++はコンパイル型言語で、ループなどが機械語に近い形で最適化されるために、行列演算や反復計算で高い性能を発揮します。また配列の確保や解放を動的に行えば巨大配列を扱うこともできるので、波数空間上の物理量の情報を格納する場合に非常に効率がよいです。数値計算のライブラリも充実しています。

デメリットとして、メモリ管理を意識した低水準な記述が前提となっているために、数値計算にバグが混じりやすくなることが挙げられます。Fortranに比べると覚える概念が多く、特にポインタは大部分の学習者が躓く場所として悪名高いもので、言語に対する正確な理解と注意深い実装が求められます。それでも私がここでC++をお勧めするのは、学習コストが高い分ほかの言語を学習する際のハードルが下がるであろうことや、Fortranに比べるとできることが多いこと、最後に身も蓋もありませんが、筆者がFortranよりも長く触れているためにドキュメントを作りやすいことがあります。例えば個人的に文字列操作はFortranよりもC++のほうが便利だと考えています。

1.1.2　Fortranについて

数値計算の言語としてはFortranも依然重要な選択肢といえます。物性理論だけでなく、科学技術計算に特化した言語として長い歴史を持つために、研究室によっては保守性

の観点から Fortran によるコーディングを強いる場合もあります。これはオリジナリティ保護の観点から理にかなっていて、研究室の強みが Fortran に支えられており、いわば研究室独自のライブラリとして使いまわすことができるのです。大規模プロジェクトならなおさら、既存の技術的な資産を一人で書き換えることも不可能でしょう。しかしながら加藤雄介研究室は数値計算の研究室ではなく、そのような蓄積はないので絶対に Fortran を使わなければいけない環境ではありません。

Fortran にはポインタなどの概念が存在せず、比較的コーディングしやすいのも利点です。数値計算以外には基本的に向いていないのがデメリットで、Fortran ができるよりも C++ができる人のほうが、今後数値計算以外のことを仕事にする場合に融通が利くのではないかと考えています。

1.1.3 Python3 について

実のところ筆者が一番長く触れている言語です。数値計算を補助する言語として Python3 も非常に有用です。C++や Fortran に比べると、Python3 はライブラリが充実していて、可視化やデータ解析を迅速に行える利点もあります。[1] 型指定もないので、簡単な計算を行いたいだけなら C++や Fortran よりも圧倒的に便利です。特に配列 (リスト) の定義が楽で、線形代数関連のライブラリが充実しており簡単に計算できるのは大きな魅力です。コードの読解も比較的やさしく、精神的な負担が少ないです。

Python3 はインタプリタ言語であり、実行速度やメモリ制御の面ではコンパイル型言語に劣るために、大規模数値計算の中核を担うのには不適切である場合もあります[2]。また型宣言がないのは実装や保守性の観点から苦しみの種となる場合もあります。しかしながら、C++や Fortran で実行した本計算の出力を Python で読み込み、プロットや解析を行うといった使い方は広く行われており、その自由度はほかの言語の追随を許しません。

1.1.4 近年注目されている言語 (Julia)

筆者は常用しておらず、軽く触れる程度しかできませんが、近年数値計算を目的としたプログラミング言語として注目されているものの一つに Julia があります。素早く実装することができ、かつ可読性が高い部分は Python に似ていますが、型安定なコードなら C や Fortran に劣らない速度で計算できるため、両者のいいとこどりをしています。現時点でのデメリットといえば、比較的新しい言語であるためにライブラリやパッケージの進化が早くバージョン管理が大変であろうことだと考えられますが、最近は研究会なども開かれており今後利用者は増えていくと見込まれます。

[1] 可視化には matplotlib, 数値計算やデータ解析には numpy や scipy といったライブラリが便利でしばしば使われます。

[2] 厳密には計算効率を上げるための工夫が様々に存在するようですが、その工夫に割くコストを考えると最初から C++や Fortran で実装した方がよいという意見が専門家の間では散見されます。

1.2 C++で使える高速数値計算ライブラリ

行列の積は添え字をループで回して和を取れば計算できますし、フーリエ変換もただの数値積分として実装することは一応できます。しかしそれらを自前で実装するのは時間がかかりますし、何より思わぬバグを招いてなかなか研究が進まないといった問題に直面しえます[2]。物性理論で頻出するこうした計算は、標準的なライブラリを呼び出して使うのが一般的です。代表的なものを以下に列挙しますが、実装例はのちの章で軽い文法とともに解説します。

1.2.1 BLAS/LAPACK(線形代数用ライブラリ)

BLAS はベクトルおよび行列に対する基本的な演算を提供する数値線形代数のライブラリで、ベクトルの内積をはじめ行列とベクトルの積、行列と行列の積も計算することが出来ます。これらの演算はそれぞれ Level 1, 2, 3 のように分類されています。LAPACK は BLAS を基礎として、行列の対角化や連立一次方程式の求解などを行うことができます。LU 分解や QR 分解などの行列の分解を行うこともできます。これらの外部ライブラリは時と試行回数の試練を潜り抜け、正確性を保証した業界のデファクトスタンダードとなっています。

1.2.2 FFTW3(フーリエ変換)

高速フーリエ変換 (FFT) は読んで字のごとく、フーリエ変換を高速に行うためのライブラリです。多次元のフーリエ変換も行うことが出来ます[1]。関数にはいろいろあり、実数関数と複素関数で使い分ける必要があります。また逆フーリエ変換とフーリエ変換の係数の違いは反映していないので、こちらで指定する必要があります。

1.2.3 OpenMP(並列計算)

ループ計算を行うとき、メモリ並列を行って複数の計算を同時に行うことで時間が短縮できることがあります。その手段として OpenMP が用いられます。並列化には MPI という手段もありますが、MPI はプロセスごとに情報の通信を行う必要があり、やや学習コストが高いものとなっています。スレッド並列なので既存のコードをほとんど変えずに並列化可能なのが利点とも言えます。

---

[2]勉強のために一から実装するのは効果的ではありますが、修士博士を途中まで経た身からすると、そうしたことにかまけている時間はそれほどなくお勧めできません

[1]昔はメッシュサイズが 2 の冪でないと計算できなかったようですが、アルゴリズムの進歩により任意のメッシュサイズで計算できるようになっています。

第2章　C++による数値計算の基礎

2.1　C++の文法速習

2.2　C++による対角化

本節では、C++からLAPACKを呼び出してエルミート行列を対角化するための手順をまとめます。複素行列を扱うか実対称行列を扱うかで用いるLAPACKのルーチンが異なる点にも注意すべきですが、呼び出し自体は数行で済みます。対角化には、エルミート行列を対角化するzheevか、dsyevを用います。LAPACKはもともとFortranで用いられていたライブラリであるため、column-majorでメモリの連続領域に格納されています。そこでC++では`std::vector<std::complex<double>>`を用いることで、要素H_{ij}に$H[i+N*j]$としてアクセスすることにします。

サンプルプログラムは**chapter2-1.cpp**です。

2.2.1　zheevの引数

zheevの引数はこちらが考えて指定するものはそれほど多くなく、慣れれば簡単に実装できるようになりますが、一応各引数の説明をします。

- `jobz`：char型。'V'か'N'の値をとる。'V'では固有値と固有ベクトルを計算する。'N'では固有値のみを計算する。
- `uplo`：char型。'U'か'L'の値を取る。エルミート行列は上三角部分と下三角部分さえ参照すれば全体の情報が分かるので、どちらの部分を参照するかを指定する。
- `n`：int型。対角化する行列のサイズを入力する。3×3行列なら3と入力する。
- `a(n*n)`：`std::vector<std::complex<double>>`型。入力として、対角化したい行列を与える。
- `lda`：int型。普通はnでよい。これは1次元配列として行列を考えたときに、次の列の情報をメモリのどこに渡すかを指定するものなので、nより大きな値を指定しても動く場合が多い。むしろnより小さな値を与えると意図しない結果をもたらすことになる。
- `w`：`std::vector<double>`型。行列の固有値を昇順に返す。エルミート行列の固有値はすべて実数であるので、double型としてよい。

- `lwork` : int 型で、最初に-1 を指定する。対角化しようとしている行列サイズで作業配列がどれくらい必要かを LAPACK に問い合わせるための引数。
- `work` : `std::complex<double>*`型。lwork で取得したサイズで work を確保する。
- `rwork` : `std::vector<double>`型。ふつうは `3*n-2` で指定する。
- `info` : int 型。対角化の一連の作業が正常終了したかどうかを表し、例外処理などに用いる。info が 0 なら正常終了しているが、それ以外の場合はどこかで失敗している。info が正の値であると、−info 番目の引数に不正があったことが示される。info が正の場合は固有値計算が収束しなかったことを示す。

zheev は入力で与えた行列を上書きするので、元のハミルトニアンを別に使いたい場合は、対角化前に行列をコピーして保持しておくべきです。

2.2.2 コンパイルおよび実行

LAPACK は単体では動かず、内部で BLAS も呼び出しています。したがって、C++で LAPACK を使う場合、リンク時に LAPACK と BLAS を同時にリンクする必要があります。コンパイルは

```
g++ chapter2-1.cpp -O2 -llapack -lblas -o main
```

のように行います。無事コンパイルが終わると実行ファイル `main` が生成されるので、

```
./main
```

で実行できます。

2.2.3 対角化の手順

chapter2-1.cpp は、3×3 のエルミート行列

$$H = \begin{pmatrix} 1.0 & 0.2 + 0.1i & 0 \\ 0.2 - 0.1i & 2.0 & 0.3 \\ 0 & 0.3 & 3.0 \end{pmatrix} \tag{2.1}$$

を対角化し、行列の固有値と固有ベクトルを出力するコードです。また対角化が出来ているかどうかの検証も行っています。

LAPACK の関数を C++から呼ぶために、`extern "C"`という宣言を行っています。もともと `dsyev` と `zheev` は Fortran で実装されている関数で、C++から呼び出すのために必要な宣言となっています。[1]

[1] C++では name mangling といい、関数の名前を定めるときに、引数の型や返却値の型など複数の意味を含めて修飾する手法があります。C++で参照する名前と Fortran で参照する名前が違う場合に、`extern "C"`とすることで C 言語と互換な名前解決を試みることが出来ます。

またここでは行列を表現する配列を実装する際に `template <class T>`を使うことで、`T` を型パラメータとして、T が double 型であっても、複素型であっても同じ行列として認識できるようにしています。

では実際にプログラムを実行してみましょう。この行列の固有値は

$$\lambda_1 \approx 0.95028847, \qquad \lambda_2 \approx 1.96487426, \qquad \lambda_3 \approx 3.08483726.$$

となっていて、解析的にはきれいに求まりません。きちんと対角化できているかをチェックするには、エルミート行列の固有値及び固有ベクトルの性質に問題がないかを確かめる必要がありますが、ここではエルミート行列の以下のような性質を用いて検証します。

- 固有値は実数である
- 固有ベクトルは正規直交系をなす

上の 2 つの性質から、固有ベクトルを並べた行列 V が $V^\dagger V = \mathcal{I}$ を満たすことがわかります。

chapter2-1.cpp を実行すると、出力結果は以下のようになります。

```
Eigenvalues (ascending):
  w[0] = 0.950288
  w[1] = 1.96487
  w[2] = 3.08484

Eigenvectors (columns):
v[0] = ( (-0.872671,-0.436335) (0.216909,0) (-0.0317472,0) )
v[1] = ( (-0.194332,-0.0971661) (-0.937531,0) (0.271715,0) )
v[2] = ( (0.0260935,0.0130468) (0.272004,0) (0.961854,0) )

Max residual norm2: 4.44957e-16
```

要素数 3 の 1 次元配列 w に固有値が昇順に格納されています。また規格化された固有ベクトルが 1 次元配列 v に格納されています。Max residual norm2 は、各固有値 λ_i と対応する固有ベクトル $\mathbf{v}_i$ について $r_i = |\hat{H}\mathbf{v} - \lambda_i \mathbf{v}_i|^2$ を計算し、その最大値として与えています。最大値が 10^{-16} 程度なので、数値誤差の程度で一致しており、対角化の結果が正しいことが保証されています。

2.3 C++による FFTW3 を用いたフーリエ変換

次に FFTW を用いたフーリエ変換の解説に移ります。FFTW は高速フーリエ変換を行うためのライブラリです。大まか流れとしては、まずフーリエ変換したい配列を用意した後、`fftw_plan_*`のように用意されている関数で plan を作成し、`fftw_execute(plan)` で plan(フーリエ変換) を実行、その後 plan を破壊してメモリを開放する、という流れになります。フーリエ変換後に plan とメモリを開放することを忘れないようにしてください。

2.3.1 FFTW における型

FFTW を用いる際は、通常では見慣れない型がいくつか登場します。まず FFTW における複素数型は

```
typedef double fftw_complex[2];
```

のように定められています。[0] には実部が、[1] には虚部が対応しています。FFTW で扱う引数は、std::complex<double>を受け付けないので、FFT の際は複素数型に必ず fftw_complex を指定しましょう。

次に、FFTW で要となる fftw_plan 型についてです。この型の中身はユーザーからは見えないのですが、入力値や次元数、最適化の仕方やアルゴリズムなどが内部に保持されています。我々の行うことは、フーリエ変換のための plan を作成し、それを fftw_plan で実行することです。

2.3.2 plan とは何か

FFTW で FFT を実行する場合は、fftw_plan 型を持つ変数 plan を用意します。例えば 2 次元 DFT を複素関数から複素関数に対して与える場合は

```
fftw_plan plan = fftw_plan_dft_2d(...);
fftw_execute(plan);
```

のように実行します。ここで plan は次の情報を内部で保持しています。

- 変換の次元：1 次元、2 次元、3 次元、あるいは 4 次元以上
- 各次元のメッシュサイズ：離散的な値を教える必要があるので、関数の刻み方をこちらから指定する
- 実数か複素数か
- 入力・出力の配列
- 使用する FFT アルゴリズム：順変換か逆変換か？

なお plan を作成し終わった後にユーザーが中身を参照することはできません。

2.3.3 plan 実行のための関数

FFTW で plan を作成する関数は何次元で、どういった変換を行うかが分かるようになっています。

FFTW は扱うデータ型に応じて関数名が分かれており、以下のように分類されます。

- fftw_complex 型から fftw_complex 型への変換: 複素関数から複素関数の変換に対応し、fftw_plan_dft_*と名前が付いた関数を使う。[1]

[1] 筆者はこれしか使ったことがありません。

- double 型から double 型 への変換： 実関数から実関数への変換に対応し、fftw_plan_dft_r2c_*と名前がついた関数を用いる。
- double 型から fftw_complex 型への変換： 実関数から複素関数への変換に対応し、fftw_plan_dft_c2r_*と名前の付いた関数を用いる。
- fftw_complex 型から double 型への変換： 複素関数から実関数への変換に対応し、fftw_plan_dft_r2r_*と名前の付いた関数を用いる。

他にも、関数名の先頭を fftwf_として float 型を扱う関数や、関数名の先頭を fftwl_として long double 型を扱う関数が存在します。

また、FFTW で用意されている関数は何次元の変換かを指定することが出来ます。具体的には、1 次元、2 次元、3 次元、そして任意次元の関数です。

- fftw_plan_dft_1d：1 次元フーリエ変換。
- fftw_plan_dft_2d：2 次元フーリエ変換。
- fftw_plan_dft_3d：3 次元フーリエ変換。
- fftw_plan_dft：任意次元のフーリエ変換。引数の中で次元を指定する必要がある。

FFTW の関数名は、どの型からどの型への変換か、また何次元の変換かを組み合わせて指定されます。例えば、実数から実数の変換で 3 次元フーリエ変換を行いたい場合は fftw_plan_r2r_3d と書きます。

2.3.4 fftw_plan fftw_plan_dft_2d の引数

試みに 2 次元フーリエ変換を行う関数がどのような構成になっているか見てみましょう。以下は fftw_plan plan = fftw_plan_dft_2d(Nx, Ny, in, out, FFTW_FORWARD, FFTW_MEASURE); の引数の解説です。

- n0：int 型。第 1 次元の格子点の数。
- n1：int 型。第 2 次元の格子点の数。
- in： fftw_complex*型。入力となる複素数配列を示す。実部と虚部を切り分ける必要があり、ある数 a に対して in[ix * n1 + iy][0] = a.real(); in[ix * n1 + iy][1] = a.imag(); を指定しなければならない。
- out：fftw_complex*型。出力用の複素数配列。入力の配列が破壊されてしまうことを許容すれば in == out としてもよい。
- sign：int 型。指定可能な値は FFTW_FORWARD か FFTW_BACKWARD の 2 つである。それぞれ順変換と逆変換に対応する。逆変換後には正規化を行うべし。
- flags：unsigned 型。FFTW_ESTIMATE と FFTW_MEASURE の 2 つがある。

2.3.5　フーリエ変換の手順

chapter2-2.cpp に FFTW を用いて実空間表示の関数を波数空間の関数にフーリエ変換するプログラムを示します。実空間表示の関数として、ここでは 2 次元格子上ローレンチアン

$$f(x,y) = \frac{\delta^2}{x^2+y^2+\delta^2} \tag{2.2}$$

を採用します。ここで $\gamma > 0$ はローレンチアンの幅に対応するパラメータで、値を大きくすると実空間で緩やかに減衰します。また、数値計算では、無限平面 $\mathbb{R}^2$ を直接扱うことはできないため、x, y を有限サイズの正方格子

$$x = -\frac{N_x}{2}, \ldots, \frac{N_x}{2}-1, \quad y = -\frac{N_y}{2}, \ldots, \frac{N_y}{2}-1 \tag{2.3}$$

上で定義します。ここでメッシュサイズを $N_x = N_y = 128$ とします。

2 次元フーリエ変換

$$F_{m_x,m_y} = \sum_{x=0}^{N_x-1} \sum_{y=0}^{N_y-1} f_{x,y}\, e^{-2\pi i\left(\frac{m_x x}{N_x}+\frac{m_y y}{N_y}\right)} \tag{2.4}$$

を数値計算して波数空間上にプロットします。波数との対応は

$$k_x = \frac{2\pi}{N_x}\, m_x, \quad k_y = \frac{2\pi}{N_y}\, m_y \tag{2.5}$$

で与えられます。

ローレンチアンのフーリエ変換は厳密解が知られていて

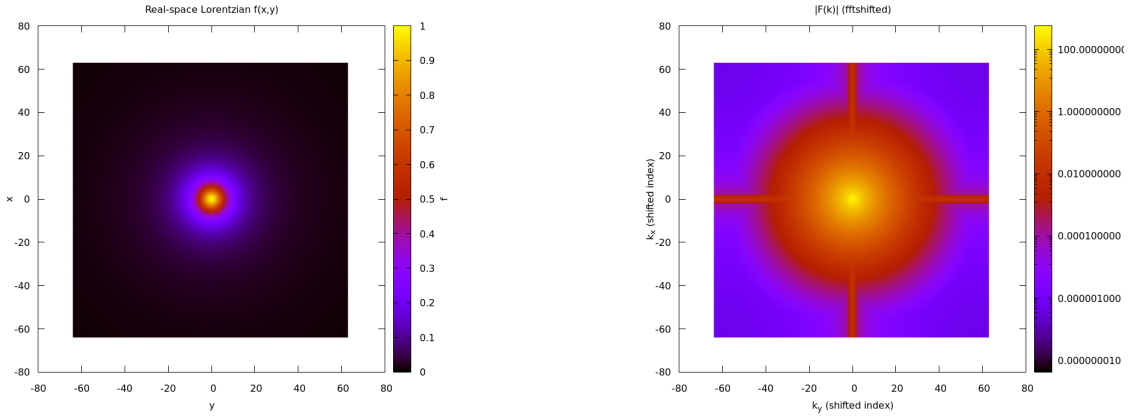
$$F(k_x,k_y) = 2\pi\delta^2 K_0(\delta|k|) \tag{2.6}$$

となります。ここで $K_0(x)$ は第 2 種変形ベッセル関数です。ところで $|\mathbf{k}| \to 0$ の極限では $K_0(z) \sim -\ln z$ のように対数発散するため、$k = 0$ 成分は数値計算で直接比較できません。そこで本研究では波数空間上の $k = 0$ となる点を除外し、$|\mathbf{k}| > 0$ の領域において数値結果との比較を行います。

図 2.2(a) にはフーリエ変換前の実空間のローレンチアンのカラーマップ、(b) にはフーリエ変換後のローレンチアンのカラーマップを示しています。また図 (c) では、$k_y = 0$ でスライスしたフーリエ変換の実部と厳密解とを比較しています。数値計算結果と厳密解がいい結果を示していることが分かります。

2.4　C++による OpenMP を用いた並列計算

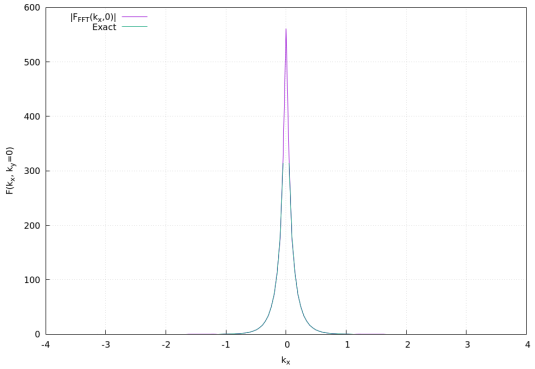
OpenMP はメモリ共有によって並列計算を行うための API で、C++や Fortran のコードに対して並列化を行うことができます。1 つのプロセスの内部で複数のスレッドを生成し、それらが同一のメモリを共有しながら計算を行います。したがって、配列やグローバル変数をそのまま共有して用いることが出来ます。まず parallel region を作り、その中でタスクをスレッドに分配し、各スレッドが同時に計算を実行する、というのが OpenMP による並列化のおおまかな流れです。



(a) (a) Real space

(b) (b) k space

図 2.1: Real-space Lorentzian and its Fourier transform.



(a) (c) compare exact

図 2.2: Real-space Lorentzian and its Fourier transform.

2.4.1 いかなるときに並列化が有効か

並列化は、ループの反復が互いに独立であるときに適しています。最も頻繁に使われるのは多重積分でしょう。また、パラメータの掃引においても並列化は威力を発揮します。独立であるかを判断する目安として、並列化させる計算の順序を入れ替えても良いかどうかを検討するとよいです。

もちろんすべてのプログラムが並列化可能なわけではありません。例えば、各 for 文の計算が非常に軽い場合はむしろ並列化のためのオーバーヘッドのほうが負荷になります。また各ループの計算が独立でない場合は意図しない結果をもたらしたり、そもそも並列化不可能であったりします。例えば漸化式のように、i 番目の計算を出力するのに i-1 番目の結果が必要になる場合は並列化を適用できません。

2.4.2 基本的な使い方

OpenMP はコンパイラが対応している必要がありますが、本稿で主に用いる g++ではデフォルトで使うことが出来ます。使用時はコンパイルオプションに-fopenmp を付ければよく、例えばコンパイル時に

```
g++ -O2 -fopenmp main.cpp -o main
```

とします。

対角化やフーリエ変換と比べると、メモリや操作順序を意識しなければならないという点で並列化には別の困難がありますが、実装自体は大変簡単で、以下の 3 つの指示を覚えれば基本的には事足ります。

- `parallel / parallel for` : parallel は並列領域を作成する指示。ほとんど for 文と組み合わせて使うので parallel for と書くことが多い。
- `shared / private` : 変数を各スレッドごとにコピーするか、全てのスレッドで共有されるものかを区別するのに使う指示。
- `reduction` : 複数のスレッドが計算した結果を 1 つにまとめるために使う。例えば、総和や内積など。

スレッド数は実行時に環境変数を指定すれば変更することが出来ます。例えば実行前に

```
export OMP_NUM_THREADS=4
```

とすれば、OpenMP は 4 スレッドで実行されます。

2.4.3 `parallel / parallel for`

chapter2-3-1.cpp をコンパイルし実行してみましょう。正しい挙動を示せば出力は以下のようになるはずです。

```
thread 9 / 16
thread 10 / 16
thread 2 / 16
...
```

まず`#pragma omp parallel` で parallel region を作成すると、OpenMP がスレッドを複数作成します。この場合は 16 本作成されています。`omp_get_thread_num()` がスレッド番号を、`omp_get_num_threads()` が parallel region の中のスレッド数を指しています。つまり `thread i / 16` は 16 本あるスレッドのうち i 番目のスレッドが標準出力を実行したことになります。

chapter2-3-2.cpp をコンパイルし実行すると、出力は以下のようになります。

```
i = 6, thread = 6
i = 0, thread = 0
i = 7, thread = 7
...
```

このように各スレッドでバラバラに計算が行われます。

2.4.4 OpenMP における変数の扱い (shared/private)

OpenMP では、parallel region に入ったときに各変数が全てのスレッドで共有されているのか、あるいはスレッドごとに独立に定められるのかを意識する必要があります。例えばすべてのスレッドで統一したかった変数が各スレッドで書き換えられてしまい、実行ごとに値が変わってしまうトラブルが起こり得ますが、こうした問題を防ぐために変数が shared か private かを定めておく必要があります。shared 変数は全スレッドが同じメモリ領域を参照するので、1 つの変数を全員で共有していると考えることが出来ます。一方で private な変数は、各スレッドが専用にコピーを保持し、他のスレッドが参照するのを防ぎます。

簡単な例でその違いを実感してみましょう。chapter2-3-3.cpp は、x に 1 を加算していく操作を並列して行うもので、直観的にはスレッド数が出力されるはずです。しかし結果は実行ごとに異なる値を吐き出します。並列計算では変数をデフォルトで shared としており、計算の競合が発生するためです。[1]

chapter2-3-4.cpp では x を private 変数として保持しています。各スレッドで `x += 1` し、全てのスレッドで $x = 1$ となっていることがお分かりになると思います。

2.4.5 reduction

OpenMP における shared 変数の扱いを見ると、並列計算には慎重な取り扱いが必要であることが分かります。例えば for 文を使って配列 a と配列 b の内積をとる操作においても、脚注で示したような競合が発生し、正しく総和が取られない可能性があります。そこで導入されるのが `reduction` です。

reduction は各スレッドに private な変数を用意して、その変数を更新します。並列領域がクローズした後に、各スレッドで更新済みの変数を合計します。これにより競合をさけ、予期しない挙動を防ぎます。

---

[1] 実行結果が毎回異なるのは少々発展的な話題であるので脚注にまとめます (しかし大事です！)。まず x に 1 を足すという操作 (`x += 1`) は、CPU の中で複数の手順を通じて行われます。「メモリから x を読み込み、レジスタ上でインクリメントし、結果をメモリに書き戻す」というのが `x += 1` で行われていることです。今、スレッド 1 とスレッド 2 が同時に `x += 1` を実行すると、先にスレッド 1 がメモリから $x = 0$ を読み込んだとしても、インクリメントする前にスレッド 2 がメモリから $x = 0$ を読み込んでしまいます (本文で「競合」と呼んだ内容)。すると、スレッド 1 が $x = 1$ をメモリに書き戻した後に再びスレッド 2 が $x = 1$ をメモリに書き込んでしまいます。これにより、スレッド 1 つ分の操作が消えてしまうことになります。つまり運よく x がスレッド数と等しくなることはあるでしょうが、実行のタイムスケジュールは毎回異なるので、多くの場合は x は実際のスレッド数より少ない値を吐き出すことになるのです。

reductionの基本的な構文は`#pragma omp parallel for reduction(operator : variable)`で与えられます。例えば、変数Xに何かの総和を記録するときは、`#pragma omp parallel for reduction(+ : X)`とします。

第3章　強束縛模型の解析

3.1　次近接ホッピングを導入した2次元正方格子の強束縛模型の構築

まず2次元正方格子上のシングルバンドの強束縛模型を実空間で構成し、その後フーリエ変換でによって波数空間表示を導きます。格子定数は1とします。格子点は $\boldsymbol{R}_i = (x_i, y_i)$ で表され、 i 番目の格子に対しフェルミオンの生成・消滅演算子 $c_i^\dagger, c_i$ を導入します。スピン添え字は省略します。

正方格子における最近接ベクトルは

$$\boldsymbol{\delta} \in \{(\pm 1, 0), (0, \pm 1)\} \tag{3.1}$$

であり、次近接ベクトルは

$$\boldsymbol{\delta}' \in \{(\pm 1, \pm 1)\} \tag{3.2}$$

となります。最近接ホッピング t と次近接ホッピング t' を含めた実空間のハミルトニアンは

$$H = -t \sum_i \sum_{\boldsymbol{\delta}} c_i^\dagger c_{i+\boldsymbol{\delta}} - t' \sum_i \sum_{\boldsymbol{\delta}'} c_i^\dagger c_{i+\boldsymbol{\delta}'} - \mu \sum_i c_i^\dagger c_i \tag{3.3}$$

と書けます。ここで実空間と波数空間の演算子の関係を

$$c_i = \frac{1}{\sqrt{N}} \sum_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}_i} c_{\boldsymbol{k}}, \tag{3.4}$$

$$c_i^\dagger = \frac{1}{\sqrt{N}} \sum_{\boldsymbol{k}} e^{-i\boldsymbol{k}\cdot\boldsymbol{R}_i} c_{\boldsymbol{k}}^\dagger \tag{3.5}$$

と定義して、式 (1) のハミルトニアンの第一項

$$-t \sum_i \sum_{\boldsymbol{\delta}} c_i^\dagger c_{i+\boldsymbol{\delta}} \tag{3.6}$$

にフーリエ変換の表式を代入すると，

$$c_i^\dagger c_{i+\boldsymbol{\delta}} = \frac{1}{N} \sum_{\boldsymbol{k},\boldsymbol{k}'} e^{-i\boldsymbol{k}\cdot\boldsymbol{R}_i} e^{i\boldsymbol{k}'\cdot(\boldsymbol{R}_i+\boldsymbol{\delta})} c_{\boldsymbol{k}}^\dagger c_{\boldsymbol{k}'} \tag{3.7}$$

となります。第2項の次近接ホッピングに関する項も同様に計算することができます。

ここで格子点 i について和を取ると、N を格子点の数として

$$\sum_i e^{i(\boldsymbol{k}'-\boldsymbol{k})\cdot\boldsymbol{R}_i} = N\delta_{\boldsymbol{k},\boldsymbol{k}'} \tag{3.8}$$

であることを用いて

$$-t\sum_{\boldsymbol{k}}\left(\sum_{\boldsymbol{\delta}} e^{i\boldsymbol{k}\cdot\boldsymbol{\delta}}\right) c^\dagger_{\boldsymbol{k}} c_{\boldsymbol{k}} - t'\sum_{\boldsymbol{k}}\left(\sum_{\boldsymbol{\delta}'} e^{i\boldsymbol{k}\cdot\boldsymbol{\delta}'}\right) c^\dagger_{\boldsymbol{k}} c_{\boldsymbol{k}} \tag{3.9}$$

を得ます。特に正方格子では

$$\sum_{\boldsymbol{\delta}} e^{i\boldsymbol{k}\cdot\boldsymbol{\delta}} = e^{ik} + e^{-ik} = 2\cos k_x + 2\cos k_y \tag{3.10}$$

となります。また次近接を考慮した項について和を取ると

$$\sum_{\boldsymbol{\delta}'} e^{i\boldsymbol{k}\cdot\boldsymbol{\delta}'} = e^{i(k_x+k_y)} + e^{i(k_x-k_y)} + e^{-i(k_x-k_y)} + e^{-i(k_x+k_y)} \tag{3.11}$$

$$= 2\cos(k_x+k_y) + 2\cos(k_x-k_y) \tag{3.12}$$

$$= 4\cos k_x \cos k_y \tag{3.13}$$

となります。

以上をまとめると、次近接まで考慮した波数表示での 2 次元正方格子ハミルトニアンは

$$H = \sum_{\boldsymbol{k}} \varepsilon(\boldsymbol{k}) c^\dagger_{\boldsymbol{k}} c_{\boldsymbol{k}} \tag{3.14}$$

であり、分散関係は

$$\varepsilon(\boldsymbol{k}) = -2t(\cos k_x + \cos k_y) - 4t'\cos k_x \cos k_y - \mu \tag{3.15}$$

で与えられます。

並進対称性を仮定した場合、波数表示で既にハミルトニアンは対角的になっているので、対角化をする必要はありません。

3.2 バンド図とフェルミ面

次に、前章で導出した 2 次元正方格子のエネルギー分散 $\epsilon(\boldsymbol{k})$ を用いて、High symmetry path に沿ったバンド図及びフェルミ面を数値計算によって実際に書いてみましょう。ここではパラメータを $t = 1.0$, $t' = -0.2$ に固定し、バンド構造を観察してみます。また化学ポテンシャル μ を変えることでフェルミ面がどのように変化するかも観察してみましょう。

3.2.1　バンド図の描画

2 次元正方格子の 1st ブリルアンゾーンにおける代表的な高対称点は $\Gamma = (0, 0)$, $X = (\pi, 0)$, $M = (0, \pi)$ です。例えば縦軸にエネルギーをとり、平面上に波数空間を取る手法が自然に考えられますが 3 次元的な分散はかけても解析が難しいことのほうが多いので、高対称点を結ぶ High symmetry path に沿ってエネルギーをプロットしてバンド図とすることが多いです。

3.2.2　フェルミ面の描画

フェルミ面は $\varepsilon(k) = 0$ を満たす波数 $\boldsymbol{k}$ の集合からなります。金属の性質はおおかたフェルミ面近傍の電子が決めており、その形状は物理量のふるまいや発現する超伝導状態などに寄与します。そのためフェルミ面は金属の顔とも呼ばれ、出発点としてまずはフェルミ面を計算することに心血が注がれます。

フェルミ面を数値的に求める手順は、大まかには次の通りです。エネルギー分散は数値計算で離散的に求められますが、各点を線形補完し、$\varepsilon(\boldsymbol{k}_i)$ と $\varepsilon(\boldsymbol{k}_{i+1})$ とで符号が反転する箇所を取り出します。このようにして得られたフェルミ面はあくまで点の集合ですが、メッシュサイズを十分細かくすることによって視覚的には滑らかなフェルミ面として扱うことができます。

関連図書

[1] Hofstadter, Douglas R., Energy levels and wave functions of Bloch electrons in rational and irrational magnetic fields, Phys. Rev. B **14**, 2239(1976).

[2] C R Dean 1, L Wang, P Maher, C Forsythe, F Ghahari, Y Gao, J Katoch, M Ishigami, P Moon, M Koshino, T Taniguchi, K Watanabe, K L Shepard, J Hone, P Kim, Hofstadter's butterfly and the fractal quantum Hall effect in moiré superlattices, Nature 497(7451), (2013)