ライブラリマニュアル

吉沢徹

2017 年 12 月 15 日

目次

第 1 章

ライブラリの構成

本ライブラリは以下の 7 つのヘッダによって構成されている.

bit.h

システムのサイズと粒子数から計算基底を構築する.

observable_base.h

bit.h で形成した計算基底の元で，物理量を行列表示するための基底クラスが記述されている.

observable.h

observable_base.h に基づいて具体的な物理量を定義するファイル.

translation.h

bit.h で形成した計算基底を並進対称操作の固有ベクトルで張られる基底に移す.

ss.h

Sakurai-Sugiura 法でエネルギーシェル内部の固有状態を計算するファイル.

free_fermion.h

フリーフェルミオンで記述される系に関する計算を行う.

log.h

log 出力に関するクラスが記述されている.

第 2 章

事前にインストールするライブラリ

このライブラリを利用するに当たって，事前にインストールする必要があるライブラリについて説明する．なお，以下の説明では，ライブラリは全て ～/cpp_library　の下に展開するもと仮定する．他のディレクトリに展開をしたい場合には，そのディレクトリに置き換えて行うこと．

このライブラリは C++ の行列演算ライブラリ Eigen を用いて計算を行っている．そのため，事前に Eigen をダウンロードして CPATH が通ったライブラリ上に展開しておく必要がある．Eigen はテンプレートライブラリなので，ビルドを行う必要はなく，公式サイト (http://eigen.tuxfamily.org/index.php?title=Main_Page) からソースコードをダウンロードし，展開するだけで利用することができる．CPATH を通すためには，.bashrc に

```
export CPATH=$CPATH:~/cpp_library
```

を追加すればよい．

また, ss.h では線形方程式を解くために, fortran のライブラリ KOmega を用いている．KOmega は Github からダウンロードすることができる (https://github.com/issp-center-dev/Komega)．ダウンロードしたファイルを展開したディレクトリに移動し，

```
mkdir ~/cpp_library/Komega
./configure --prefix=\$(~/cpp_library/Komegaへのフルパス)  --enable-threadsafe
make
make install
```

と入力すると ss.h で用いることができるようになる．本ライブラリでは，Komega を動的ライブラリとして用いているため，Komega にパスを通さなくてはならない．これは.bashrc に

```
export LIBRARY_PATH=$LIBRARY_PATH:~/cpp_library/Komega/lib
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:~/cpp_library/Komega/lib
```

を追加すればよい．(Linux の場合には DYLD_LIBRARY_PATH ではなく LD_LIBRARY_PATH を書き換える)

第 3 章

ビルド・インストール

本ライブラリは autotools によって MAKEFILE を半自動的に生成することができる．そのため，KOmega と同様に，展開したディレクトリに移動して，

```
mkdir ~/cpp_library/tylib
./configure --prefix=\$(~/cpp_library/tylib へのフルパス)
make
make install
```

と入力すればインストールすることができる．このコマンド例では，～/cpp_library/tylib に本ライブラリがインストールされる．

また，共有ライブラリとして使用するために，KOmega と同様にパスを通す必要がある．

```
export LIBRARY_PATH=$LIBRARY_PATH:~/cpp_library/tylib/lib
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:~/cpp_library/tylib/lib
```

第 4 章

ライブラリの使用

本ライブラリは，C++ プログラムのヘッダ部分に

```
#include <tylib/include/bit.h>
#include <tylib/include/observable.h>
```

などと，利用するヘッダファイルをインクルードすることで用いる．

本ライブラリでは C++11 で実装されている乱数ライブラリを用いていることから，コンパイルの際には，C++11 を用いるオプションをつけなければならない．また，ライブラリの実態とリンクをしなければならないため，-ltylib というオプションも必要である．このオプションは適切にパスが通っていないと使えないため，コンパイルの際にエラーが出る場合には，まずパスの確認をおすすめする．以上を踏まえると，コンパイルは以下の例のように行う．

```
g++ $(filename) -O2 -std=c++11 -ltylib
```

ss.h では KOmega のサブルーチンを使用しているため，ss.h を使う場合にはコンパイルオプションにさらに -lkomega -lblas を追加して

```
g++ $(filename) -O2 -std=c++11 -ltylib -lkomega -lblas
```

としなければならない．（Linux でこのライブラリを用いる場合には，ss.h の使用の有無に関わらず -lkomega -lblas のオプションが必要となるので注意．）また，コンパイラが intel コンパイラの場合には　-lblas　ではなく -mkl を用いる必要がある．

第 5 章

各ヘッダファイルの詳細

この章では，各ヘッダファイルのメンバとその使い方について説明する．

5.1 bit.h

class computational_basis

計算基底を張るクラス．以下にクラスメンバを示す．計算基底は辞書式で並べられている．

- computational_basis::computational_basis(int L, int N)
  コンストラクタ．L はサイト数，N は粒子数（スピンアップの数）を表す．N = -1 の場合には，粒子数非保存（磁化比保存）モードになり，2^L 次元の基底が張られる．
- int computational_basis::index(Eigen::VectorXi &bits)
  与えられたビット列 Eigen::VectorXi bits に対応する計算基底のインデックスを返す．
- Eigen::VectorXi computational_basis::bits(long index)
  与えられた計算基底のインデックスに対応するビット列を返す．
- int computational_basis::ref_L()
  L の値を返す．
- int computational_basis::ref_N)
  N の値を返す．
- int computational_basis::ref_d()
  計算基底の次元 d の値を返す．
- bool computational_basis::isConserved()
  粒子数保存モードのときに true，非保存モードのときには false を返す．

5.2 observable_base.h

本ライブラリでは，物理量はクラスとして定義される．observable_base.h は observable_base と observable_base_complex という物理量クラスを定義する上での基底クラスとなる 2 つのクラスと，物理量の定義を支援するための幾つかの関数からなる．

class observable_base

物理量の基底クラスとなるクラス．定義される物理量が計算基底の元で行列表示した場合に実行列になる場合にはこれを用いる．

- virtual std::vectorstd::pair¡int, double¿¿ observable_base::cal_elements(int index, computational_basis &basis)
  物理量を計算基底 basis の元で行列表示する計算規則の定義．与えられた計算基底のインデックスの行で非ゼロの成分を持つ列のインデックスとその成分で形成される std::pair¡int, double¿ を std::vector に格納して返す．observable_base では，この関数は仮想的に実装されているだけで，空の std::vector¡std::pair¡int, double¿ ¿を返す．実際に物理量を定義する際には，この関数を定義したい物理量に即したものに書き換えることで，物理量の計算基底の元での行列表示が得られる．
- void observable_base::cal_matrix(Eigen::SparseMatrix¡double¿ &H, computational_basis &basis)
  cal_elements で定義された行列成分の計算規則を用いて，物理量の計算基底の元での行列表示を計算して疎行列 H に格納する．この関数は，物理量によらず共通の処理を行うため，継承しても書き換える必要はない．

class observable_base_complex

物理量の基底クラスとなるクラス．定義される物理量が計算基底の元で行列表示した場合に複素行列になる場合にはこれを用いる．

- virtual std::vectorstd::pair¡int, std::complex¡double¿ ¿¿ observable_base_complex::cal_elements(int index, computational_basis &basis)
  物理量を計算基底 basis の元で行列表示する計算規則の定義．
- void observable_base_complex::cal_matrix(Eigen::SparseMatrix¡std::complex¡double¿ ¿ &H, computational_basis &basis)
  cal_elements で定義された行列成分の計算規則を用いて，物理量の計算基底の元での行列表示を計算して疎行列 H に格納する．この関数は，物理量によらず共通の処理を行うため，継承しても書き換える必要はない．

std::vector¡std::pair¡int, int¿ ¿ make_bond(int L, int neighbor)

1 次元格子において，neighbor 次近接するサイト同士の pair を作成する．neighbor = 1 で最近接，neighbor = 2 で次近接の bond を作成することができる．この関数で作成される bond は以下の関数の引数として用いることができる．

std::vector¡std::pair¡int, double¿ ¿ cal_hopping(int l, computational_basis &basis, std::vector¡std::pair¡int, int¿ ¿ &bonds, double J)

バードコアボソンにおけるホッピング $J\sum_{<i,j>}\left[\hat{b}_i^\dagger\hat{b}_j + h.c.\right]$ の行列成分を計算する．この関数は非対角的であるので，l 行に非ゼロ成分を持つ列とその成分の pair を返す．どのサイト間でホッピングがあるかは std::vector¡std::pair¡int, int¿ ¿ bonds で定義される．

std::vector¡std::pair¡int, double¿ ¿ cal_hopping(int l, computational_basis &basis, std::vector¡std::pair¡int, int¿ ¿ &bonds, std::vector¡double¿ &J)

バードコアボソンにおけるホッピング $\sum_{<i,j>}J_{ij}\left[\hat{b}_i^\dagger\hat{b}_j + h.c.\right]$ の行列成分を計算する．ボンドごとにホッピングの強さが異なる場合にはこれを用いる．

std::vector¡std::pair¡int, double¿ ¿ cal_Sx(int l, computational_basis &basis, double h)

一様な横磁場 $h\sum_i \hat{S}_i^x$ の行列成分を計算する．

std::vector¡std::pair¡int, double¿ ¿ cal_Sx_interaction(int l, computational_basis &basis, std::vector¡std::pair¡int, int¿ ¿ &bonds, double U)

double cal_Sz(int l, computational_basis &basis, double h);

一様な磁場 $h\sum_i \hat{S}_i^z$ の行列成分を計算する．この物理量は計算基底の元では対角的なので，l 行 l 列の行列成分を返す．

double cal_Sz_disorder(int l, computational_basis &basis, std::vector¡double¿ &h)

一様でない磁場 $\sum_i h_i\hat{S}_i^z$ の行列成分を計算する．

double cal_Sz_interaction(int l, computational_basis &basis, std::vector¡std::pair¡int, int¿ ¿ &bonds, double U)

スピンの z 方向のカップリング $U\sum_{<i,j>}\hat{S}_i^z\hat{S}_j^z$ を計算する．

double cal_interaction(int l, computational_basis &basis, std::vector¡std::pair¡int, int¿ ¿ &bonds, double U)

ハードコアボソンの近接相互作用 $U\sum_{<i,j>}\hat{b}_i^\dagger\hat{b}_i\hat{b}_j^\dagger\hat{b}_j$ の行列成分を計算する．

double cal_interaction_disordered(int l, computational_basis &basis, std::vector¡std::pair¡int, int¿ ¿ &bonds, std::vector¡double¿ &U)

ハードコアボソンの近接相互作用 $\sum_{<i,j>} U_{ij}\hat{b}_i^\dagger \hat{b}_i \hat{b}_j^\dagger \hat{b}_j$ の行列成分を計算する．相互作用の強さがボンドごとに異なる場合に使う．

double cal_onsite(int l, computational_basis &basis, std::vector¡double¿ &potential)

ハードコアボソンのオンサイトポテンシャル $\sum_i h_i \hat{b}_i^\dagger \hat{b}_i$ の行列成分を計算する．

5.3 observable.h

observable.h では，observable_base.h で仮想的に定義した基底クラスを継承して，具体的な物理量を定義している．このヘッダファイルには基本となる幾つかの物理量を定義しているが，このファイルにないような物理量を用いたいときには，自分で独自のものを作成することができる．作成方法で述べる．

以下に示すクラスはいずれも

- cal_elements
- cal_matrix

の 2 つのメンバ関数を持つため，それ以外のメンバを持つときのみ，その関数を示す．また，いずれの物理量でも境界条件は周期的境界条件を用いている．また，サイト数を L とし，和 $\sum_i$ は 1 から L まで取るものとする．

class XXZ_with_NNN

次近接相互作用を含む XXZ 模型

$$\hat{O} = \frac{1}{1+\lambda}\left[\hat{\mathcal{H}}_{XXZ} + \lambda \hat{W}\right]$$
$$\hat{\mathcal{H}}_{XXZ} = -J\sum_{i=1}^{L}\left[\hat{b}_i^\dagger \hat{b}_{i+1} + h.c.\right] + U\sum_{i=1}^{L}\hat{n}_i\hat{n}_{i+1},$$
$$\hat{W} = -J\sum_{i=1}^{L}\left[\hat{b}_i^\dagger \hat{b}_{i+2} + h.c.\right] + U\sum_{i=1}^{L}\hat{n}_i\hat{n}_{i+2},$$

を定義する．

- XXZ_with_NNN::XXZ_with_NNN(double J, double U, double lambda)
  コンストラクタ．J, U, λ の値を調整することで様々な模型を作ることができる．例えば，$J = 1$, $U = 0$, $\lambda = 0$ で XX 模型，$J = 1$, $U = 1$, $\lambda = 0$ でハイゼンベルク模型が計算で

きる．

class MBL

ハイゼンベルク模型に $[-W, W]$ の一様ランダム磁場 h_i を加えた模型

$$\hat{O} = -J \sum_i \hat{\mathbf{S}}_i \cdot \hat{\mathbf{S}}_{i+1} + \sum_i h_i \hat{S}_i^z$$

を定義する．この模型ではランダム磁場が大きい場合には many-body localization (MBL) が起きる．

- MBL::MBL(double W, double J)
  コンストラクタ．
- MBL::MBL(double W, double J, int seed)
  コンストラクタ．乱数を生成する際の seed を固定する．
- void MBL::reset_disorder()
  ランダム磁場をリセットする．

class transverse_field_ising

横磁場イジング模型

$$\hat{O} = J \sum_i \hat{S}_i^z \hat{S}_{i+1}^z + h \sum_i \hat{S}_i^x$$

を定義する．

- transverse_field_ising::transverse_field_ising(double J, double h)
  コンストラクタ．

class x_coupling

スピン x 方向のカップリング

$$\hat{O} = \frac{J}{L} \sum_i \hat{S}_i^x \hat{S}_{i+1}^x$$

を定義する．

- x_coupling::x_coupling(double J)
  コンストラクタ．

class z_coupling

スピン z 方向のカップリング

$$\hat{O} = \frac{J}{L} \sum_i \hat{S}_i^z \hat{S}_{i+1}^z$$

を定義する.

- z_coupling::z_coupling(double J)
  コンストラクタ.

sum_Sx

x 方向の磁化の和

$$\hat{O} = \frac{h}{L} \sum_i \hat{S}_i^x$$

を定義する.

- sum_Sx::sum_Sx(double h)
  コンストラクタ.

sum_Sz

z 方向の磁化の和

$$\hat{O} = \frac{h}{L} \sum_i \hat{S}_i^z$$

を定義する.

- sum_Sz::sum_Sz(double h)
  コンストラクタ.

hopping

ハードコアボソンのホッピング

$$\hat{O} = \frac{1}{L} \sum_i \left[\hat{b}_i^\dagger \hat{b}_{i+n} + h.c. \right] \tag{5.1}$$

を定義する.

- hopping::hopping(int distance)
  コンストラクタ.

momentum

ハードコアボソンの運動量分布

$$\hat{O} = \frac{1}{L} \sum_{i,j} e^{-i\frac{2\pi}{L}k(i-j)} \hat{b}_i^\dagger \hat{b}_j \tag{5.2}$$

を定義する．

- momentum::momentum(int k)
  コンストラクタ．

momentum_0

波数 $k = 0$ のハードコアボソンの運動量分布

$$\hat{O} = \frac{1}{L} \sum_{i,j} \hat{b}_i^\dagger \hat{b}_j \tag{5.3}$$

を定義する．

5.4 translation.h

class translational_basis

並進操作の固有ベクトルで張られる基底（以下では並進基底と呼ぶ）を扱うクラス．並進基底を利用すると，並進対称な物理量をブロック対角化することができる．並進操作の固有値は $e^{2\pi i \frac{r}{L}}$ と表されるため，以下では r を固有値のラベルとして用いる．

- translational_basis::translational_basis(computational_basis &c_basis)
  コンストラクタ．入力された計算基底 c_basis を基に並進基底を形成する．
- translational_basis::ref_d()
  元となる計算基底の次元を返す．
- translational_basis::ref_D(int r)
  固有値 r の基底の次元を返す．
- void translational_basis::cal_observable(Eigen::SparseMatrix¡std::complex¡double¿ ¿ &obs, observable_base &operation, int r, bool isTranslational = true)
  物理量 operation を固有値 r の並進基底の元で行列表示して行列 obs に格納する．isTranslational は物理量が並進対称であるかを示しており，true の場合には並進対称性を利用したアルゴリズムが用いられる．この引数はデフォルトでは true になっているため，並進対称でない物理量を計算する場合には第 4 引数に false を入れ忘れないように注意．
- void translational_basis::cal_observable(Eigen::SparseMatrix¡std::complex¡double¿ ¿ &obs, observable_base_complex &operation, int r, bool isTranslational = true)

物理量が複素数の場合にはこちらが呼ばれる．

- Eigen::VectorXcd translational_basis::eigenvector(int r, int index)
  並進操作に関する，固有値 r の $index$ 番目の固有ベクトルを計算基底の表示で返す．
- Eigen::VectorXcd translational_basis::c_to_t(Eigen::VectorXd &vc, int r)
  計算基底で表示されたベクトル vc の固有値 r の並進基底での表現を返す．
- Eigen::VectorXcd translational_basis::c_to_t(Eigen::VectorXcd &vc, int r)
  計算基底で表示されたベクトル vc の固有値 r の並進基底での表現を返す．
- Eigen::VectorXcd translational_basis::t_to_c(Eigen::VectorXd &vt, int r)
  固有値 r の並進基底で表示されたベクトル vt の計算基底での表現を返す．
- std::pair¡int, std::complex¡double¿ ¿ c_basis_to_t_basis(int c_index, int r)
  c_index 番目の計算基底は固有値 r の並進基底では何番目にどれだけの成分を持つかを返す．

5.5 ss.h

ss.h は SS 法を実装したクラス ss_method と，その実装に必要な線形方程式を解くクラス komega_cocg，komega_bicg，および，ランダムベクトルを作成するためのクラス random_vector からなる．

class ss_method

このクラスでは SS 法に基づいて，行列の固有ベクトルを計算する．このクラスは openmp によって並列化されており，関数 enable_openmp を呼び出すことで計算の一部を並列化することができる．このクラスは基本的に関数 eigenstate を呼び出すだけで十分に使うことができるが，より高度な並列化を行う場合などに備えて，SS 法のアルゴリズムの要所を支援する関数もメンバに含まれている．

- ss_method::ss_method(Eigen::SparseMatrix¡double¿ &H)
  コンストラクタ．固有ベクトルを求めたい疎行列 H を入力する．
  ss_method::ss_method(Eigen::SparseMatrix¡std::complex¡double¿ ¿ &H)
  コンストラクタ．疎行列 H が複素数の場合にはこちらが呼ばれる．
- void ss_method::eigenstate(Eigen::MatrixXcd &evec, Eigen::VectorXd &E, double gamma, double rho)
  $[\gamma-\rho, \gamma+\rho]$ の範囲に含まれる固有ベクトル・固有値を計算して evec, E に格納する．
- Eigen::MatrixXcd ss_method::projection(Eigen::VectorXcd &v, double gamma, double rho, int m)
  ベクトル v の $[\gamma-\rho, \gamma+\rho]$ の範囲の固有ベクトルに関する射影ベクトルを計算する関数．一本のベクトル v から m 本の射影ベクトルが計算される．
- int ss_method::num_states(double gamma, double rho)

の $[\gamma-\rho, \gamma+\rho]$ の範囲の固有ベクトルの数を見積もる関数．openmp によって並列化されている．

- void ss_method::projection_to_eigenstate(Eigen::MatrixXcd &evec, Eigen::VectorXd &E, Eigen::MatrixXcd &S)
  計算した射影ベクトル S　から固有ベクトルと固有値を計算して evec，E に格納する．
- void ss_method::precision_check(Eigen::MatrixXcd &evec, Eigen::VectorXd &E)
  SS 法で計算した固有ベクトルと固有値の精度を調べる試験用の関数．
- void ss_method::enable_openmp(int n_threads)
  openmp による並列化を有効にする．呼び出すと，射影ベクトルの計算が n_threads 並列で計算されるようになる．
- void ss_method::memory_setting(int memory)
  ss_method::eigenstate では，メモリを過剰に使いすぎてしまわないように，固有ベクトル数を見積もった時点で計算に必要なメモリ数を計算してそれが上限を超える場合には停止するようになっている．この上限値はデフォルトでは 32GB になっているが，この関数を呼び出すことで変更することができる．
- void ss_method::itermax_setting(int itermax)
  線形方程式を反復法で解く際の，反復回数の上限を設ける．デフォルトでは上限はない設定になっている．
- void ss_method::threshold_setting(int threshold)
  線形方程式を反復法で解く際の，解の精度を設定する．デフォルトでは 10^{-10} となっている．
- void ss_method::contour_setting(int contour, int n, double contour_para)
  SS 法における，複素積分の積分経路を設定する．このクラスでは，複素積分を n 点での台形則で評価している．contour は積分経路の形状を表しており，contour = 0 のときは楕円，contour = 1 の時は長方形を積分経路とする．また，contour_para は経路を特徴付けるパラメータであり，楕円のときは長径と短形の比，長方形のときは高さ（虚軸側）の 1/2 を表す．デフォルトでは contour = 0，contour_para = 0.1 となっている．
- void ss_method::copy_setting(int m, int step)
  一本のベクトルから何本の射影ベクトルを計算するかを設定する．一本目の射影ベクトルを $|s\rangle$ としたとき，$|s\rangle$, $\hat{\mathcal{H}}^{step}|s\rangle$, $\hat{\mathcal{H}}^{2*step}|s\rangle$, $\cdots$, $\hat{\mathcal{H}}^{(m-1)*step}|s\rangle$ の m 本のベクトルが射影ベクトルとして計算されるようになる．デフォルトでは m = 4, step = 1 となっている．
- void ss_method::kappa_setting(double kappa)
  num_states で見積もった必要な射影ベクトル数の何倍の射影ベクトルを求めるかの設定．数値計算で求めた射影ベクトルは，数値誤差により積分経路外の固有ベクトルについてもわずかに成分を持ってしまっているため，見積もり数より多くの射影ベクトルを計算しておかないと，射影演算子をうまく構築することができない．デフォルトでは kappa = 2 となっている．

- void ss_method::cutoff_setting(double cutoff)
  求めた射影ベクトルから線形独立な直交ベクトルを取り出すために特異値分解をする際，最大特異値との比が cutoff を下回っているような特異値は 0 とみなす．デフォルトでは cutoff $= 10^{-10}$ となっている．

5.6 free_fermion.h

5.7 log.h