Kristofer Björnson

Tight-Binding ToolKit

Quick start

Contents

1. Introduction

1.1 Purpose

The purpose of this document is to provide essential information needed to get started with TBTK.

1.2 Audience

The target audience are physicists interested in using TBTK to perform numerical calculations involving bilinear Hamiltonians. While TBTK is designed to be handled from the terminal, prior experience with the terminal is not assumed.

1.3 Overview

1.3.1 Working in the terminal

To enable users without experience of working in the terminal, the most important terminal commands are introduced in chapter 2. The experienced user can skip this chapter.

1.3.2 Setting up TBTK

Chapter 3 begins with a section listing prerequisites. Most supercomputers should come configured such that the required prerequisites are satisfied by default, but configuration may be necessary, in particular if TBTK is setup on a personal computer rather than a supercomputer. Users with proper system administration experience can use this section to ensure that their system is correctly configured before installing TBTK. If the user does not feel confident setting this up, or lack the adequate system administration rights, the section is instead intended to provide enough information to allow for effective communication with the system administrator.

Once the prerequisites are satisfied, information regarding how to install and update TBTK is provided, as well as information regarding initialization required each time a terminal window is opened. The chapter ends with a section that describes how to create a new project.

1.3.3 Getting started with TBTK

In this chapter two template projects are examined in detail in order to make the user able to write stand alone applications. Starting with the **EmptyProject** template, basic C++ syntax is introduced with the purpose of helping the user with little experience of C++ to get started. In the second part a complete project in which a two-dimensional tight-binding model is setup and solved is investigated. After having read this chapter, the user should be able to understand and continue learning from the template projects that can be found in the folder **TBTK/Templates/**.

1.4 Feedback

Feedback on this document is very appreciated, so do not hesitate to send an email to **kristofer.bjornson@physics.uu.se** if anything is unclear, missing, not working, etc. The code has so far only been tested on a few different types of machines, with a limited number of compiler versions etc. Information regarding the success or failure with using the code on machines with a different configuration is therefore greatly appreciated.

2. Working in the terminal

2.1 Basic terminal commands

Type	Command	Action
Symbol	**.**	Current directory
	..	Directory one level up in the directory tree
	~	Home directory
Command	**ls**	List files in current directory
	cd directory	Move into directory
	cd ..	Move up one level in the directory tree
	cp file newFile	Copy file
	cp -r folder newFolder	Copy a folder and its content
	rm file	Remove file
	mkdir newDirectory	Create directory
	rmdir directory	Remove empty directory
	rm -r directory	Remove directory and its content
	man mkdir	Manual page for a command (here mkdir)

2.2 Edit a text file

There are many different text editors, each with their own strengths and weaknesses. Each line below opens the text file in a different editor. Nano is a very simple to use editor, while vi, vim, and emacs have a lot more useful features, but can initially be more difficult to work with.

```
nano textfile
vi textfile
vim textfile
emacs textfile
```

2.3 Login to supercomputer (or other remote machine)

```
ssh username@supercomputer.location.se
password: ...
```

2.4 Copy files to and from supercomputer

Copy from personal computer to home folder on supercomputer

```
scp someFile username@supercomputer.location.se:~/some/path/
```

Copy from home folder on supercomputer to personal computer

```
scp username@supercomputer.location.se:~/some/path/ .
```

2.5 Packing and compressing files

Pack and compress **file1, file2, file3, folder1**, and **folder2** into **archiveName.tgz**

```
tar -cvzf archiveName.tgz file1 file2 folder1 file3 folder2
```

Uncompress and unpack

```
tar -xvzf archiveName.tgz
```

3. Setting up TBTK

3.1 Prerequisites

Summary of operating systems that TBTK has been verified to work with

Operating system	Distribution	version
Linux	Scientific Linux Ubuntu	? 16.04 LTS
Mac OS	?	?
Windows	?	?

Summary of libraries and softwares that are required to use TBTK, as well as optional libraries and softwares that allow for more features to be used.

Type	Name	Required	Used for
Library	**Blas**	**Yes**	Linear algebra
	Lapack	**Yes**	Linear algebra
	FFTW3	No	Fourier transform
	Arpack	No	ArnoldiSolver
	SuperLU v5.2.1	No	ArnoldiSolver
	OGRE	No	3D visulaization
	OpenCV	No	TBTKImageToModel
Software	**GCC** (min v4.9)	**Yes**	Compiling TBTK
	NVCC	No	Compiling with GPU support
	Python	No	Plotting

While **blas** and **lapack** usually are installed on supercomputers and are available by default, the other libraries are not necessarily so. Similarly, multiple versions of the **gcc** compiler can be expected to be installed on most supercomputers. However, to check which particular compiler that is used by default, type

```
gcc --version
```

If the version number is not 4.9 or higher, it is most often possible to switch the compiler by loading the appropriate **module**. Type

```
module avail
```

to see what modules are available on the system, and find the module name which indicates that it contains gcc v4.9. The exact name can differ from system to system, but assuming that the above command shows that a module with the name **gcc/4.9** is available, then load this module using

```
module load gcc/4.9
```

Verify that the correct compiler is loaded by once again typing

```
gcc --version
```

Similarly the CUDA compiler **nvcc** usually has to be loaded using a similar statement. Assume that the **module avail** command above also shows that a module named **cuda/v9.0** is available. Then type

```
module load cuda/v9.0
```

To check that the compiler is available, type

```
nvcc --version
```

3.2 Install TBTK

Download source code

```
git clone https://github.com/dafer45/TBTK
```

Enter TBTK folder

```
cd TBTK
```

Choose specific TBTK version (here v0.9). This is optional and should only be done if you know that you want to use a specific version.

```
git checkout tags/v0.9
```

Setup environment

```
source init_session.sh
```

Install TBTK. Execute the first line if the computer does not have CUDA support (GPU) in the form of a nvcc compiler. Execute the second if it has.

```
./install.sh
./install.sh -CUDA
```

Perform a basic test by compiling all the templates

```
./test.sh
```

3.3 Update TBTK

Setup environment. Only do this if this has not been done since the terminal window was opened.

```
source init_session.sh
```

Download the lates version.

```
git pull
```

Reinstall the updated library. Execute the first line if the computer does not have a CUDA support (GPU) in the form of a nvcc compiler. Execute the second if it has.

```
./update.sh
./update.sh -CUDA
```

3.4 Each time a new terminal window is opened

Whenever a new terminal window is opened. First make sure that the correct **gcc** compiler is loaded according to Section 3.1, and that if needed, so also **nvcc**. Next, **cd** into the TBTK folder and execute

```
source init_session.sh
```

3.5 Setup a new project

It is recommended to create a separate folder, here referred to as **TBTKProjects**, and for each project create a subfolder according to

```
TBTKProjects/project1/
TBTKProjects/project2/
```

When starting a new project, it is recommended to copy one of the Template-projects in the directory **TBTK/Templates/** into **TBTKProjects/**, and to then modify the code of these templates. TBTK provides the command **TBTKCreateProject** to simplifying this procedure. For example, assuming that the current directory is the **TBTKProjects** folder, use the following line to create a new project that builds on top of the **EmptyProject** template

```
TBTKCreateProject NewProject EmptyProject
```

Enter into the new project folder

```
cd NewProject/
```

Modify the source code

```
nano src/main.cpp
```

Compile (create executeable file from the source code). The resulting binary will be located in **build/a.out**.

```
make
```

Run the program

```
./build/a.out
```

4. Getting started with TBTK

4.1 Empty project

To get familiar with the basic structure of a TBTK program we first take a look at the **EmptyProject** that was created, compiled, and executed at the end of the previous chapter. To the experienced C++ programmer this example should be self-explanatory. However, we go through it in detail to allow users without prior knowledge of C++ to be able to write code as well. The code looks as follows

```
#include "Streams.h"

#include <complex>

using namespace std;
using namespace TBTK;

const<double> i(0, 1);

int main(int argc, char **argv){
  Streams::out << "Hello quantum world!\n";

  return 0;
}
```

The first line gives us access to the TBTK library class **Streams**, which is used to handle output. For each componenent of the TBTK library that is used in a program, a corresponding **include** statement is needed. Similarly, the third line gives access to the C++ standard library class **complex**, which is used to handle complex numbers.

All components of the TBTK library are contained in the namespace **TBTK**, while those in the C++ standard library are contained in the namespace **std**. What this means is that by default it is necessary to refer to them as **TBTK::Streams** and **std::complex**, respectively. However, most of the time this becomes unecessarily tedious. Line five and six are used to allow for them to instead be reffered to as **Streams** and **complex**, respectively.

On line seven the complex number **i** is defined. Although **i** is not used in this example, it is so often useful to have when setting up models and doing calculations that it is included by default in all TBTK templates.

The program itself starts at the **main** function. The function definition itself tells us that it is a function that takes two arguments in the form of an **int**

(integer) and a **char**** (pointer to character array) and returns an **int**. This is the standard way of defining an entry point in a C++ program, and we therefore do not go into details here. We only point out that the final statement **return 0;**, is the point where the actual **int** is returned.

The remaining line is the first line inside the **main** function, which simply prints **Hello quantum world!** to **Streams::out**, which by default means printing to the terminal. The character **\n** at the end of the string is a new line character that ensures that any following output will occur on the next line. In addition to the standard output stream **Streams::out**, TBTK also provides the possibility to write to the error stream **Streams::err** and the log stream **Streams::log**. Of course, the user may also use any other output streams such as the C++ standard library streams in **iostream**.

4.2 Basic diagonalization

4.2.1 Introduction

Here we take a look at the **BasicDiagonalization** template to understand the structure of a complete TBTK application. The basic structure is the same as for the **EmptyProject** above and we will therefore skip to explain code that already was explained there.

4.2.2 Problem formulation

The model that is going to be solved here is

$$H = -\mu \sum_{\mathbf{x}\sigma} c^\dagger_{\mathbf{x}\sigma} c_{\mathbf{x}\sigma} - t \sum_{\langle \mathbf{x}\mathbf{y}\rangle\sigma} c^\dagger_{\mathbf{x}\sigma} c_{\mathbf{y}\sigma}, \tag{4.1}$$

where $c^\dagger_{\mathbf{x}\sigma}$ ($c_{\mathbf{x}\sigma}$) is a cretion (annihilation) operator for a spin σ on site $\mathbf{x}$, angle brackets indicates summations over nearest-neighbor indices, μ is the chemical potential, and t is the nearest-neighbor hopping strength. The lattice is taken to be a two-dimensional square lattice with non-periodic boundary conditions and size 20×20. In particular, this means that the site indices are of the form $\mathbf{x} = (x, y)$. The quantities that we are going to extract from the model are the eigenvalue spectrum and the density of states.

Before moving on to implementation, the first step is to write the Hamiltonian on the form

$$H = \sum_{\mathbf{ij}} a_{\mathbf{ij}} c^\dagger_{\mathbf{i}} c_{\mathbf{j}}. \tag{4.2}$$

This is achieved by introducing the notation $\mathbf{i} = (x, y, \sigma)$ and writing

$$a_{\mathbf{ij}} = \begin{cases} -\mu & \mathbf{i} = \mathbf{j}, \\ -t & \mathbf{i} = \mathbf{j} \pm \hat{x}, \\ -t & \mathbf{i} = \mathbf{j} \pm \hat{y}, \\ 0 & otherwise, \end{cases} \tag{4.3}$$

where $\hat{x}$ ($\hat{y}$) is the lattice vector in the x-(y-)direction. In TBTK these coefficients are referred to as hopping amplitudes irrespectively of whether they correspond to a "real" hopping such as a nearest-neigbor hopping or a term such as for example the chemical potential. The indices **i** and **j** are often referred to as **to** and **from** indices, drawing from the observation that the individual terms $a_{\mathbf{ij}} c^\dagger_{\mathbf{i}} c_{\mathbf{j}}$ annihilates a state on site **j** and recreates it on site **i** with amplitude $a_{\mathbf{ij}}$.

4.2.3 Workflow

One of the design goals with the TBTK library is to allow for the implementations of specific problems to result in highly readable code, where the physics can be read of immediately from the code. This is ensured by proper design decissions being taken on many different levels : from the choice of data structures, the naming schemes of individual classes, the way in which different classes interact with each other, all the way up to the overall structure of the typical TBTK application. As a result, the implementation of a new problem usualy follows a simple workflow that is divided into three steps: **setup model**, **choose solver**, and **extract properties**. In addition to being conceptually useful and enhancing the readability of the code, the clear separation of the three stages allows for them to be modified independently. That is, the **model**, **solver**, or **extracted properties** can each be changed without having to modify any of the other two.

4.2.4 Header files

The first part of any program is the header file section. Apart from the already introduced standard library header file for complex numbers, the **BasicDiagonalization** template includes the following files

```
#include "DOS.h"
#include "DiagonalizationSolver.h"
#include "DPropertyExtractor.h"
#include "EigenValues.h"
#include "FileWriter.h"
#include "Model.h"
```

The purpose of each of these components will be explained as they are encountered in the code.

4.2.5 Parameters

The two first lines of the main function reads

```
const int SIZE_X = 20;
const int SIZE_Y = 20;
```

This defines two constants that will be used to determine the size of the model. Similarly the next two lines are used to specify the model parameters

```
complex<double> mu = -1.0;
complex<double> t = 1.0;
```

4.2.6 Setup model

The modeling stage begins by creating a **model** with the line

```
Model model;
```

Next all the indices of the model are looped over

```
for(int x = 0; x < SIZE_X; x++){
  for(int y = 0; y < SIZE_Y; y++){
    for(int s = 0; s < 2; s++){
      //Some code here
    }
  }
}
```

This can usefully be thought of as carrying out the summation in Eq. (4.2). The purpose of this loop is to add all the **HoppingAmplitude**s ($a_{\mathbf{ij}}$) to the model. For the chemical potential this is done with the line

```
model.addHA(HoppingAmplitude(-mu, {x, y, s},  {x, y, s}));
```

The arguments supplied to the **HoppingAmplitude** should in order be understood to be the value, the first, and the second index of $a_{\mathbf{ij}}$, respectively.

The nearest-neighbor hopping is slightly more complicated as it involves the hopping between sites with different indices and we need to ensure that we handle boundary terms appropriately. We note that the hermiticity of the matrix means that for every term $a_{\mathbf{ij}}c_{\mathbf{i}}^{\dagger}c_{\mathbf{j}}$ with $\mathbf{i} \neq \mathbf{j}$, there is a corresponding term $a_{\mathbf{ji}}^{*}c_{\mathbf{j}}^{\dagger}c_{\mathbf{i}}$. The current code utilizes this by only explicitly adding the "forward hopping" term using the expression (and a similar expression for the y-direction)

```
if(x+1 < SIZE_X)
  model.addHAAndHC(
    HoppingAmplitude(-t,  {(x+1)%SIZE_X, y, s}, {x, y, s})
  );
```

Here the if-statement is added to ensure that the boundary term is excluded. Further, the function **addHAAndHC** is used rather than **addHA** in order to

simultaneously add the hermitian conjugate. If **addHA** had been used instead, a corresponding line adding the ”backward hopping” term would have been needed, where the if-statement instead would have had to guard agains the addition of a boundary term at $x = 0$. We also point out that rather than letting the first index x-component be given by x+1, it is here given as (x+1)%SIZE_X, where % is the modulus sign. This allows for periodic boundary conditions to be implemented by simply removing the if-statement.

The final step in setting up the model is the line

```
model.construct();
```

This sets up a mapping from all the indices that have been provided to the model on a ”physical” form {x, y, s} to the corresponding Hilbert space indices, thereby making the information fed to the model available on a format that enables the implementation of efficient algorithms for solving the model.

4.2.7 Choose solver

The next step is to choose a solver, which here is done by setting up and running a **DiagonalizationSolver** using the lines

```
DiagonalizationSolver dSolver;
dSolver.setModel(&model);
dSolver.run();
```

The first line creates the solver, while the second instructs the solver that the model it is going to solve is the model setup in the previous step. In the third line the model is finally diagonalized, which puts the solver into a state where it is ready for properties to start being extracted.

4.2.8 FileWriter

In the next two lines we initialize the **FileWriter** using

```
FileWriter::setFileName("TBTKResults.h5");
FileWriter::clear();
```

The first line sets the file name of the file that all data will be written to, while the second line deletes any file in the current directory with the same name. In particular, we note that TBTK uses the file format HDF5, which is file format constructed to handle scientific data and which has wide support in languages such as c++, matlab, python, etc.

4.2.9 Extracting properties

The final step in the code is to extract properties, which is done with the help of a **PropertyExtractor**. The purpose of the **PropertyExtractor**s are to hide

unecessary numerical details associated with the particular solvers and instead provide a physically relevant interface. This makes it possible to with a minimal amount of extra work exchange one solver for another without affecting most of the code associated with the property extraction. Because we here use the **DiagonalizationSolver**, we setup the corresponding **DPropertyExtractor** using

```
DPropertyExtractor pe(&dSolver);
```

Some of the properties, such as the DOS in this case, are calculated on a specified interval with a certain energy resolution. In this tempalte this information is set with the following four lines

```
const double UPPER_BOUND = 6;
const doulbe LOWER_BOUND = -4;
const int RESOLUTION = 1000;
pe.setEnergyWindow(LOWER_BOUND, UPPER_BOUND, RESOLUTION);
```

These lines instructs the **PropertyExtractor** that when a quantity that is a function of E is calculated, it shoud do so for 1000 evenly spaced points on the interval $[-4, 6]$.

The first property to be calculated is the eigen value spectrum.

```
Property::EigenValues *ev = pe.getEigenValues();
FileWriter::writeEigenValues(ev);
delete ev;
```

Here the first lines extracts the eigen values and sets up the variable **ev** to point to these. The eigen values are then written to file using the **FileWriter**,which permanently store them in order to give access to the results also after the program has finished. The third line frees memory associated with the eigen values. This is a subtle but important step, and the user should be aware that whatever quantity that is received from a **PropertyExtractor** has to be freed in this way. If this is not done, memory leaks can arise that are hard to debug and which may cause unexpected behavior.

Having understood how the eigen values are extracted and written to file, it is now straight forward to understand how also the DOS is calculated and saved

```
Property::DOS *dos = pe.calculateDOS();
FileWriter::writeDOS(dos);
delete dos;
```

Note the identical syntax. Similarly every other property that can be extracted by the **PropertyExtractor** has a corresponding **FileWriter** call for writing the property to file.

4.2.10 Plotting

After the program has been compiled and executed, as described in section 3.5, the project folder contains a file named **TBTKResults.h5**. The data stored in this file can immediately be plotted using the templates plot script by typing

```
./plot.sh
```

The resulting output is found in the **figures** folder.

Let us take a closer look at the plot script itself. On inspection we find the following two commands inside

```
TBTKPlotDOS.py TBTKResults.h5 0.15
TBTKPlotEigenValues.py TBTKResults.h5
```

These two lines plots the DOS and eigen values respectivly. Similar functions exist for each property that can be extracted using the **PropertyExtractor** and always takes the filename of the .h5 file as first argument. However, depending on which property that is plotted, the number of parameters that follows can vary. For example, in the above example we see that the eigenvalues do not require any additional parameters. In contrast, to plot the DOS an additional parameter is needed which specifies the sigma that is used for applying Gaussian smoothing.