Computational Methods in Solid State Theory

Harald O. Jeschke

Institut für Theoretische Physik
Goethe-Universität Frankfurt am Main
Max-von-Laue-Straße 1, 60438 Frankfurt
Email: jeschke@itp.uni-frankfurt.de

April 10, 2012

Abstract

- This lecture gives an overview of computational methods that are important for the condensed matter theorist.
- The lecture will focus on methods that are suitable for solving model Hamiltonians in solid state theory.
- Part of the lecture will deal with dynamical mean field theory, a method that is approximate in finite dimension but has been increasingly successful over the last twenty years.
- The second part of the lecture will deal with numerical methods like exact diagonalization and quantum Monte Carlo.
- The lecture will be enriched by practical exercises and discussion of available software or libraries and methods of implementation.

Incomplete list of topics

1. Tight binding
2. Density functional theory
3. Hartree Fock mean field
4. Matsubara Greens functions; analytic continuation
5. Random phase approximation
6. Exact Diagonalization
7. Monte Carlo
8. Quantum Monte Carlo

Motivation

- **Evaluation of theoretical models**: Many theoretical approaches involve a numerical evaluation of resulting equations as a final step. This is not a trivial part: Success of a theory often depends crucially on the feasibility of its numerical evaluation.
- **Making contact to real materials**: Nearly all measurable quantities in real materials are off limits for analytical computation. If you want to compare a theory to experiment, numerical methods are needed to account for the complexity of chemical interactions, real lattice structures, interplay of various phenomena present at the same time, and so on.
- **Introducing computational physics**: This area of physics is of growing importance as computers become more powerful and as more and more nontrivial aspects of experiment and technology can be computed or simulated, yet it is hardly mentioned in physics classes.

My perspective

Fields of research	Methods I use
Tight binding molecular dynamics on time dependent potential energy surfaces	Matrix diagonalization Integration of differential equations Fast Fourier transform
Dynamical mean field theory for lattice models (Hubbard, Anderson)	Integral equations Splines Exact diagonalization
Ab initio density functional theory	Minimization techniques

Aims of the lecture

- Giving a feeling for "What's inside the box?" for a number of computational methods.
- Discussion of strengths, weaknesses, approximations, pitfalls of widely used methods.
- Providing some insight into what can be calculated and how in theoretical physics. Knowing how to calculate a quantity makes it more accessible or intelligible.
- Pointing out some sources of information and software for computational tasks.
- Giving you the confidence "I can calculate that, too!".
- Studying many interesting details. Everybody has heard "exact diagonalization" or "quantum Monte Carlo" as catchwords, but what are actually the parameters influencing their results?

What are your additional wishes?

C++ crash course

Why C++?

- C++ is a multi-paradigm programming language that supports procedural programming, data abstraction, object-oriented programming, and generic programming.
- It is used widely in open source software. It is a non-proprietary language. Good free compilers are available.
- You can call routines and functions written in other languages from C++ (examples Fortran, Perl).
- It is fully developed and standardized and suitable for any size of project. It is sufficiently general to serve for an extremely wide range of purposes and problems.
- It is a compiled language that can achieve highest levels of performance (if you make sure performance critical parts are C like or can be efficiently optimized by the compiler).

C++ crash course

But: C++ may not be suitable for every problem.

- One always needs to preserve the flexibility to learn an additional language (say perl, python).
- It is often necessary read and modify existing codes (for example Fortran77/90).

This crash course is intended to give you a taste of C++.

- Numerical methods become practical and useful when implemented on a computer.
- Preparation for the discussion of implementation details.

Compilation of minimal C++ program

The minimal C++ program is

```
int main() {}
```

To compile and run this program, enter the following three lines at a unix command prompt:

```
$ echo "int main() {}" > example0.cc
$ g++ example0.cc -o example0
$ ./example0
```

This creates the program file `example0.cc` (line 1), compiles it (line 2) and executes it (line 3).
The g++ command (gnu c++ compiler) does a number of things; try using its verbose mode by typing `g++ -v` in the example.

Compilation of minimal C++ program I

- It first calls `cc1plus` to produce assembler code (with extension .s, written to some file in /tmp).
- Next, it calls the assembler `as` to produce an object file (with extension .o, again only temporarily in /tmp).
- Finally it calls `collect2`. `collect2` looks at the object files (the one produced by `as` and others that belong to C++), builds an additional object if needed, and invokes the linker `ld` which produces the executable.
- For a slightly more complicated program that contains a preprocessor directive like `#include`, `cc1plus` would also call the preprocessor `cpp`.

Compilation of minimal C++ program II

To understand the compilation process better, you can produce intermediate results:

```
$ g++ -E example0.cc -o example0.E
$ g++ -S example0.cc -o example0.s
$ g++ -c example0.cc -o example0.o
$ g++ example0.o -o example0
```

The first line produces precompiled code (still C++), the second assembler code; they are text files and you can inspect them.
The third line produces the object file which you can analyse with the command `nm`:

```
$ nm example0.o
                 U __gxx_personality_v0
0000000000000000 T main
```

Compilation of minimal C++ program III

This shows you the symbols in the object file, one **U**ndefined, another in the **T**ext section. The last of the four lines above invokes the linker and produces a dynamic executable; you can find out which shared libraries it looks for at runtime using the `ldd` command:

```
$ ldd example0
linux-vdso.so.1 =>  (0x00007fffaabff000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00002b57c00ae000)
libm.so.6 => /lib/libm.so.6 (0x00002b57c03b5000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00002b57c0638000)
libc.so.6 => /lib/libc.so.6 (0x00002b57c084e000)
/lib64/ld-linux-x86-64.so.2 (0x00002b57bfe8b000)
```

Standard Library I

Output

In C++, the inevitable "Hello, world!" program reads like this:

```
#include <iostream>
int main() {
  std::cout << "Hello, world!\n";
}
```

The first line makes the precompiler include the input/output streams library. The only command in `main()` uses the standard `cout` stream to write the "Hello, world!" string and a newline character to standard output. Values sent to `cout` with the << operator are converted into a sequence of characters.
The `std::` stipulates that the `cout` of the standard namespace is to be used, rather than some other `cout`.

Standard Library II

You can make the `std` name accessible without `std::` by dumping the `std` namespace into the global namespace:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello, world!\n";
}
```

The iostream library defines output for every built-in type. It is easy to define output for a user-defined type.

Input

`cin` is the standard input stream, and >> is used as input operator. The following program performs cm $\leftrightarrow$ inch conversions:

Standard Library III

```
#include <iostream>
using namespace std;
int main() {
  const double factor = 2.54;
  double x, in, cm;
  char ch=0;
  cout << "enter length (x cm or y in): ";
  cin >> x >> ch;
  switch(ch) {
  case 'i':
    in=x; cm=x*factor; break;
  case 'c':
    in=x/factor; cm=x; break;
  default:
    in=cm=0; break;
  }
  cout << in << " in = " << cm << " cm\n";
}
```

Standard Library IV

Strings

The standard library provides a string type and various string operations, such as concatenation that is written like addition:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  string s1="Hello";
  string s2="world!";
  string s3=s1+", "+s2+"\n";
  cout << s3;
}
```

```
$ ./example2
Hello, world!
```

Standard Library V

Further examples of string operations:
`substr()` that returns a string that starts at the index given in the first argument (C++ counting starts at 0!), of the length given by the second argument.
`replace()` replaces part of a string with another, starting at the index in the first argument and replacing the number of characters given by the second.

```
  string s1="Good bye, world!\n";
  string s2=s1.substr(0,8)+"!\n";
  s1.replace(0,8,"Hello");
  cout << s1 << s2;
```

```
$ ./example3
Hello, world!
Good bye!
```

Built-in types

C++ provides a number of types; beyond that, the user can define his own shorthand for complicated types using `typedef`. Besides, composite objects can be defined using classes.
Some important types:

- There is the logical `bool` with the values `true` or `false` (1 or 0).
- There is `char` with a range from -128 to 127, represented by 1 byte. There is `unsigned char` with a range from 0 to 255. There are many different types of `int`, like `short int`, `int`, and `long int` which are all by default `signed`. The number of bytes corresponding to them and thus the ranges are machine dependent.
- Then there is `float`, `double` and `long double`, providing floating point numbers of different precision (and thus different overflow and underflow values), again machine dependent.
- Complex types in C++ are provided by a standard library `complex`.

Arrays I

Dynamic memory allocation in C++ is performed by the `new` command which returns a pointer to the memory location of the first element of the array.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
  int num=20;
  double* vec=new double[num];
  if(vec == NULL) {
    cerr << "Not enough memory";
    exit(1);
  }
  for(int i=0; i<num; i++) vec[i]=0.0;
}
```

Arrays II

- The `NULL` pointer is returned if there was not enough memory available.
- The elements of the newly allocated array are not initialized; therefore, they are set to zero in the example program.
- The `new` command can also allocate memory for arrays of any user defined type or class.
- Elements are accessed by square brackets.
- The first element has index 0, the last index `num-1`.
- The programmer is responsible for making sure the array index doesn't leave this range; otherwise, a Segmentation Fault runtime error can occur (but is unfortunately not guaranteed to occur).
- Segmentation Fault errors are among the hardest to trace and eliminate. Safe programming strategies are important.

Flow of control I

Control flow statements allow variations in the sequential order of execution.
In C/C++ no direct jumps ("`goto`") are available.

Conditional execution

One type of conditional execution was realized by the `switch()` command above.
The following program simulates the flipping of a coin:
Depending on the value of a pseudo random number produced by `rand()` it either executes the command printing "head" or the one printing "tails".
The standard library `ctime` provides the command `time()` which returns the time since January 1, 1970 in seconds. This value is used as seed for the pseudo random number generator which would otherwise give the same pseudo random number each time the program is executed.

Flow of control II

```
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;
int main() {
  srand(time(0));
  int a=rand();
  if(a < RAND_MAX/2) {
    cout << "head\n";
  }
  else {
    cout << "tails\n";
  }
}
```

Flow of control III

Loops

The
`do {...} while(condition);`
statement loops until the condition becomes true.

If the condition is set to the `bool` value `true` as in the following example, the loop is endless unless terminated by a `break` statement.

In this example, the same result could have been achieved with the
`while(condition) {...};`
statement.
But otherwise it matters if the condition is tested before or after execution of the loop body!

Flow of control IV

```
#include <iostream>
#include <ctime>
using namespace std;
int main() {
  srand(time(0));
  int counter=0;
  do {
    int a=rand();
    if(a < RAND_MAX/2)
      break;
    counter++;
  } while(true);
  cout << "Threw " << counter << " times ";
  cout << "head before throwing tails." << endl;
}
```

Flow of control V

Indexed loops

The indexed iteration in C++ has the syntax `for(initial-statement; test-expression; iteration-expression) {...}`. The example program takes an integer number and calculates the factorial:

```
#include <iostream>
using namespace std;
int main() {
  int a, fact = 1;
  cout << "Factorial of: "; cin >> a;
  for(int i=a; i>1; i--) {
    fact *= i;
  }
  cout << " is " << fact << endl;
}
```

Flow of control VI

- The index variable `i` is only defined within scope of the `for` loop.
- It is first set equal to `a`, and if it is larger than 1, the loop body is executed. Then `i` is diminished by 1.
- If it is still bigger than 1, the loop is executed again until the condition is not fulfilled any more.
- If the condition doesn't hold at the first test, the loop body is never executed.

Functions

In C++, a function name is preceded by the return type and has the argument list in round brackets. Each argument has a type. The following example program calculates the square of a number.
Before `main()`, the prototype of a function has to be given. This is mandatory except if the function is fully defined before being used.

Flow of control VI

```
#include <iostream>
using namespace std;
double sqr(double& number);
int main() {
  double a, b;
  cout << "Number to square: "; cin >> a;
  b=sqr(a);
  cout << "The square is " << b << endl;
}
double sqr(double& number) { return number*number; }
```

The `double&` indicates that the argument is passed by reference, which means that the function only gets a pointer to the argument. Without `&` the argument would be passed by value, meaning that it would be copied into a temporary.

Classes

- Classes are at the heart of C++'s object-orientation (OO).
- When you design a program the class design is crucial. Simplicity, efficiency and maintainability are important factors.

What is the basic idea? If you were using an older, non-OO language to write a database containing details about people (their weight, height, name, etc) you might have to have an array for each feature, so the details for a particular person might be contained in weight[237], height[237], name[237], etc. Object-orientated languages let you design more natural programs by creating a person class, with each object containing all the information about a person, as well as ways to process and display the information.

Classes: Public and private members I

The following defines a class named `person` that has one `public` member:

```
class person {
public:
  double height; // height in meters
};
```

That means that you can now create an instance of the class and assign a value to the member by writing:

```
person p;
p.height = 1.53;
```

But as we can directly access the (`public`) member, we can also write nonsense into it (for example `p.height = 15.3`).
Another problem of `public` access is a lack of control over side effects.

Classes: Public and private members II

A solution to the unrestricted access problem is to make `height` a `private` member, so that it can't be changed from the outside (not even read). But member functions (i.e. functions that are part of the class) can access private members, so if these functions are public, the outside world can access the `height` member in a controlled way via these functions. The following example uses two such functions to get the height and to set it with some error-checking.

```
class person {
private:
  double height;
public:
  bool set_height(double h)  {
    if (h<0 or h>3) return false;
    else { height=h; return true; }
  }
  double get_height() { return height;}
};
```

Classes: Public and private members III

Now the uses of the class look like this:

```
int main() {
  person p("Mike",1.60);
  if(!p.set_height(3.20)) {
    cerr << "Implausible height value!\n"; exit(1);
  }
  double h=p.get_height();
}
```

and our program would crash:

```
$ ./example14
Implausible height value!
```

This only demonstrates in principle how checks can be built into the data structure.

Classes: Constructors and destructors I

A constructor is a function that explicitly initializes an object. A class should always have one or more constructors in order not to rely on default constructors with possible unwanted results. There are two possibilities; using the member initialization list:

```
class person {
private:
  double height;
  string name;
public:
  person(const string& _name, double _height):
    name(_name), height(_height) {}
};
```

and making assignments in the constructor body:

```
  person(const string& _name, double _height) {
    name = _name;
    height = _height;
  }
```

Classes: Constructors and destructors II

The first option may often be better as it also works for `const` members and can be more efficient.

A destructor specifies what should happen when a class goes out of scope; one can rely on default destructors except if memory has been allocated with `new`; to avoid memory leakage one should delete the allocated memory:

```
~vector() { delete[] array; }
```

As memory leakage is a serious problem, it is necessary to figure out if the compiler has a way of knowing at which point to deallocate memory.

On the other hand, it may be useful for performance reasons to disable the destruction of objects by making them `static`.

Templates I

C++'s templates are parameterized types. They support generic programming by allowing the same methods to deal with various data types. For example, the following code swaps 2 integers in usual style:

```
void swap (int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

If you wanted to swap other types of variables (including user defined ones) you could copy this code, replacing int by (say) double. Here, templates come in:

```
template <class T>
void swap (T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

Templates II

Here the name T is arbitrary (like a formal parameter in a function definition). Note that the function is written much as before. When the compiler is now given the following code it will create a version (instantiation) of swap for each type required:

```
int a = 3, b = 5;
double f=7.5, g=9.5;
swap (a, b);
swap (f, g);
```

This brief introduction to C++ leaves out many interesting aspects of C++ like casting, polymorphism, inheritance...

Some nice online instruction on C++ can be found at
`http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/doc/doc.html`