# C++的一些有用的高级特性

编程语言也在随着时间不断进步，最新的C++版本是C++11。C++11是C++标准委员会于2011年发布的新的C++标准，增加了众多实用的功能。 这些特性，不仅给程序员带来了巨大的方便，而且使得C++焕然一新，C++之父Bjame Stroustrup说C++11就像一个新语言。考虑到C++11是比较新的技术，大多数人对其并不熟练，因此单列一章进行讲述。C++11带来的新特性有很多，本章的目的并不是对C++11的新特性做一个全面而详细的介绍，而只是对开发过程中用到的特性做简单介绍，需要了解更多信息的读者可以参考相应的书籍[[1]](#endnote-1)。由于C++11是比较新的版本，所以编译器对其支持并不是很好，目前对C++11支持最好的是编译器是gcc，因此推荐使用gcc来编译。

OpenMP是一套并行计算的API，它支持C语言，C++以及Fortran。OpenMP使用起来十分简单，因此在开发过程中也大量应用OpenMP来进行并行计算。

## 用户自定义字面量

用户自定义字面量是C++11新引入的特性之一。

字面量是代码中直接写出来的值。比如说：

* double a = 2.5; 这里面的2.5就是个字面量。
* double b = a; 这里面的a就不是个字面量。
* double b = a + 3; 这里面的3就是个字面量。
* char \*s = “Hello World!!!”; 这里面的“Hello World!!!”就是个字面量。
* char a = ‘4’; 这里的’4’就是个字面量。

C++11支持用户自定义字面量。自定义字面量需要编写字面量运算符。字面量运算符有两种模式：生模式和熟模式。

生模式的语法如下：

|  |
| --- |
| 返回类型 operator”” \_字面量标识 (const char \*a,size\_t b) {  你的代码  } |

注意双引号与下划线之间有空格。

例如，如果定义这样一个运算符：

|  |
| --- |
| int operator”” \_length(const char \*a,size\_t b) {  return b;  } |

这样的话，就可以这样使用字面量了：

|  |
| --- |
| int len1 = “abcdefg”\_length;  int len2 = “PB10203091”\_length; |

这样可以得到abcdefg与PB10203091两个字符串的长度。

熟模式定义字面量的语法如下：

|  |
| --- |
| 返回类型 operator”” \_字面量标识(unsigned long long) {  你的代码  } |

或者

|  |
| --- |
| 返回类型 operator”” \_字面量标识(long double) {  你的代码  } |

注意参数列表中必须是unsigned long long或者long double，不可以是其他类型（比如int, double等都不行）

比如如果这样定义了一个字面量：

|  |
| --- |
| double operator”” \_KHz(long double f) {  return 1000\*f;  } |

这样的话，就可以这样使用字面量了：

|  |
| --- |
| double f1 = 3.5\_KHz;  double f2 = -2.5\_KHz; |

这样就可以自动把KHz转换成Hz。

## Lambda表达式

Lambda表达式通常出现在我们需要一个函数，但又不想费神去命名一个函数的时候。比如说，现在有一个函数plot(f,l,r)，该函数接受3个参数，第一个参数f是一个回调函数，第二个参数l以及第三个参数r是两个double类型的数，plot函数的作用是绘制函数f在之间的图像。我们现在的任务是，调用plot函数绘制函数在之间的图像，其中要用户输入。要调用plot，必须给他提供一个回调函数。如果没有Lambda表达式，我们只能手工创建一个函数f，函数接受x为参数，计算并返回y，然后将这个函数作为回调函数传给plot。关键麻烦的一点是，函数值计算的时候要用到用户输入的变量，这个时候还得给将作为参数传递给函数f，但是f一旦接受四个参数，它就不符合plot对回调函数的要求（只接受一个参数）。这个问题一种解决方法是创建一个类，重载它的()运算符，另一种解决方法是则是使用bind。老版本的C++中的bind可以说是相当难用，所以bind用起来也相当麻烦。C++11中的bind有所改善，但是仅仅为了一个回调函数这么折腾也够麻烦了。还有一种解决方法是使用全局变量，但是全局变量在大的项目里面会带来深层次的问题，比如说变量重名。大的项目往往是好几个人同时参与开发，如果每个人都在需要的时候随随便便创建一个全局变量来满足自己的需要，那么不同人之间的变量重名的几率非常大。这个时候就会产生难以调试的bug。

作为例子，老版本的C++代码写起来大概是这个样子：

|  |
| --- |
| class ff {  public:  double a;  double b;  double c;  double operator()(double x) {  return a\*x\*x\*x + b\*sin(x) + c\*exp(x);  }  };  int main(){  ………  ff f;  cin >> f.a;  cin >> f.b;  cin >> f.c;  plot(f,-1,1);  ……..  } |

但是如果有了Lambda表达式，则代码可以简化为：

|  |
| --- |
| int main(){  ………  double a,b,c;  plot([=](double x){ return a\*x\*x\*x + b\*sin(x) + c\*exp(x); },-1,1);  ……..  } |

这样代码就显得紧凑多了，同时也更有条理了。

Lambda表达式的语法如下：

|  |
| --- |
| [捕获列表](形参列表) ->返回值类型{ 函数体 } |

捕获的意思是，允许函数体内部访问当前位置的局部变量。捕获包括按值捕获跟按引用捕获两种类型。按值捕获类似函数调用时的参数的按值传递，Lambda的函数体得到的仅仅是被捕获变量的副本，函数体对被捕获变量的修改不会影响到原变量的值。按引用捕获则类似于函数调用时的参数的按引用传递，Lambda的函数体得到的是原变量的引用，任何对被捕获变量的修改都会写入到原变量。捕获列表的写法如下：

* [] 不捕获任何变量，函数体内将不能使用任何当前位置的局部变量
* [x,&y] 按值捕获x，按引用捕获y
* [=] 按值捕获所有被用到的变量
* [&] 按引用捕获所有被用到的变量
* [&, x] 按值捕获x，按引用捕获其他用到的变量
* [=,&z] 按引用捕获z，按值捕获其他用到的变量

形参列表类似函数定义的形参列表，如果函数不需要任何形参，则形参列表及其外面的圆括号均可以省略。返回值类型是函数的返回值类型，如果函数体只有一个return语句，则可将返回值类型及其前面的“->”一同省略，此时返回值的类型自动根据return语句判断。

## 初始化列表

旧版C++中，使用初始化列表仅能用来初始化数组，而不能初始化其他的元素，例如：

* int a[3] = { 1,2,3 }; //正确
* std::vector<int> a = { 1,2,3 }; //错误

而在C++11中，引入了统一初始化的方法，可以使用初始化列表来初始化STL中的容器，以及自己定义的类型的对象。例如：

* std::vector<int> a = { 1,2,3 }; //正确

C++11将会把初始化列表的概念绑到std::initializer\_list类上。如果想让自己编写的类能够使用初始化列表来初始化对象，则需要创建接受std::initializer\_list参数的构造函数。例如：

|  |
| --- |
| class SequenceClass{  public:  SequenceClass(std::initializer\_list<int> list);  }; |

这样，就可以允许SequenceClass类使用一连串整数构造：

|  |
| --- |
| SequenceClass someVar = {1, 4, 5, 6}; |

## 模板元编程

C++11引入的很多新的特性使C++11的模板元编程能力大大增强。

### 自动类型推导与decltype关键字

C++11开始支持自动类型推导。声明变量的时候不需要指定类型，编译器自动推导变量类型。

语法：

|  |
| --- |
| auto 变量名 = 初始值; |

auto是C++的关键字，由于编译器需要根据初始值推断变量的类型，所以使用auto声明变量的时候必须给变量赋初值。

例如：

|  |
| --- |
| auto i = 1; //相当于int i = 1;  auto b = 2.5; //相当于double b = 2.5; |

除了自动类型推导以外，C++11还提供decltype的关键字，它的用法如下：

decltype(表达式)是一个类型，类型为表达式结果的类型。例如：

* decltype(3)相当于int
* decltype(9.4)相当于double
* decltype(9.2+2)相当于double
* decltype(1>2)相当于bool

需要注意的是，类型是在编译时就确定的，所以表达式在程序执行过程中并不会真正被执行。例如：

* int i=0; decltype(i++) j=0; std::cout << i; //输出0，而不是1

### 新的函数定义语法

旧版本C++中，函数定义的语法为：

|  |
| --- |
| 返回值类型 函数名(形参列表) {  函数体  } |

C++11中引入了一种新的函数定义语法：

|  |
| --- |
| auto 函数名(形参列表) ->返回值类型 {  函数体  } |

旧的语法在C++11中可以继续使用。

新的语法的好处是，返回值类型位于形参列表之后，这样就可以在返回值类型中使用一些类型表达式，例如，下面的模板函数计算两个变量的和，返回值的类型“自动推导”：

|  |
| --- |
| template<typename T1,typename T2>  auto myplus(T1 arg1,T2 arg2)->decltype(arg1+arg2) {  return arg1+arg2;  } |

这个函数的好处是，他能够计算任意两个变量的和，这两个变量可以是不同的类型，只要两个类型之间的operator+有定义即可。调用myplus(1,2)返回的是int，调用myplus(1,2.0)则返回double。在之前的版本的C++中，这个功能是实现不了的。

之所以要引入新的语法，而不用旧的语法，是因为，如果要实现返回值类型的“自动推导”，使用旧的语法必然要这样写代码：

|  |
| --- |
| template<typename T1,typename T2>  decltype(arg1+arg2) myplus(T1 arg1,T2 arg2) {  return arg1+arg2;  } |

这样写代码的问题是，在编译器编译到decltype(arg1+arg2)的时候，arg1跟arg2还没有定义，这样就会出现变量未定义先被使用的情况。

### 递归模板

C++支持递归模板，例如使用如下方法在编译时计算阶乘：

|  |
| --- |
| template<int n>  class factorial{  public:  const static int value=factorial<n-1>::value\*n;  };  //通过特化模板来终止递归  template<>  class factorial<0>{  public:  const static int value=1;  }; |

### 变长参数模板

C++11引入变长参数模板的支持。

在模板声明的语句中，使用typename … Tn或者class … Tn来表示一个“模板形参包（template parameter pack）”其中typename跟class都是关键字，Tn是包的名称，可以自己随便取自己喜欢的名称。模板形参包是一个可以接受零个或多个模板实参的模板形参，例如，如果有声明：

template<class ... Types> struct Tuple { };

则：

* Tuple<> t0; //正确：Types不含任何实参
* Tuple<int> t1; //正确：Types含有一个实参：int
* Tuple<int, float> t2; //正确：Types含有两个实参：int和float
* Tuple<0> error; //错误：0不是一个类型

如果Tn是一个模板形参包，则可以使用Tn … args来表示一个“函数形参包（function parameter pack）”。函数形参包是一个接受零个或多个函数实参的函数形参。例如，如果有声明：

template<class ... Types> void f(Types... args);

则：

* f(); //正确：Types与args均不含有任何实参
* f(1); //正确：Types含有一个实参int，args含有一个实参1
* f(2, 1.0); //正确：Types含有两个实参int和double，args含有两个实参2以及1.0

变长参数模板通常需要同模板递归结合使用。比如说，下面例子计算N个int的和：

|  |
| --- |
| int s(int a) { return a; }  template<typename...T> int s(int a, T...b) { return a+s(b...); } |

当我们调用s(1,2,3,4)的时候，编译器的处理过程如下：

1+s(2,3,4)

s(1,2,3,4)

int s(int a, T...b) { return a+s(b...); }

int s(int a, T...b) { return a+s(b...); }

2+s(3,4)

int s(int a, T...b) { return a+s(b...); }

3+s(4)

int s(int a) { return a; }

## C++中使用OpenMP进行并行计算

C++中使用OpenMP的方法比较简单。如果你有一个for语句，for语句中的每次迭代互相之间不存在数据依赖，这样不同的迭代可以并行执行，此时只需要在for语句前加入预处理器指令“#pragma omp parallel for”即可。例如：

|  |
| --- |
| int array[9999];  #pragma omp parallel for  for(int i=0;i<9999;i++)  array[i] = i; |

这样，程序在执行的时候，就会自动根据系统的CPU核心数目创建相应数目的线程，多个线程并行执行。

## gcc编译器的简单教程

在这里，矩阵运算采用的是Eigen库，Eigen 是一个开源的C++的线性代数库，它的特点是：快速，方便，对不同系统、编译器支持良好并且功能强大。

Eigen库是一个header-only库，这种库只有头文件，没有静态链接库或者动态链接库，使用这种类型的库，只需要把下载下来的库中的头文件拷贝出来，在你的程序中#include一下相应的头文件就行。

Eigen中的所有的代码使用的都是#include <文件名>而不是#include “文件名”，所以Eigen库的目录必须位于编译器的目录集中，这就需要配置编译器。

每个编译器都有自己的include目录集，目录集中包含了一系列目录，C++标准库的头文件（比如iostream）就存放在include目录集中的某个目录里面。

C++标准对#include指令的规定如下：

* #include <abc.h> 从编译器的include目录集中的目录中寻找abc.h，若无法找到，则报错。
* #include “abc.h” 从当前目录中寻找abc.h，若找不到，则从编译器的include目录集中的目录中寻找abc.h，若仍然无法找到，则报错。

比如说D:\eigen3存放着Eigen库，你的程序名叫abc.cpp，存放在E:\programs中，想要生成的文件为def.exe，想要使用gcc完成上述任务，需要打开一个命令行窗口，然后键入如下三条命令：

|  |
| --- |
| E:  cd E:\programs  g++ abc.cpp –I “D:\ eigen3” –std=c++11 –static –fopenmp –Ofast –o def.exe |

其中

|  |
| --- |
| E:  cd E:\programs |

用于把命令行的当前工作路径设置为E:\programs

“g++ abc.cpp –I “D:\ eigen3” –std=c++11 –static –fopenmp –Ofast –o def.exe”这条命令中：

* abc.cpp 是源文件名
* –I “D:\ eigen3” 表示将D:\ eigen3添加到include目录集
* –std=c++11 表示使用C++11标准
* –static 表示静态编译，只有这样才能保证编译出来的程序脱离MingW环境后仍然能运行
* -fopenmp表示开启OpenMP的支持
* -Ofast表示将程序的性能优化到最快，这个指令一定要有的，如果没有的话，编译器生成程序的时候，不会对程序做任何优化，这就导致生成的程序运行缓慢。
* –o def.exe 表示生成的目标程序为def.exe

如果程序编译的机器就是运行的机器，或者运行的机器比起编译的机器有着相同型号或者更高级的CPU，还可以增加如下指令来进一步优化程序的性能：

* -march=native

这条指令告诉编译器，充分利用本机（编译程序的机器）的指令集中的指令，以便产生高效程序。如果本机的CPU支持某些指令，恰好编译器又利用了这些指令来优化程序，而在程序运行的机器上却不支持这些指令，那么程序在执行的时候就会非法指令而终止。

# 量子力学

## 矩阵指数的算法

对于对角阵

它的指数：

对于非对角阵A，如果A可对角化，可以将A进行对角化。如果，其中D是对角阵，那么。这样，只要求出P与D，就可以求出。对于一个一般的矩阵，求出P、D并不容易，但是对于厄米阵来说相对容易的多。对于厄米阵，P是酉阵，即。

哈密顿量H是一个厄米阵，非常容易被对角化，于是，求的步骤为：

1. 求出H的所有本征值，以及每个本征值对应的本征矢量
2. 令

## 两个矩阵乘积的迹的算法

如果A与B都是矩阵，则计算需要的时间。然而可以在时间内计算完成：

## 偏迹的算法

设有三个希尔伯特空间A、B、C，它们的维度分别为M,N,P，则直积空间的维度为。中的算子可以用的矩阵表示。这里规定，在使用矩阵表示直积空间中的算子时，总是按照字母顺序，即不论还是（其中分别是A,B空间中的算子），在用矩阵表示做Kronecker积时，总是A在前B在后即与的矩阵表示都是  
设算子，则的计算方法，用矩阵表示，为：

1. 将矩阵分块，每为一块，共分成块
2. 将每个小块看成一个整体，则整个矩阵经过分块以后，可以看成一个的矩阵，每个矩阵元都是一个的小块。对这个的矩阵求迹，求迹时每个的块看成整体参与运算，不对这些块求迹，这样得到的结果是一个的矩阵。

的计算方法，用矩阵表示，为：

1. 将矩阵分块，每为一块，共分成块
2. 将每个小块看成一个整体，则整个矩阵经过分块以后，可以看成一个的矩阵，每个矩阵元都是一个的小块。对这个小块中的每个小块求迹，每个的块经过求迹都变成个一个数，这样得到的结果是一个的矩阵。

根据上面的分析，不难得到三个空间直积的情况下偏迹的求法。设算子，则的计算，可以将看成一个整体。的计算，可以将看成一个整体。的计算方法，用矩阵表示，为：

1. 将矩阵分块，均分成块，每一块称为一个中等块，大小为
2. 将每一块中等块继续分块，均分成个小块，每个小块大小
3. 对这个中等块中的每个中等块求迹，求迹时每个的小块看成整体参与运算，不对这些块求迹，每个中等块经过求迹都变成个一个的小块，这样得到的结果是一个的矩阵。

推广到N个空间的情景：假设有N个空间，他们的维数分别是，使用矩阵表示时，总是按照序号的大小，序号小的在前，序号大的在后。今有算子 ，它的矩阵元为，则是一个  
维的矩阵，其矩阵元（这里规定，所有矩阵元的编号从0而不是1开始）为：  
其中

## Operator类简介

我们进行量子系统模拟的时候，都是使用矩阵运算，当量子系统涉及到多个比特，在书写的某个比特的算子的时候，我们还需要人为地将这个算子与其他比特的单位算子做Kronecker积，非常麻烦。同时，随着比特数目的增加，手工写代码计算偏迹也是一个不小的挑战。显然，这些工作应该是机器来做，而不是人，作为一个物理工作者，我们显然更希望我们脑子里面想的是物理，而不是那些哪个比特在前，那个比特在后，需要与一个多大的单位阵做Kronecker积，或者求偏迹的时候应该把哪些矩阵元相加得到的是哪些矩阵元。这就是编写Operator类的动机。Operator类是我对物理中的算符的抽象，是对矩阵的进一步封装。

比如说，有3个自旋，要计算  
用矩阵代码需要这么写：

|  |
| --- |
| MatrixXcd tmp1,tmp2,H;  kroneckerProduct(Sx,I,tmp1);  kroneckerProduct(tmp1,I,H);  kroneckerProduct(I,Sx,tmp1);  kroneckerProduct(tmp1,I,tmp2);  H+=tmp2;  kroneckerProduct(I,I,tmp1);  kroneckerProduct(tmp1,Sx,tmp2);  H+=tmp2; |

但是有了Operator类，你只需要这样写：

|  |
| --- |
| Operator H = Sx(0)+Sx(1)+Sx(2); |

代码从9行缩减为1行，也变得更加直观，该做的Kronecker积，Operator会自动帮用户做，不需要用户来劳神干这种无聊的事情，这就是面向对象编程展现出来的巨大威力。

每一个Operator对象都有其子空间，子空间是一个int型变量，从0开始编号，比如说有1个电子自旋和7个核自旋，可以令电子自旋的子空间编号为0，核自旋的子空间分别编号是1-7。一个Operator可以处在多个子空间的直积空间内，比如就处在电子自旋子空间与核自旋子空间的直积空间内。

Operator内部是用矩阵存储的，如果一个Operator对象处于八个子空间，编号分别为，这八个子空间每个子空间的维度都是2，则这个对象内部存储一个矩阵的矩阵。使用矩阵存储的一个重要问题是，不同子空间在矩阵中的相对顺序。这里采取的策略是，编号低的子空间在前，编号高的子空间在后。例如算子内部矩阵存储为

Operator类位于头文件quantum.hpp中，所有的定义都在名空间Quantum下，如果要使用Operator类，需要包含如下代码：

|  |
| --- |
| #include <quantum.hpp>  using namespace Quantum; |

并将quantum.hpp所在的目录添加到编译器的include目录集中去。

## 创建一个Operator类的对象

### 单一子空间

创建一个位于单一子空间的Operator对象的使用如下构造函数：

* Operator(int subspace,const MatrixXcd &matrix)

其中，第一个参数指明算符所处的子空间，第二个参数为算符的矩阵表示。

### 直积空间

创建一个位于直积空间的Operator对象的使用如下构造函数：

* Operator(vector**<int>** subspace\_dim,MatrixXcd matrix)

第一个参数给出算符处的子空间信息。其中subspace\_dim[i]表示编号为i的子空间的维度，如果算符不处于此子空间中，就应该将subspace\_dim[i]设置为的值，算符所处的空间将是所有使得的i的按从小到大的顺序直积到一起的空间。

matrix为算符的矩阵表示，表示时，子空间的顺序问题需要与Operator内部表示相同。

### 无空间算符

为了使用方便，引入无空间算符的概念。无空间算符有两种类型：

* 无空间0算符：不属于任何子空间，他与任意算符A相加的结果是A。相比较之下，子空间0中的0算符与子空间1中的A算符的和是而不是A。
* 无空间1算符：不属于任何子空间，他与任意算符A相乘的结果是A。相比较之下，子空间0中的1算符与子空间1中的A算符的和是而不是A。

想要生成无空间0算符，可以直接使用默认构造函数，或者给构造函数传递一个int型的参数0：

|  |
| --- |
| Operator H; //默认构造函数  Operator H(0); //给构造函数传递一个参数0 |

要生成一个无空间1算符，应该给构造函数传递一个int型的参数1：

|  |
| --- |
| Operator rho(1); |

无空间算符通常配合循环一起用：

|  |
| --- |
| //某哈密顿量的初始化过程：  Operator H(0);  for(int i=0;i<n;i++) H += A[i]\*Sz(i);  //某密度矩阵的初始化过程：  Operator rho(1);  for(int i=0;i<n;i++) rho \*= rhos[i]; |

### 零算符与单位算符

使用函数O(int subspace,int dim)创建一个位于子空间subspace中的dim维的零算符。

使用函数I(int subspace,int dim) 创建一个位于子空间subspace中的dim维的单位算符。

### 从矩阵元创建

使用函数Op<int dim>(int subspace,…)创建一个位于subspace子空间中的，n维的算符，其中的“…”用矩阵元代替，矩阵元按行书写，中间用逗号隔开。例如：

|  |
| --- |
| Op<2>(1, 0.0, 0.5, 0.5, 0.0); |

上面的代码生成一个位于子空间1的算符，算符的矩阵表示为：

建议书写的时候不要像上面的例子那样，因为那样书写代码可读性比较差，比较好的书写方式为：

|  |
| --- |
| Op<2>(1, 0.0, 0.5,  0.5, 0.0); |

这种书写方式语义不变，但是可读性好好很多。

## Operator对象的基本性质与操作

### 矩阵

有时候，我们需要将Operator对象中的矩阵提取出来，以便运用一些矩阵的算法进行某些操作。提取算符A的矩阵的方法是：

|  |
| --- |
| A.matrix() |

需要注意的是，上述函数返回值是一个左值，这就意味着，可以对返回值进行修改，修改结果将会直接写入算符A。如果不希望修改的结果被写回，请保留一份拷贝。

特别需要注意的是，如果你对返回值进行修改，而且不小心改变了矩阵的维度，那么可能会产生各种意想不到的bug。

### 输出

可以直接使用标准输出流对算符进行输出，输出的将是算符的矩阵表示，格式与Eigen库中矩阵输出的格式完全相同。例如：

|  |
| --- |
| cout << A << endl; |

上面的代码输出算符A的矩阵表示，并在输出的最末另起一行。

### 访问矩阵元

使用小括号运算符访问矩阵元，小括号运算符接受2个整形参数，返回对应坐标的矩阵元。矩阵元的编号从0开始，返回值是个左值，可以对其进行赋值。例如A是一个算符：

* cout << A(i,j); //向标准输出流输出A的第元素。
* A(i,j) = 5; //将A的第元素赋值为5。

如果要批量对所有矩阵元进行赋值，那么写一大堆“A(i,j) = …”这种代码对矩阵元挨个进行赋值是个非常麻烦而且不易读的方法。正确的做法是使用<<运算符，用法跟Eigen库中使用<<来对矩阵赋值的用法完全一样。示例代码：

|  |
| --- |
| Operator mySx = O(0,2);  mySx << 0.0 , 0.5,  0.5 , 0.0; |

上面代码将mySx设置为

## 空间与维度

### 空间比较

函数same\_space用于判断两个算符是否处在同一空间，如要判断A,B是否处于同一空间，用法为：

|  |
| --- |
| A.same\_space(B) |

或者

|  |
| --- |
| B.same\_space(A) |

### 维度

函数dim返回算符所处的空间的总维度，返回值为int型，要获得A的总维度，用法为：

|  |
| --- |
| A.dim() |

### 空间扩展

使用expand函数对算符的空间进行扩展，这里的扩展指的不是增加空间的维度，而是让算符处于一个更大的积空间中。例如，初始时A = I(0,3)是一个处于空间0的三维算符，我们可以人为的将其扩展到空间去，这样扩展后的算符就处在空间中。从矩阵的角度，扩展操作会得到更大的矩阵，新的矩阵是由原矩阵与单位阵按照合适的顺序做Kronecker积得到。

expand函数有两种用法（假设要对算符A的空间进行扩展）：

* A.expand(int subspace,int dimension)
* A.expand(const Operator &op)

两种用法均不会改变原算符A，结果将被存储在返回值中。第一种用法将A扩展到空间subspace中去，subspace的维度由dimension指定。如果A已经处于subspace中，函数会抛出一个异常。如果A不处于中，则返回值是一个处于 的算符。

## Operator的四则运算

Operator类支持“+-\*/”四则运算。

* 其中“+,-”运算的两个操作数必须是两个Operator对象，返回值类型也是Operator
* “\*”运算的两个操作数可以是一个Operator与一个数，返回值类型是Operator
* /运算的左操作数必须是Operator对象，右操作数必须是一个数，返回值类型是Operator。
* 单目运算符“+,-”的操作数是Operator对象，返回值类型是Operator。
* 单目运算符“\*” 的操作数是Operator对象，返回值类型是Operator，返回值为操作数的厄米共轭（共轭转置）。

每个双目运算符的都有对应的赋值语句：

* A += B等价于A = A + B，其中A、B都是算子
* A -= B等价于A = A - B，其中A、B都是算子
* A \*= B等价于A = A \* B，其中A是算子，B是算子或复数
* A /= B等价于A = A / B，其中A是算子，B是复数

进行运算时，如果运算的两个对象不再一个子空间内，Operator类将自动与相应空间的单位算符做Kronecker积，使用者不需要操心。例如，的结果等效于  
而不是

由于，所以如果A与B不再同一个子空间内，使用运算符\*计算A\*B得到的结果就是，所以没有单独定义运算符的必要。

## 算符求迹

假设算子。

可以使用如下两种方法对A求迹：

* tr(A)
* A.tr()

这两种方法完全等效，结果是一个复数，作为返回值放回。

求偏迹的方法如下：

* B = A.tr(0); //对0空间求偏迹求掉，保留1-9空间
* B = tr(A,0) //与上面等效
* B = A.tr(0,1,2,3,4,5,6,7,8) //对空间0-8求偏迹求掉，保留空间9
* B = tr(A, 0,1,2,3,4,5,6,7,8) //效果同上

不管是使用A.tr(空间列表)这种方法，还是使用tr(A,空间列表)这种方法，算符A都不会改变，计算的结果作为返回值输出，而不是直接写入A。

另外，求偏迹的函数可以接受任意个参数，我们想对几个空间求偏迹，就在空间列表中写几个参数。

不要试图对一个算符的所有空间都求偏迹，你不会如期得到一个复数，相反，你应该使用求迹函数（不传递任何参数），而不是使用偏迹函数（把所有空间都写入参数列表）。

如果要计算两个算子的乘积的迹，先计算两个算子的乘积，然后对乘积求迹不是个聪明的做法。因为这样做不是最高效的。正确的做法是使用tr\_of\_prod函数。使用方法非常简单：

|  |
| --- |
| result = tr\_of\_prod(A,B); |

结果是一个复数，作为返回值返回。

该函数使用前面推导的求乘积的迹的方法来进行求迹操作，所以，如果函数的两个算符位于不同的子空间内，将不会得到正确的结果。tr\_of\_prod在进行运算之前，会检测两个算子是否位于相同的子空间内，如果不是，tr\_of\_prod会抛出一个异常。如果想用tr\_of\_prod求两个位于不同子空间的算符的乘积的迹，请先手动将两个算符扩展到两个算符所在空间的积空间去。

## 演化算子

量子系统模拟的常见步骤是，给定哈密顿量以及初态，求态随着时间的演化。这个时候就需要计算时间演化算子。时间演化算符的计算涉及到矩阵指数的计算（整个过程中始终取）。由前面的叙述，计算矩阵指数的方法是先对角化，然后计算指数。很多情况下我们需要针对多个不同的t计算时间演化算子，这个时候，并不需要每次计算都对H进行一遍对角化，我们只需要对H进行一次对角化，这次对角化的结果可以被用来针对多个不同的t计算演化算子。

这里，人为地将演化算子的计算分成上半部分跟下半部分。上半部分的工作是对角化，而下半部分的工作则是又对角化结果计算指数。比如要计算，其中H是厄米算子（厄米矩阵），上半部分的工作是求出H的所有本征值，以及每个本征值对应的本征矢量，并将矢量存储为。下半部分则是计算矩阵乘法。这样划分的好处是，如果要针对多个不同的t计算指数，上半部分只需要执行一次，下半部分则需要执行多次。

实现上，上半部分的工作由Operator的成员函数U()完成，该函数不接受任何参数。使用方法也非常简单，对算符H调用U方法，并将返回值存储起来，即使用代码：

|  |
| --- |
| auto ret = H.U(); |

这里用到了关键字auto，意思是让编译器自动推导返回值的类型。实际上，该函数返回一个闭包，上半部分的计算结果就存储在这个闭包中。闭包是函数的一种，这里可以简单地理解为，闭包与普通函数的差别在于，普通函数只有运算功能，函数本身没有存储空间，所以不能够存储上半部分得到的结果，而闭包则具有存储空间，可以将上半部分的计算结果存储在闭包中。这个闭包接受一个double类型的参数t，完成下半部分的计算，并将结果作为一个Operator对象返回。该方法假定A是一个厄米算符，如果A不是一个厄米算符，得到的将是一个错误的结果，但是不会有任何错误提示或者输出。闭包是在C++11中才开始得到支持，不同的闭包的类型不同，但是他们的共同点是，闭包的类型程序员不需要关心，只需要使用auto关键字进行自动类型推导就行了。

刚刚的示例代码已经将上半部分的计算结果存储在变量ret中，并且我们知道，ret现在可以当作一个函数使用，ret这个函数是一个计算器，该计算器接收double t为参数，计算并返回，实际计算过程中只需要计算下半部分，因为上半部分的结果已经在之前计算完毕并且存储起来了。

具体用法，可以参考下面例子：

|  |
| --- |
| Operator H = …..; //哈密顿量  Operator rho0 = ….; //初态密度算符  auto U = H.U(); /\*上半部分，只需要计算一次\*/  /\* 现在，可以通过调用U(t)得到t时刻的时间演化算符了 \*/  for(int t=0;t<100;t++) {  /\* 每一次调用U(t)，都是进行一次下半部分的计算 \*/  Operator rho = U(t)\*rho0\*U(-t); //计算末态密度矩阵  cout << rho << endl; //输出末态密度矩阵  } |

# 张量运算

本库提供张量运算的功能，但是要求张量必须是正交坐标系中的张量。张量运算位于头文件tensor.hpp中，并且使用名空间Tensor，如果要使用这些功能，需要在程序中加入代码：

|  |
| --- |
| #include <tensor.hpp>  using namespace Tensor; |

并将tensor.hpp所在的目录添加到编译器的include目录集中去。

一般的张量运算库通常只支持同种数据类型之间的相互运算。比如说两个数据类型都是double类型的张量可以相互进行运算（加，减，数乘，张量积，并矢，缩并等），但是double类型的张量跟Operator类型的张量则不能相互之间进行运算，这就给我们的应用带来了极大的不方便。在实际应用的过程中，经常使用到不同类型的张量之间的运算，比如：自旋1与自旋2之间的dipole-dipole相互作用哈密顿量可以写成，其中与是自旋矢量算子，在编程的时候通常被实现为数据类型为Quantum::Operator类型的矢量。而则是两个自旋之间的耦合张量，通常被实现为数据类型为double类型的张量。数据类型为Quantum::Operator类型的矢量与数据类型为double类型的张量并不是相同的数据类型。为了实现这一点，本库中大量运用了C++11的新特性decltype进行模板元编程来实现。

## 类型

正交坐标系下的张量运算的核心是模板类template <typename scalar,int order,int dimension> tensor。其中order是张量的阶，默认为2，对于矢量order=1，对于二阶张量order=2。dimension是张量的维，默认为3。scalar是张量的域的类型（即张量所对应的标量的类型）。

例如：

* tensor<double>表示实二阶张量，数据用double存储。
* tensor<std::complex<double>,1>则表示复二阶矢量。
* tensor<Quantum::Operator,1>表示矢量算子。

该类在设计的时候，尽可能地使其行为与数组保持一直。

根据习惯，定义了一个常用的模板typedef：

template <typename scalar,int dimension> vector等价于tensor<scalar,1,dimension>，同样，模板中的dimension默认为3。注意的是，C++的标准模板库（STL中也有vector模板类，该类与这里的vector重名，所以写代码的时候可能需要明确指定是哪个vector，不然编译器会报错。使用Tensor::vector使用本库中的vector模板类，使用std::vector使用标准库中的模板类）例如：

* vector<Quantum::Operator>矢量算子。
* vector<double>实矢量，使用double存储数据。

每个tensor类（包括经过typedef得到的模板类）都有一个与之对应的数组类型，对于标量类型为scalar的M阶N维张量，这个类型是一个每一维大小都为N的M为数组。使用tensor:: array\_type来访问这个数组类型。例如：

* vector<double>::array\_type为double[3]
* tensor<Quantum::Operator>::array\_type为Quantum::Operator[3][3]

## 构造函数

tensor包含两个构造函数：不接受任何参数的默认构造函数，以及使用C++的统一初始化特性初始化的构造函数。默认构造函数的行为与C++中数组声明（但不初始化）的行为一致。统一初始化的行为则与C++中数组元素的统一初始化行为一致。例如：

* vector<double> a; //默认构造函数，创建三维矢量a，其值不确定
* vector<double> b = { 1.0, 2.0, 3.0 }; //统一初始化，创建三维矢量b，它的三个分量分别是1.0,2.0,3.0。

## 运算符

圆括号()运算符：括号运算符用于访问张量的分量。如果对象不是const类型，则圆括号的返回值是个左值（lvalue）。如果对象是const类型，则圆括号的返回值是个右值（rvalue）。对于N阶张量，该运算符接受N个参数，访问张量的第分量。tensor类的数据内部存储为一个变量名为components的数组（private变量，外部无法访问到），数组类型为该tensor类的array\_type。假设有tensor<T,M,N>类的变量a，则a访问的是。

方括号[]运算符：也可以使用方括号来访问张量的分量。按照数据在机器中的存储顺序来访问。由于张量内部数据存储采用的是多元数组，因此数据在机器中的存储顺序与多元数组相同。对于N阶M维的张量a，与访问的是同一个元素。与圆括号相同的一点是，如果对象不是const类型，则方括号的返回值是个左值（lvalue）。如果对象是const类型，则方括号的返回值是个右值（rvalue）。

例如，变量a,b的声明分别是tensor<double> a; vector<double> b;

* a(0,2) = 2.5;
* a[2] = 2.5; //效果同上
* b(1) = 4.3;
* b[1] = 4.3; //效果同上
* cout << a(2,1);
* cout << a[7]; //效果同上
* cout << b[0] << b[1] << b[2];
* cout << b(0) << b(1) << b(2); //效果同上

加减运算符：实现张量的加减运算。加减运算要求两个张量阶数维数都相同，但是张量的标量类型可以相同或者不同。假设现有tensor<T1,M,N>对象a、tensor<T2,M,N>对象b，T1与T2之间定义有加法运算，则a+b得到的是tensor<decltype(T1()+T2()),M,N>类的对象，记为c。它的元素。a-b得到的是tensor<decltype(T1()-T2()),M,N>类的对象，记为d。它的元素。

例如，两个变量vector<int\*> a; vector<int> b;，则：

* a+b，b+a，a-b均是vector<int\*>类型的
* b-a编译无法通过

一元加减运算符：假设现有tensor<T,M,N>对象a，T与定义有单目加法运算，则+a得到的是tensor<decltype(+T()),M,N>类的对象，记为b。它的元素。-a得到的是tensor<decltype(-T()),M,N>类的对象，记为c。它的元素。

乘除运算符：乘除运算符实现张量的数乘操作，规则同张量的定义。注意的是，乘法运算符并不能计算张量的张量积，张量的点乘或者是叉乘。有意义的运算为scalar\*tensor, tensor\*scalar, tensor/scalar，而scalar/tensor，tensor/tensor以及tensor\*tensor则是无效的。假设有tensor<T1,M,N>类型的变量a，T2类型的变量b（T2不能是任意张量类型），则a\*b类型为tensor<decltype(T1()\*T2()),M,N>，b\*a的类型为tensor<decltype(T2()\*T1()),M,N>，a/b的类型为tensor<decltype(T1()/T2()),M,N>，b/a、a/a以及a\*a无法通过编译。例如，现有double w1; vector<Quantum::Operator> S则：

* w1\*S，S\*w1，S/w1类型都是vector<Quantum::Operator>。
* S\*S，S/S，w1/S无法通过编译。

## 方法

张量积：使用prod方法计算张量的张量积。这是一个不定参数个数的函数，可以接收任意多个参数，这些参数的类型必须是tensor模板类的某个实例。prod按照参数从左到右的顺序对参数中的张量进行张量积运算，并将结果作为返回值返回。要计算张量积，则使用。

例如，有3个张量vector<Quantum::Operator> S1; vector<Quantum::Operator> S2; Tensor<double> D12;，则：

* prod(S1,D12,S2); //返回张量积，结果是一个tensor< Quantum::Operator,4>。
* prod(S1, S2); //返回张量积，结果是一个tensor< Quantum::Operator >。
* prod(S1,S1,S2,S2); //返回张量积，结果是一个tensor< Quantum::Operator,4>。
* prod(S1); //编译失败

缩并：使用contract方法对张量进行缩并操作，使用方法为contract<i,j>(t)，其中i与j是编译时常量，t为一个超过二阶的tensor类型，该操作实现对张量t的第i,j个维度的缩并。维度的编号从0开始。例如有张量tensor<double,3> a;则：

* contract<0,1>(a); //对a的头两个维度进行缩并，结果为vector<double>。

点乘：使用dot计算张量的点乘积，这是一个不定参数个数的函数，可以接收任意多个参数，这些参数的类型必须是tensor模板类的某个实例。dot按照参数从左到右的顺序对参数中的张量进行点乘积运算，并将结果作为返回值返回。要计算点乘积，则使用。

例如，有3个张量vector<Quantum::Operator> S1; vector<Quantum::Operator> S2; Tensor<double> D12;，则：

* dot(S1,D12,S2); //返回点乘积，结果是一个tensor< Quantum::Operator,0>。
* dot(S1, S2); //返回点乘积，结果是一个tensor< Quantum::Operator,0>。
* dot(S1,D12); //返回点乘积，结果是一个vector< Quantum::Operator>。
* dot(S1); //编译失败

并联双点乘：使用ddotp计算并联双点乘，该函数接受两个参数a与b，返回结果为。如果a的类型为tensor<T1,M,N>，b的类型为tensor<T2,P,N>则返回值的类型为tensor<decltype(T1()\*T2()),M+P-4,N>。

串联双点乘：使用ddots计算串联双点乘，该函数接受两个参数a与b，返回结果为。如果a的类型为tensor<T1,M,N>，b的类型为tensor<T2,P,N>则返回值的类型为tensor<decltype(T1()\*T2()),M+P-4,N>。

叉乘：使用cross计算三维一阶张量（三维矢量）双点乘，该函数接受两个参数a与b，这两个可以具有不同的，返回结果为。如果a的类型为vector<T1 >，b的类型为vector<T2>则返回值的类型为vector<decltype(T1()\*T2())>。

矢量的模：使用abs计算矢量的模，该函数接受一个参数a，返回值为。

生成单位张量：使用模板函数I<T,dimension>()来生成二阶单位张量，其中T是标量类型，dimension是张量的维度。dimension可以不指定，这种情况下dimension为默认值3。

## 示例程序

#include <tensor.hpp>

#include <iostream>

using namespace Tensor;

int main() {

//定义并初始化向量，以及二阶张量

vector<int> a = { 1,2,3 };

vector<int> b = { 4,5,6 };

tensor<int> A = { 1,2,3,

4,5,6,

7,8,9 };

//计算点乘积

std::cout << dot(a,A,b) << std::endl;

//计算点乘积

std::cout << dot(a,b) << std::endl;

//计算并矢

auto abab = prod(a,b,a,b);

//输出并矢的第二个分量

std::cout << abab(0,0,0,1) << std::endl;

//输出并矢的第一个分量

std::cout << abab[0] << std::endl;

//计算矢量积

vector<int> c = cross(a,b);

//输出并矢的第三个分量

std::cout << c(2) << std::endl;

//输出的模

std::cout << abs(c) << std::endl;

}

# 自旋体系

根据实验室的需要，我还专门对自旋体系定义了一些有用的字面量、自旋算子等等，同时还对一些常用的数学转换定义了字面量。这些定义位于头文件spin.hpp中，并且使用名空间Spin，如果要使用这些功能，需要在程序中加入代码：

|  |
| --- |
| #include <spin.hpp>  using namespace Spin; |

并将spin.hpp所在的目录添加到编译器的include目录集中去。

## 数学

最常用的是常数，在这里被定义为常量constexpr double pi。

还有就是角度弧度的转换，使用字面量“\_deg”自动将角度转换为弧度。

例如

|  |
| --- |
| cout << 180\_deg/pi; //输出1 |

另一个常见的则是虚数的输入，这里引入\_i字面量。写复数更方便自然，不再需要写complex<double>(0,1.5)，现在只需要写1.5\_i就行。例如：

|  |
| --- |
| //  complex<double> a = 2\_i;  complex<double> b = -1\_i;  complex<double> c = 3.5\_i;  complex<double> d = 5.3+2.1\_i; |

## 时间、频率、磁场

为各种频率引入字面量：\_Hz,\_KHz,\_kHz,\_MHz,\_GHz,\_THz。这些字面量自动将频率转换成Hz。例如：

|  |
| --- |
| double f1 = 1.5\_MHz; //  double f2 = 3\_Hz; //f2 = 3  double f3 = 9.7\_GHz; // |

为各种时间引入字面量：\_ns,\_us,\_ms。这些字面量自动将时间转换成秒。例如：

|  |
| --- |
| double t1 = 2.5\_ns; //  double t2 = 3.4\_us; //  double n = t2/1\_ns; //t2用纳秒为单位是多少纳秒？ |

为各种磁场引入字面量：\_G,\_T。这些字面量自动将各种磁场单位转换成特斯拉。例如：

|  |
| --- |
| double B1 = 3500\_G; //B1 = 0.35  double B2 = 0.4\_T; //B2 = 0.4  double gamma = 4.5\_MHz/1\_G; //转换成Hz/T |

为各种距离引入字面量：\_nm,\_um,\_mm。这些字面量自动将距离转换成米。例如：

|  |
| --- |
| double r1 = 2.5\_nm; //  double r2 = 3.4\_um; //  double n = r2/1\_nm; //r2用纳米为单位是多少纳米？ |

## 算子

Quantum名空间中的函数O(int subspace,int dim)创建一个位于子空间subspace中的dim维的零算符。函数I(int subspace,int dim) 创建一个位于子空间subspace中的dim维的单位算符。在Spin名空间中同样也定义了I函数与O函数，他们的原型为O(int subspace)与I(int subspace)，用于生成二维的单位算子与零算子，算子位于空间subspace中。这样，如果要生成的零算子或者单位算子的维度是2，则可以省略维度这个参数。例如：

|  |
| --- |
| I(1,3) //空间1中的3维单位算子  I(0) //空间0中的二维单位算子  O(2) //空间2中的二维零算子 |

定义了三个函数：

* Sx(int subspace,int dim=2)
* Sy(int subspace,int dim=2)
* Sz(int subspace,int dim=2)

用于生成subspace空间中的三个方向的自旋算子。第二个参数表示算符的维度，对于自旋的粒子，维度是2；对于自旋是的粒子，维度是。 该参数是个可选参数，默认值是2，对应于自旋。

同时还定义了函数S，其原型为

|  |
| --- |
| Tensor::vector<Quantum::Operator,3> S(int subspace,int dim=2) |

该函数生成subspace空间中的矢量算符，第二个参数表示算符的维度，是个可选参数，默认值是2，对应于自旋。

# GRAPE算法

GRAPE由Khaneja于2005年最先提出了使用梯度下降的算法来实现最优控制的脉冲序列控制[[2]](#endnote-2)。Khaneja考虑形式为

Equation

|  |
| --- |
|  |

的哈密顿量，将按照时间切成片，编号为，从开始编号，直到，从开始编号，直到。使用来衡量当前合成的操作与要合成的目标操作之间的“重叠程度”。通过迭代  
来对进行优化。其中，  
其中，  
其中  
为每个时间片的传播子。

## 计算的一点小技巧

实际实现时，注意到  
与无关，可以提到循环外面进行计算，以及  
其中，可以看成一个整体，与无关，可以提到循环外面进行计算。

## 变量代换

另一个问题是，我们系统的哈密顿量并不总是Equation 1这种形式，而有可能是  
这里，我们需要优化的是，而不是。此时需要的迭代是  
由导数的链式法则  
上式中与*j*有关，就是文献中给出的结果，而与无关，仅由的形式决定。上式表示成矩阵，为：  
其中  
是关于的雅可比矩阵。

## 变量值的限制

有时候我们对我们优化的变量有要求，比如说变量不能变化太快，再比如说变量不能要过某个固定值。这种要求通过两个地方来实现：第一个是惩罚项，通过人为地在中增加一些新的项，从而改变迭代方程，来使得迭代过程中变量更倾向于满足某些条件。第二个是约束，优化过程中，在自变量不满足条件时，强制按照某些规则修改自变量的值，来使得自变量满足条件。

## 噪声

由于噪声的存在，通常会使得并不是一个固定的值，而是按照某个分布存在，要处理这种情况，需要对算法进行一定的修改。这里的做法是，通过按照的分布对进行抽样，每组样本单独计算其保真度，以及，最终的保真度以及都是这些样本的平均。

## 病态迭代的识别

这是我自己引入的一个实验性的特性。GRAPE并不能保证一定收敛到最大值，而是只能保证收敛到局部极大值。在旧有的算法中，为了防止迭代陷入局部极大值无法自拔，采用的策略是限制最大迭代次数，如果在最大迭代次数之内保真度仍然无法达到阈值，则认为算法陷入了局部极大值，此时应该终止算法，给定新的初值，并且重新进行迭代。有的算法还会检测梯度的大小，如果梯度已经足够小（已经十分接近极大值），但是保真度仍然无法达到阈值，此时也认为算法陷入了局部极大值，终止算法冰给定新的初值重新进行迭代。并且实际使用中，发现这种做法存在很严重的性能浪费：在初值选择合适的前提下，算法很快收敛达到阈值，而如果初值选择不合适，算法总是执行最大迭代次数，最大迭代次数选择面临两难的困境：过大则初值选择失败的代价太大，过小则会导致本来能够正确收敛到最大值的迭代在还没来得及到达最大值的时候就已经被误杀了。因此，提出新的方法来判断收敛到局部极小值非常重要。

理论上可以得到，在足够接近极大值（或者说迭代次数足够大）的时候，迭代次数与成线性关系，其中是本次迭代收敛到的极大值，是迭代进行到第次的时候达到的保真度。如果我们想要达到的阈值为，则考察函数随着迭代次数的变化，如果迭代的最终收敛值恰好为，则在在足够接近极大值（或者说迭代次数足够大）的时候该函数与成线性关系，而如果迭代最终的收敛值比大，则足够接近时，函数会超越线性地增长，如果迭代最终收敛值比小，在函数会缓慢增长最后收敛到，我们的目的，就是找到算法识别出这种情况，以便及时终止尝试避免不必要的浪费。

下面是实际测试得到的几个的图像：

观察发现，收敛到<threshold的局部极大值的情况下，有一个明显的转折点，在转折点之前增长非常迅速，转折点之后增长非常缓慢。做出的差分，即在第次迭代与第次迭代的差，上面四张图对应的差分图分别如下：

可以发现，第一种情况下差分会收敛到0，其他情况都不会，于是想到引入一个参数：病态迭代判定阈值。当本次迭代对应的差商的绝对值小于时，程序终止。

引入这个终止条件以后进行实测发现取中的任何数都没发现问题，如果担心误杀就把取小点，如果想要提高效率就把取大点，如果想要禁用该特性，则将die\_diff\_ratio取成0。如果取成1的话，基本上在拐点处程序就会被终止，而如果取成的话，则基本上拐点区域刚过，程序刚开始缓慢收敛的时候被终止，如果取的更小则可以看到更长的收敛曲线。但是上面实际测试中，均未发现误杀现象。

## 实际实现

### 算法描述

系统哈密顿量满足如下形式：  
其中满足一定分布，固定不变，控制向量是关于时间的分段常数函数，每个时间片的长度相等，共有个时间片，仅为的函数。在一定约束条件下，优化控制向量，使得在这段时间完成给定操作。

### 输入

* 目标操作
* 系统描述system，其中包括：根据满足的分布抽取的个样本、各个的值、惩罚计算函数penalty、对控制向量的约束constrain、的计算函数transform、雅可比矩阵的计算函数Jacobian。
* 总时间T
* 每次行进步长step
* 目标保真度threshold
* 病态迭代判定阈值die\_diff\_ratio
* 迭代次数上限max\_times
* 控制向量在不同时间片的初始值

### 输出

* 结果保真度
* 优化过的控制向量在不同时间片的值

### 算法流程

记控制向量

雅可比矩阵  
梯度

算法流程表述如下：

1. 针对每个时间片，调用transform，由计算。
2. 针对每个时间片，调用Jacobian，由计算。
3. 调用penalty，由所有时间片的计算所有时间片的惩罚项。
4. 针对每个样本的，计算系统哈密顿量。然后从每个样本的计算出每个样本每个时间片的传播子。
5. 对于每个样本，从小到大依次计算每个时间片的。
6. 对于每个样本，其实际传播子。
7. 对于每个样本，计算内积。
8. 对于每个样本，从大到小依次计算每个时间片的。
9. 对于每个样本每个时间片，计算。
10. 对于每个样本每个时间片以及的每个分量，计算
11. 对于每个样本每个时间片，计算
12. 对于每个时间片，对样本进行平均
13. 对于每个时间片，计算新的控制向量
14. 调用constrain，对所有时间片的新的控制向量进行处理，得到可接受的所有时间片的控制向量。
15. 对于每个时间片，将写入控制向量，替换原有值。
16. 对于每个样本，计算旧的对应的保真度  
    其中，为系统所在的希尔伯特空间维度。
17. 对样本进行平均，得到平均保真度
18. 计算
19. 如果并且并且当前迭代次数没超过max\_times，从步骤1开始执行新一轮迭代。

### 惩罚项

惩罚计算函数penalty的输入为所有时间片的，输出为所有时间片的惩罚项。需要注意的是，一次penalty调用一次性接受所有时间片的，而不是某个时间片的，penalty会一次性计算出所有时间片的惩罚项，而不出某个时间片。因此，每一轮迭代，penalty只会被调用一次，而不是次。对于算法来说，penalty只是一个回调函数，算法本身并不实现penalty，对penalty没有任何要求，但是要达到良好的效果，需要由用户根据算法对的使用方式来决定如何实现penalty。

算法对的使用方式为：

Equation

|  |
| --- |
|  |

也就是说，是直接被加到梯度上的。

如果对没有任何要求，那么直接设置即可。如果对没有一定的要求，通常的做法是，修改优化问题本身。如果不加任何惩罚项，则此处GRAPE算法优化的目的是使得达到最大值。比如说，我们如果要求对于每个时间片，不能太大，同时不同时间片之间的所有分量不能变化太快。这种要求之下，可以把优化问题的目的修改为使得  
达到最大值，其中是可调的参数，具体取值试具体问题而定。要让达到最大值，由Equation 2，我们应该使得  
于是  
即

### 限制

惩罚计算函数penalty的输入为所有时间片的，输出为所有时间片的惩罚项。需要注意的是，一次penalty调用一次性接受所有时间片的，而不是某个时间片的，penalty会一次性计算出所有时间片的惩罚项，而不出某个时间片。因此，每一轮迭代，penalty只会被调用一次，而不是次。对于算法来说，penalty只是一个回调函数，算法本身并不实现penalty，对penalty没有任何要求，但是要达到良好的效果，需要由用户根据算法对的使用方式来决定如何实现penalty。

## 编程接口

程序方面，

1. C++高级编程, Marc Gregoire, Nicholas A. Solter, Scott J. Kleper著，侯普秀 郑思遥译 [↑](#endnote-ref-1)
2. Khaneja, N., Reiss, T., Kehlet, C., Schulte-Herbruggen, T. & Glaser, S. J. Optimal control of coupled spin dynamics: design of NMR pulse sequences by gradient ascent algorithms. J. Magn. Reson.172, 296–305 (2005). [↑](#endnote-ref-2)