QuantaPlus library Documentation

Mohammed Maher Abdelrahim Mohammed

July 28, 2023

Contents

Chapter 1

Introduction to QuantaPlus

1.1 Overview of QuantaPlus Library

The development of QuantaPlus stemmed from the need for a powerful and user-friendly library dedicated to solving and exploring problems in quantum mechanics. Quantum mechanics is a fundamental theory that describes the behavior of particles at the atomic and subatomic levels, and it has numerous applications in physics, chemistry, material science, and other scientific disciplines. The motivation behind QuantaPlus was to provide researchers, students, and developers with a comprehensive toolset to facilitate their work in quantum mechanics. The developers recognized the complexity and mathematical intricacies involved in quantum mechanics calculations and simulations, and they aimed to simplify these processes. The goal of QuantaPlus is to empower users to tackle quantum mechanics problems more efficiently and effectively. By providing a library that offers a range of classes, functions, and utilities specifically designed for quantum mechanics calculations, QuantaPlus eliminates the need for users to develop their own custom solutions from scratch. This saves time and effort, allowing users to focus on the core aspects of their research or projects. Additionally, QuantaPlus aims to bridge the gap between theoretical concepts and practical implementation. It offers a user-friendly interface that abstracts away complex mathematical operations and provides intuitive functions for common quantum mechanics tasks. This approach allows users to focus on understanding and exploring quantum phenomena rather than getting bogged down in the technical details of the implementation. The development team behind QuantaPlus also envisioned a collaborative and open-source environment. By making QuantaPlus freely available and encouraging contributions from the community, they aimed to foster knowledge sharing and innovation in the field of quantum mechanics. This approach allows researchers and developers to build upon each other's work and collectively

advance the understanding and application of quantum mechanics.

Overall, the background and motivation behind the development of QuantaPlus can be summarized as a desire to provide a comprehensive, user-friendly, and collaborative toolset for solving quantum mechanics problems. By simplifying the implementation of quantum calculations and simulations, QuantaPlus empowers users to delve deeper into the fascinating world of quantum mechanics and unlock new insights and discoveries.

Overview of its goals and objectives in solving quantum mechanics problems

The goals and objectives of QuantaPlus in solving quantum mechanics problems can be summarized as follows:

1. Comprehensive Problem Solving: QuantaPlus aims to provide a comprehensive solution for tackling a wide range of quantum mechanics problems. Whether it involves calculating eigenvalues and eigenvectors, simulating quantum systems, manipulating quantum states, or performing various quantum operations, QuantaPlus offers a rich set of functionalities to address these challenges.

2. Efficiency and Accuracy: QuantaPlus is designed to deliver efficient and accurate results in quantum mechanics calculations. The library incorporates optimized algorithms and numerical methods to ensure high-performance computations while maintaining precision and accuracy. By leveraging the power of C++ and template programming, QuantaPlus minimizes computational overhead and enhances the efficiency of calculations.

3. User-Friendly Interface: The developers of QuantaPlus recognize the importance of providing a user-friendly interface that simplifies the usage of the library. The design principles of QuantaPlus prioritize intuitive and easy-to-use functions and classes, allowing users to focus on the underlying quantum mechanics concepts rather than getting caught up in technical complexities. The interface promotes seamless integration of QuantaPlus into existing projects, enabling users to quickly start solving quantum mechanics problems.

4. Flexibility and Customization: QuantaPlus emphasizes flexibility and customization to cater to diverse research needs and scenarios. The library offers a modular structure, allowing users to selectively include the required modules based on their specific requirements. Moreover, QuantaPlus utilizes C++ templates, enabling users to work with various data types and adapt the library to their specific use cases.

5. Documentation and Resources: QuantaPlus recognizes the importance of comprehensive documentation and resources to support users in effectively utilizing the library. The documentation provides detailed explanations of classes, functions, and usage examples, helping users understand the capabilities and proper usage of QuantaPlus. Additionally, the availability of online resources, forums, and a supportive community fosters knowledge sharing and collaboration among users.

6. Collaborative and Open Source: QuantaPlus embraces a collaborative and open-source development model. The library is freely available, allowing researchers, students, and developers to access and utilize it without barriers. The development team encourages community contributions, bug reporting, and feature suggestions, fostering an environment of collective learning and improvement.

By aligning its goals and objectives with these principles, QuantaPlus aims to be a reliable and versatile tool for solving quantum mechanics problems. It strives to empower users with efficient and accurate calculations, a user-friendly interface, customization options, extensive documentation, and a collaborative community, ultimately advancing research and exploration in the field of quantum mechanics.

1.2 Features and Capabilities of QuantaPlus

QuantaPlus offers a range of features and capabilities that make it a powerful tool for solving quantum mechanics problems. Here are some key features and capabilities of QuantaPlus:

1. Quantum State Manipulation: QuantaPlus provides classes and functions for creating, initializing, and manipulating quantum states, represented as ket and bra vectors. Users can perform operations such as addition, subtraction, scalar multiplication, inner product, outer product, and tensor product on quantum states.

2. Quantum Operators: QuantaPlus includes a comprehensive set of quantum operators commonly used in quantum mechanics, such as identity, Pauli matrices, and angular momentum operators. Users can perform operations on operators, calculate eigenvalues and eigenvectors, and apply operators to quantum states.

3. Harmonic Oscillators: QuantaPlus offers functionality for working with harmonic oscillators, including the creation and annihilation operators. Users can calculate energy eigenvalues and eigenstates of harmonic oscillators, enabling simulations and analysis of harmonic oscillator systems.

4. Clebsch-Gordan Coefficients: QuantaPlus provides functions for calculating Clebsch-Gordan coefficients, which are essential for understanding the coupling of angular momenta in quantum mechanics. Users can compute and utilize these coefficients in calculations involving coupled systems.

5. Mathematical Utilities: QuantaPlus includes a collection of mathematical utilities for quantum mechanics calculations. These utilities cover a wide range of functions, including statistical measures, complex number manipulation, factorials, binomial coefficients, and more. Users can leverage these utilities to simplify their calculations and streamline their workflow.

6. LaTeX Code Generation: QuantaPlus offers utilities for generating LaTeX code for mathematical expressions commonly used in quantum mechanics. This feature allows users to create LaTeX representations of matrices, vectors, symbols, and equations, aiding in the creation of professional-looking reports, presentations, and publications.

7. Performance Optimization: QuantaPlus incorporates efficient algorithms and numerical methods to ensure high-performance calculations. It leverages the power of C++ and template programming to minimize computational overhead and enhance the efficiency of operations. Users can benefit from optimized calculations and obtain results quickly and accurately.

8. Flexibility and Customization: QuantaPlus is designed to be flexible and customizable to suit diverse research needs. It provides a modular structure, allowing users to selectively include the required modules based on their specific requirements. QuantaPlus also supports various data types, enabling users to work with different numerical representations and adapt the library to their specific use cases.

9. Comprehensive Documentation and Resources: QuantaPlus is accompanied by comprehensive documentation that explains the functionality, usage, and examples of its classes and functions. Users can refer to the documentation to understand the capabilities of QuantaPlus and effectively utilize its features. Additionally, online resources, forums, and a supportive community contribute to the availability of learning materials and assistance.

These features and capabilities make QuantaPlus a versatile and powerful library for solving quantum mechanics problems. By providing a wide range of functionalities, QuantaPlus enables researchers, students, and developers to perform quantum calculations, simulations, and analysis efficiently and accurately.

1.3 Installation and Setup

The package can be downloaded from github repository.

Files

The gzipped tarred file (QuantaPlus.vvvv.tar.gz) will produce a directory QuantaPlus.vvvv with a number of subdirectories. vvvv is version information. The created directory is called the main directory in the remainder. The main directory contains the files LICENSE, README.md, and a Makefile. The subdirectory doc contains the documentation. The subdirectory lib will after compiling contain the compiled libraries libquantaplus.a. The subdirectory include contains all the needed header files. examples contains the testing and example programs. outputs contains the output obtained after execute some of the example files. There are a few extra files around as well. These typically contain inputs needed or large sets of constants.

Installation

The main step is to run makefile in the main directory. To run the makefile, first make sure that you have the make command installed on your system. On Ubuntu, you can install make by running the command "sudo apt-get install make". Once make is installed, navigate to the directory where the makefile is located using the terminal. Then, simply run the command "make" to execute the instructions in the makefile.

For example, in the above makefile, the first command checks if the Eigen3 library is installed on the system. If it is not found, it will be installed using the command "sudo apt-get install libeigen3-dev". The next command copies the QuantaPlus folder to the /usr/include/ directory. The makefile also contains instructions to build and run example programs, generate output, and run tests. You can build and run an example program by running "make" followed by the name of the example. For example, "make example1" will build and run the example1 program. Similarly, you can generate output by running "make" followed by the name of the output file, and run tests by running "make" followed by the name of the test. To clean the output files you can use "make clean" command. It is important to note that the above makefile is just an example, and the specific instructions may vary depending on the actual makefile you are using. Also, depending on the system you are using, you may need to run the make command with administrative privileges, i.e. use "sudo make" instead of just "make".

testroutines and tutorials

The provided makefile includes several routines for installing and testing the QUANTAPLUS package.

To install the package, the user can simply run the command 'make install' in the terminal while in the same directory as the makefile. This will check if the Eigen3 library is installed and if not, it will install it. Then it will copy the QuantaPlus folder to the /usr/include/ directory.

To run the test routines, the user can run the command 'make test name' where 'test name' is the name of the test file they want to run (e.g. 'make test angular momentum'). The makefile has several test files included, such as test angular-momentum, test braket, test latex test eigenvalue, test operators, and test clebschGordon. These test files are located in the tests/ directory.

The makefile also includes several tutorial files which can be run by running the command 'make example name' where 'example name' is the name of the example file you want to run (e.g. 'make example1'). These example files are located in the examples/ directory.

The makefile also includes a target to generate latex files 'make genLatex' , and also includes a target for the output directory 'make OUTPUT' but the user should check the dependencies and requirements before running those targets.

Additionally, the user can run the command 'make clean' to remove any compiled files.

In summary, the makefile provided allows the user to easily install the QUANTAPLUS package, run test routines, and execute tutorial files by simply running commands in the terminal. The user should ensure that they are in the same directory as the makefile before running the commands.

Getting started

The QUANTAPLUS library is a collection of C++ classes and functions that can be used to solve quantum mechanics problems. The library includes classes for operators, bra-ket notation, angular momentum, Clebsch-Gordon coefficients, and more.

To use the library, first make sure that you have Eigen3 installed on your system. If Eigen3 is not already installed, the makefile included in the QUANTAPLUS package will install it for you.

Next, use the makefile to install the QUANTAPLUS library. To do this, open a terminal window and navigate to the directory where the QUANTAPLUS package is located. Then, type "make install" and press enter. This will copy the QUANTAPLUS folder to the /usr/include/ directory.

Once the library is installed, you can use it in your C++ programs by including the library in your C++ projects by adding the following primary include header file to your source code:

```
#include <quantaplus.h>
```

The QUANTAPLUS package also includes several tutorial files and test routines that demonstrate how to use the library to solve different types of problems. To run the tutorials, type "make" followed by the name of the tutorial you want to run. For example, to run the first tutorial, type "make example1" and press enter.

Similarly, to run the test routines, type "make" followed by the name of the test you want to run. For example, to run the test for the angular momentum class, type "make testangular momentum" and press enter.

You can also customize the makefile to run the examples and test routines according to your own needs, as well as to generate output files and plots.

It's also worth mentioning that the makefile also has a 'clean' target that allows you to remove all the compiled files and outputs generated by the makefile.

To run the makefile, you need to have a C++ compiler (such as GCC) and make installed on your system. If you're using a Ubuntu based Linux distribution, both should be already installed, otherwise, you might need to install them manually.

In summary, the QUANTAPLUS library is a powerful tool for solving quantum mechanics problems, and the included makefile makes it easy to install and use. The tutorial files and test routines demonstrate how to use the library to solve different types of problems, and the makefile can be customized to run examples and tests according to your own needs

Chapter 2

Quantum State vectors

2.1 Bra Class

2.1.1 Description

The `Bra` class is a C++ implementation of a row vector representing a "bra" vector in quantum mechanics. A "bra" vector is a complex-valued row vector that represents an element of the dual space (also known as the "bra space") of a Hilbert space. In quantum mechanics, "bra" vectors are used to represent quantum states in the context of the bra-ket notation.

2.1.2 Template Parameters

- `T`: The data type used for the elements of the "bra" vector. It is typically used with `std::complex<double>` to handle complex numbers used in quantum mechanics.

2.1.3 Constructors

- `Bra()`: Default constructor that creates an empty "bra" vector.
- `Bra(int col)`: Constructor with an integer parameter `col`, which creates an empty "bra" vector with a fixed number of columns specified by `col`.

2.1.4 Member Functions

- `Print()`: Prints the elements of the "bra" vector in fractional symbolic form. For example, for a complex number $z = \frac{A}{B} + i\frac{C}{D}$, where i is the imaginary unit, the function will print the elements of the "bra" vector as $\frac{A}{B} + i\frac{C}{D}$.

- `NPrint()`: Prints the elements of the "bra" vector in fractional numeric form. For example, for a complex number $z = \frac{A}{B} + i\frac{C}{D}$, the function will print the elements of the "bra" vector as $\frac{A}{B}, \frac{C}{D}$ (numerical fractions).
- `NormFactor(const Bra<T> &bra)`: Calculates the norm factor (normalization constant) of the input "bra" vector. The norm factor is computed as the square root of the inner product of the "bra" vector with its corresponding ket vector (complex conjugate). The function returns the norm factor as a complex number.
- `Normalize(const Bra<T> &bra)`: Normalizes the input "bra" vector by dividing each element of the vector by its norm factor. The function returns the normalized "bra" vector.

2.1.5 Physics Explanation

Quantum States and Hilbert Space

In quantum mechanics, a quantum system is represented by a vector space called a Hilbert space. The Hilbert space is a complex vector space that describes the state of the quantum system. Quantum states are represented by vectors in this Hilbert space. A "ket" vector ($|ket\rangle$) is used to represent a quantum state, and a "bra" vector ($\langle bra|$) represents an element of the dual space (adjoint space) of the Hilbert space.

Inner Product and Norm

The inner product of two vectors in the Hilbert space is a fundamental operation in quantum mechanics. For a "bra" vector $\langle\psi|$ and a corresponding "ket" vector $|\phi\rangle$, the inner product $\langle\psi|\phi\rangle$ calculates the amplitude of transitioning from the state $|\phi\rangle$ to the state $|\psi\rangle$. The inner product is used to calculate transition probabilities and perform other quantum mechanical calculations.

The norm of a vector represents its length or magnitude in the Hilbert space. In quantum mechanics, to ensure valid quantum states, "ket" vectors are normalized to have a unit norm (length). The normalization factor is the square root of the inner product of a "bra" vector with its complex conjugate (the corresponding "ket" vector).

Normalization of Bra Vectors

The `Normalize()` function in the code is designed to normalize a given "bra" vector. It calculates the norm factor of the input "bra" vector using the `NormFactor()` func-

tion and then scales each element of the "bra" vector by the norm factor to obtain a properly normalized "bra" vector.

By normalizing "bra" vectors, we ensure that they represent valid quantum states and satisfy the properties of the Hilbert space. Normalization is crucial for performing quantum mechanical calculations, as it ensures that probabilities are well-defined and physical observables are consistent.

2.2 Ket Class

2.2.1 Description

The `Ket` class is a C++ implementation of a column vector representing a "ket" vector in quantum mechanics. A "ket" vector is a complex-valued column vector that represents a quantum state in a Hilbert space. In quantum mechanics, "ket" vectors are used to represent quantum states in the context of the bra-ket notation.

2.2.2 Template Parameters

- `T`: The data type used for the elements of the "ket" vector. It is typically used with `std::complex<double>` to handle complex numbers used in quantum mechanics.

2.2.3 Constructors

- `Ket()`: Default constructor that creates an empty "ket" vector.
- `Ket(int row)`: Constructor with an integer parameter `row`, which creates an empty "ket" vector with a fixed number of rows specified by `row`.

2.2.4 Member Functions

- `Print()`: Prints the elements of the "ket" vector in fractional symbolic form. For example, for a complex number $z = \frac{A}{B} + i\frac{C}{D}$, where i is the imaginary unit, the function will print the elements of the "ket" vector as $\frac{A}{B}, \frac{C}{D}$ (numerical fractions).
- `NPrint()`: Prints the elements of the "ket" vector in fractional numeric form. For example, for a complex number $z = \frac{A}{B} + i\frac{C}{D}$, the function will print the elements of the "ket" vector as $\frac{A}{B} + i\frac{C}{D}$.

- `ExpectValue(const Eigen::Matrix<T,Eigen::Dynamic,Eigen::Dynamic> &mat, const Ket<T> &ket)`: Computes the expectation value of a given operator (matrix) with a normalized quantum state (ket). The function returns the complex scalar representing the expectation value.
- `NormFactor(const Ket<T> &ket)`: Calculates the norm factor (normalization constant) of the input "ket" vector. The norm factor is computed as the square root of the inner product of the "ket" vector with its complex conjugate (the corresponding "bra" vector). The function returns the norm factor as a complex number.
- `Normalize(const Ket<T> &ket)`: Normalizes the input "ket" vector by dividing each element of the vector by its norm factor. The function returns the normalized "ket" vector.

2.2.5 Physics Explanation

Quantum States and Hilbert Space

In quantum mechanics, a quantum system is represented by a vector space called a Hilbert space. The Hilbert space is a complex vector space that describes the state of the quantum system. Quantum states are represented by vectors in this Hilbert space. A "ket" vector ($|ket\rangle$) is used to represent a quantum state in the Hilbert space.

Inner Product and Expectation Value

The inner product of two vectors in the Hilbert space is a fundamental operation in quantum mechanics. For a "ket" vector $|\psi\rangle$ and a corresponding "bra" vector $\langle\phi|$, the inner product $\langle\phi|\psi\rangle$ calculates the amplitude of transitioning from the state $|\psi\rangle$ to the state $|\phi\rangle$. The inner product is used to calculate transition probabilities and perform other quantum mechanical calculations.

The expectation value of an operator (represented by a matrix) with respect to a quantum state is a crucial quantity in quantum mechanics. It represents the average value of the operator when the system is in the given quantum state. The expectation value is computed as the inner product of the "bra" vector corresponding to the quantum state and the matrix representing the operator, multiplied by the "ket" vector representing the quantum state.

Normalization of Ket Vectors

The `Normalize()` function in the code is designed to normalize a given "ket" vector. It calculates the norm factor of the input "ket" vector using the `NormFactor()` function and then scales each element of the "ket" vector by the norm factor to obtain a properly normalized "ket" vector.

By normalizing "ket" vectors, we ensure that they represent valid quantum states and satisfy the properties of the Hilbert space. Normalization is crucial for performing quantum mechanical calculations, as it ensures that probabilities are well-defined and physical observables are consistent.

2.3 Eigen System

The `EigenSystem` class is a part of the QuantaPlus library and provides support for computing eigenvalues and eigenvectors of a given square matrix. The class allows users to set the matrix to be analyzed and then provides methods to compute specific or all eigenvectors and eigenvalues.

2.3.1 Class Overview

The `EigenSystem` class is designed to compute eigenvalues and eigenvectors for a given square matrix. It uses the Eigen library's `SelfAdjointEigenSolver` to efficiently compute these values. The class provides the following main methods:

`EigenSystem()`

- Constructor that creates an `EigenSystem` object with default values.

`EigenSystem(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mat)`

- Constructor that creates an `EigenSystem` object with the provided matrix `mat`.

`void setMatrix(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mat)`

- Setter method that allows the user to set the matrix to be analyzed.
- **Parameters:** `mat` - A square matrix with dynamic size.

`Eigen::Matrix<T, Eigen::Dynamic, 1> computeEigenVector(int col_number)`

- Method that computes the specified eigenvector of the matrix.
- **Parameters:** `col_number` - The column number of the desired eigenvector.
- **Returns:** The computed eigenvector as an Eigen column vector.

`Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> computeAllEigenVectors()`

- Method that computes all eigenvectors of the matrix.
- **Returns:** A matrix containing all computed eigenvectors, where each column represents an eigenvector.

`T computeEigenValue(int i)`

- Method that computes the specified eigenvalue of the matrix.
- **Parameters:** `i` - The index of the desired eigenvalue.
- **Returns:** The computed eigenvalue.

`Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> computeAllEigenValues()`

- Method that computes all eigenvalues of the matrix.
- **Returns:** A matrix containing all computed eigenvalues as a single column vector.

Example Usage

Below is an example usage of the `EigenSystem` class to compute eigenvalues and eigenvectors for a sample matrix.

Listing 2.1: Example Usage of EigenSystem Class

```
#include <iostream>
#include "Eigen/Dense"
#include "eigensystem.h"

using namespace std;
using namespace Eigen;

int main() {
```

```
// Create a test matrix
MatrixXd test_matrix(3, 3);
test_matrix << 1.0, 2.0, 3.0,
4.0, 5.0, 6.0,
7.0, 8.0, 9.0;

// Create an EigenSystem object and set the matrix
QuantaPlus::EigenSystem<> eigenSystem;
eigenSystem.setMatrix(test_matrix);

// Compute all eigen vectors
MatrixXcd all_eigen_vectors = eigenSystem.computeAllEigenVectors();
cout << "All Eigen Vectors:\n" << all_eigen_vectors << endl;

// Compute the first eigen vector
VectorXcd eigen_vector = eigenSystem.computeEigenVector(0);
cout << "First Eigen Vector:\n" << eigen_vector << endl;

// Compute all eigen values
VectorXcd all_eigen_values = eigenSystem.computeAllEigenValues();
cout << "All Eigen Values:\n" << all_eigen_values << endl;

// Compute the second eigen value
double eigen_value = eigenSystem.computeEigenValue(1);
cout << "Second Eigen Value: " << eigen_value << endl;

return 0;
}
```

2.3.2 Conclusion

The `EigenSystem` class provides an efficient and convenient way to compute eigenvalues and eigenvectors of a square matrix. It encapsulates the functionalities of the Eigen library and makes it easier for users to perform quantum mechanics calculations using the QuantaPlus library.

Chapter 3

Quantum Operators

This code provides the matrix representation of various quantum mechanical operators. The code is written in C++ and contains functions to compute the matrix elements of the harmonic oscillator Hamiltonian operator, position operator, momentum operator, lowering operator, and rising operator. It also includes functions to compute the Hamiltonian of a given system and find the eigenvalues and eigenvectors of a square matrix. Additionally, there are utility functions to calculate wave functions and coefficients for the harmonic oscillator.

3.1 Introduction

The `QM_operator` class is an extension of the `Eigen::Matrix` class that represents quantum mechanical operators. It allows users to perform operations on quantum operators efficiently using the powerful features of the Eigen library. This class is designed to facilitate quantum mechanics calculations and simulations by providing a convenient way to work with quantum operators in matrix form.

3.1.1 `QM_operator` class

Listing 3.1: QM_operator Class Definition

```
template<class T>
class QM_operator: public Eigen::Matrix<T, Eigen::Dynamic
{
        public:
        // Constructors
        QM_operator(): Eigen::Matrix<T, Eigen::Dynamic, E
        QM_operator(int row, int col): Eigen::Matrix<T, E
```

```
        // Constructor from Eigen expressions
        template<typename Derived>
        QM_operator(const Eigen::MatrixBase<Derived>& oth
        : Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic

        // Assignment operator
        template<typename Derived>
        QM_operator& operator=(const Eigen::MatrixBase<De
        {
                this->Eigen::Matrix<T, Eigen::Dynamic, Ei
                return *this;
        }

        // Adjoint (Hermitian conjugate) of the matrix
        Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>
        {
                return this->conjugate().transpose();
        }

        // Print methods
        void Print();
        void NPrint();

        // Destructor
        ~QM_operator() {}
};
```

Class Features

Constructors

- **Default Constructor:** `QM_operator()` creates an empty matrix with dynamic size.
- **Parameterized Constructor:** `QM_operator(int row, int col)` allows creating a matrix with fixed numbers of rows and columns at compile-time. This constructor is useful when the matrix size is known in advance.

Constructor from Eigen expressions

The `QM_operator` class provides a constructor that allows creating a quantum operator from Eigen expressions. This enables the class to work seamlessly within the Eigen framework.

Assignment Operator

The assignment operator (`=`) is overloaded to assign Eigen expressions to `QM_operator` objects.

Adjoint (Hermitian Conjugate)

The `dagger()` method calculates the adjoint (Hermitian conjugate) of the matrix by taking the conjugate of each element and then transposing the matrix. The adjoint of a quantum operator is important in quantum mechanics as it helps define important properties such as observables and unitarity.

Print Methods

The `Print()` and `NPrint()` methods are used to print the elements of the `QM_operator` matrix.

QM_operator Print and NPrint Functions

The `Print` and `NPrint` functions are member functions of the `QM_operator` class. They are used to print the elements of a `QM_operator` matrix in either symbolic or numeric form.

Function Signatures

```
template<class T>
void QM_operator<T>::Print();

template<class T>
void QM_operator<T>::NPrint();
```

Parameters

Both functions do not take any parameters.

Description

The `Print` function prints the elements of the `QM_operator` matrix in fractional symbolic form. It uses the `ComplexNumPrint` function to print complex numbers in the form $z = \frac{A}{B} + i\frac{C}{D}$, where $i = \sqrt{-1}$.

The `NPrint` function prints the elements of the `QM_operator` matrix in fractional numeric form. It also uses the `ComplexNumPrint` function to print complex numbers.

Both functions access the `QM_operator` matrix elements using `Eigen::Matrix` operations and loop through each element to print it.

ComplexNumPrint Function

The `ComplexNumPrint` function is not defined in the provided code snippets. It is assumed to be a separate function that prints complex numbers in the desired format.

Example Usage

```
// Assuming QM_operator<complex> mat is initialized and contains some elements
mat.Print(); // Prints elements of mat in symbolic form
mat.NPrint(); // Prints elements of mat in numeric form
```

KroneckerDelta Function

The `KroneckerDelta` function is a mathematical function that returns 1 if its two input variables are equal, and 0 otherwise. It is used to calculate the Kronecker delta, which is a discrete function often used in linear algebra and quantum mechanics.

Function Signature

```
template<typename T>
int KroneckerDelta(const T& i, const T& j);
```

Parameters

- `i`: The first input variable of type `T`, which is usually a non-negative integer.
- `j`: The second input variable of type `T`, which is usually a non-negative integer.

Return Value

The function returns an integer value of 1 if `i` and `j` are equal, and 0 otherwise.

Description

The `KroneckerDelta` function is a simple implementation of the Kronecker delta function. It uses a ternary operator to check if `i` is equal to `j`, and returns 1 in that case, otherwise it returns 0.

Example Usage

```
int result1 = KroneckerDelta(3, 3); // result1 will be 1
int result2 = KroneckerDelta(2, 5); // result2 will be 0
```

KroneckerProduct Function

The `KroneckerProduct` function calculates the Kronecker product of two given matrices. The Kronecker product is a tensor product operation in linear algebra and quantum mechanics.

Function Signature

```
template <class T>
Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>
KroneckerProduct(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt1,
const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt2);
```

Parameters

- `mt1`: The first input matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` for which the Kronecker product will be computed.
- `mt2`: The second input matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` for which the Kronecker product will be computed.

Return Value

The function returns a new matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` containing the Kronecker product of `mt1` and `mt2`.

Description

The Kronecker product of two matrices `mt1` and `mt2` is defined as a new matrix obtained by taking each element of `mt1` and multiplying it with the entire matrix

mt2. The resulting matrix has dimensions equal to the product of the dimensions of mt1 and mt2.

Mathematically, the Kronecker product of matrices mt1 and mt2 is denoted by mt1 $\otimes$ mt2.

The function loops through each element of mt1 and mt2, and calculates the corresponding element of the result matrix by multiplying them together.

Example Usage

```
Eigen::Matrix2d mat1;
mat1 << 1, 2,
3, 4;
Eigen::Matrix2d mat2;
mat2 << 5, 6,
7, 8;
Eigen::Matrix4d result = KroneckerProduct(mat1, mat2);
```

Overloaded KroneckerProduct Function

The KroneckerProduct function is overloaded to support the calculation of the Kronecker product of three matrices. This version of the function performs the Kronecker product in a nested manner.

Function Signature

```
template <class T>
Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>
KroneckerProduct(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt1,
const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt2,
const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt3);
```

Parameters

- mt1: The first input matrix of type Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> for which the Kronecker product will be computed.

Overloaded KroneckerProduct Function

The KroneckerProduct function is overloaded to support the calculation of the Kronecker product of three matrices. This version of the function performs the Kronecker product in a nested manner.

Function Signature

```
template <class T>
Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>
KroneckerProduct(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt1,
const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt2,
const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& mt3);
```

Parameters

- `mt1`: The first input matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` for which the Kronecker product will be computed.
- `mt2`: The second input matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` for which the Kronecker product will be computed.
- `mt3`: The third input matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` for which the Kronecker product will be computed.

Return Value

The function returns a new matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` containing the Kronecker product of `mt1`, `mt2`, and `mt3`.

Description

The overloaded `KroneckerProduct` function performs the Kronecker product of three matrices by nesting the calculation of the Kronecker product of two matrices twice. First, it calculates the Kronecker product of `mt1` and `mt2` using the original `KroneckerProduct` function. Then, it calculates the Kronecker product of the resulting matrix and `mt3`.

The resulting matrix has dimensions equal to the product of the dimensions of `mt1`, `mt2`, and `mt3`.

Example Usage

```
Eigen::Matrix2d mat1;
mat1 << 1, 2,
3, 4;
Eigen::Matrix2d mat2;
mat2 << 5, 6,
7, 8;
Eigen::Matrix2d mat3;
```

```
mat3 << 9, 10,
11, 12;
Eigen::Matrix8d result = KroneckerProduct(mat1, mat2, mat3);
```

Id Function

The `Id` function is used to create an identity matrix of dynamic size for a given spin quantum number.

Function Signature

```
template<typename T>
Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> Id(const double& S);
```

Parameters

- `S`: The spin quantum number of type `double` for which the identity matrix will be created.

Return Value

The function returns an identity matrix of type `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>` with dimensions `(2S+1) x (2S+1)`.

Description

The `Id` function creates an identity matrix with dimensions `(2S+1) x (2S+1)` for a given spin quantum number `S`. The identity matrix is a square matrix with ones on the main diagonal and zeros elsewhere.

The function uses the `Eigen::Matrix Identity` method to create the identity matrix.

Example Usage

```
double S = 0.5; // Spin quantum number
Eigen::Matrix2d identity = Id<double>(S);
```

3.1.2 Physics Representation

The `QM_operator` class represents quantum mechanical operators in matrix form. In quantum mechanics, operators are mathematical objects that represent physical observables. Examples of quantum mechanical operators include position, momentum,

angular momentum, and energy operators. These operators act on quantum states (represented as kets) to produce new quantum states or measurement outcomes.

By extending the `Eigen::Matrix` class, the `QM_operator` class allows users to perform matrix operations on quantum operators efficiently. The adjoint of the matrix (`dagger()` method) is particularly important in quantum mechanics, as it is used to calculate the Hermitian conjugate of operators, which is crucial for defining Hermitian operators and unitary transformations.

Using the `QM_operator` class, researchers and students in quantum mechanics can easily perform calculations and simulations involving quantum operators and explore the properties and behaviors of quantum systems.

Conclusion

The `QM_operator` class is a well-implemented extension of the `Eigen::Matrix` class that represents quantum mechanical operators. It provides a convenient and efficient way to work with quantum operators in matrix form, enabling researchers and students to perform various quantum mechanics calculations and simulations with ease.

3.2 Harmonic oscillator

Includes

The code includes several headers:

- `operators.h`: Contains common operator functionalities.
- `eigenvec.h`: Provides functions to compute eigenvalues of a matrix.
- `eigenval.h`: Provides functions to compute eigenvectors of a matrix.

3.2.1 Harmonic Oscillator Matrix Elements

`HOscillatorMatrixElements(const int& R, const int& C)`: This function computes the matrix representation of the harmonic oscillator Hamiltonian operator in the N-space. The elements are given by $\langle n'|\hat{H}|n\rangle = \frac{\hbar\omega}{2}(n + \frac{1}{2})\delta_{n'n}$, where n and n' are the matrix indices, and $\hbar$ is the reduced Planck's constant, and ω is the angular frequency.

3.2.2 Position Operator

`PositionOperator(const int& R, const int& C)`: This function computes the matrix representation of the position operator in the N-space. The elements are given by $X = \sqrt{\frac{\hbar}{2m\omega}}(\sqrt{n}\delta_{n',n-1} + \sqrt{n+1}\delta_{n',n+1})$, where m is the mass of the particle.

3.2.3 Momentum Operator

`MomentumOperator(const int& R, const int& C)`: This function computes the matrix representation of the momentum operator in the N-space. The elements are given by $P = i\sqrt{\frac{m\hbar\omega}{2}}(-\sqrt{n}\delta_{n',n-1} + \sqrt{n+1}\delta_{n',n+1})$.

3.2.4 Lowering Operator

`LoweringOperator(const int& R, const int& C)`: This function computes the matrix representation of the lowering operator (also known as the annihilation operator) in the N-space. The elements are given by $\langle n'|\hat{a}|n\rangle = \sqrt{n}\delta_{n'n-1}$, where $\hat{a}$ is the lowering operator.

3.2.5 Rising Operator

`RisingOperator(const int& R, const int& C)`: This function computes the matrix representation of the rising operator (also known as the creation operator) in the N-space. The elements are given by $\langle n'|\hat{a}^\dagger|n\rangle = \sqrt{n+1}\delta_{n'n+1}$, where $\hat{a}^\dagger$ is the rising operator.

3.2.6 Hamiltonian

`Hamiltonian(QM_operator<T> A, QM_operator<T> B, Bra<T> coff)`: This function computes the Hamiltonian of a quantum mechanical system using the given matrices A and B and the row vector $coff$. It returns a matrix of the same size as A.

3.2.7 Eigenvalues and Eigenvectors

`eval(QM_operator<T>& A)`: This function returns the eigenvalues of the square matrix A as a dynamic-size matrix.

`evec(QM_operator<T>& A)`: This function returns the eigenvectors of the square matrix A as a dynamic-size matrix.

`evec(const QM_operator<T>& A, const QM_operator<T>& B, const Bra<T>& coff)`: An overloaded version of `evec`, computes eigenvectors using Hamiltonian matrices obtained from A, B, and the row vector $coff$.

3.2.8 Wave Functions and Coefficients

`Factorial(const int& n)`: This function computes the factorial of a given integer n.

`wave_function(const int& n, const double& x)`: This function computes the wave function for a given order n and position x for the harmonic oscillator.

`coefficients(const QM_operator<T>& A, const QM_operator<T>& B, const Bra<T>& coff, const int& n)`: This function computes the coefficients of the wave function for a given order n using the matrices A and B and the row vector $coff$.

`wave_function_H(const QM_operator<T>& A, const QM_operator<T>& B, const Bra<T>& coff, const double& x, const int& n)`: This function computes the value of the wave function for a given order n and position x using the matrices A and B and the row vector $coff$.

3.3 Angular Momentum

3.3.1 Introduction

`AngularMomentum.h` is a header file that is a part of the `QUANTAPLUS` library. It defines and handles some fundamental quantum operators related to angular momentum. These operators play a crucial role in quantum mechanics, particularly in systems with rotational symmetry.

3.3.2 Header Summary

`AngularMomentum.h` provides the following functionality:

- Definition of the `AngularMomentum` class, representing matrix representations of angular momentum operators.
- Various methods to compute the matrix representations of angular momentum operators, such as $\hat{J}^2$, $\hat{J}_x$, $\hat{J}_y$, $\hat{J}_z$, $\hat{J}_+$, and $\hat{J}_-$.
- Utility functions for checking the validity of angular momentum values and computing Kronecker Delta.

3.3.3 Class `AngularMomentum`

`AngularMomentum` is a publicly derived class from `Eigen::Matrix<T,Eigen::Dynamic,Eigen::Dynam` representing a square matrix with dynamically sized rows and columns. The class is templated on the data type `T`, which is typically a complex number type.

Constructors

- `AngularMomentum()`: Default constructor, creates an empty matrix.
- `AngularMomentum(int row, int col)`: Constructor with row and column inputs, creates an empty matrix with specified dimensions.
- `AngularMomentum(const Eigen::MatrixBase<Derived>& other)`: Constructor to construct an `AngularMomentum` from Eigen expressions.

3.3.4 Matrix Representation Methods

- `AngularMomentum_JSquare(const double& spin_value)`: Returns the matrix representation of the squared general angular momentum operator $\hat{J}^2$.
- `AngularMomentum_Jx(const double& spin_value)`: Returns the matrix representation of the angular momentum operator component $\hat{J}_x$.
- `AngularMomentum_Jy(const double& spin_value)`: Returns the matrix representation of the angular momentum operator component $\hat{J}_y$.
- `AngularMomentum_Jz(const double& spin_value)`: Returns the matrix representation of the angular momentum operator component $\hat{J}_z$.
- `AngularMomentum_JPlus(const double& spin_value)`: Returns the matrix representation of the raising operator $\hat{J}_+$.
- `AngularMomentum_JMinus(const double& spin_value)`: Returns the matrix representation of the lowering operator $\hat{J}_-$.

Utility Functions

- `validAngularMomentum(const double& x)`: Checks the validity of the given angular momentum value `x`, returning `true` only if `x` is a nonnegative half-integer value.
- `KroneckerDelta(const double &i, const double &j)`: Computes the Kronecker Delta function δ_{ij}, returning 1 if $i = j$ and 0 otherwise.

3.4 Clebsch-Gordan coefficients

3.4.1 Introduction

`cgc.h` is a part of the QUANTAPLUS library. It provides functions and classes for calculating various quantum mechanical properties, including the matrix representation of operators, Clebsch-Gordan coefficients, eigenvalues, and eigenvectors. The library is designed to assist in quantum mechanical calculations and simulations.

3.4.2 CGCcoeff Class

The `CGCcoeff` class represents Clebsch-Gordan coefficients ($\langle j_1, j_2; m_1, m_2 | j, m \rangle$). These coefficients describe the coupling of angular momenta in quantum systems.

3.4.3 Physics Behind Clebsch-Gordan Coefficients

In quantum mechanics, when two angular momenta (j_1 and j_2) are combined to form a new state with total angular momentum j, the Clebsch-Gordan coefficients give the probability amplitudes of different j states. The coefficients depend on the individual magnetic quantum numbers (m_1 and m_2) and the total magnetic quantum number (m).

3.4.4 Class Members

- `j1`: First particle's angular momentum (spin or orbital).
- `j2`: Second particle's angular momentum (spin or orbital).
- `m1`: Magnetic quantum number of the first particle (or system).
- `m2`: Magnetic quantum number of the second particle (or system).
- `j`: Total angular momentum: $|j_1 - j_2| \leq j \leq |j_1 + j_2|$.
- `m`: Total magnetic quantum number: $-j \leq m \leq j$.

3.4.5 Matrix Representation of Operators

The library provides functions to calculate the matrix representation of various operators in quantum mechanics, such as the harmonic oscillator Hamiltonian, position operator, momentum operator, lowering operator, and rising operator.

3.4.6 Physics Behind Matrix Representation of Operators

In quantum mechanics, operators represent physical observables. The matrix representation of an operator in a specific basis provides a way to calculate the expectation values of the operator and perform various quantum mechanical calculations.

3.4.7 Functions for Matrix Representation

- `HOscillatorMatrixElements(R, C)`: Returns the matrix representation of the harmonic oscillator Hamiltonian operator in the N-space. The elements are given by $\langle n'|H|n\rangle = \hbar\omega\left(n+\frac{1}{2}\right)\delta_{n',n}$, where n and n' are quantum numbers, $\hbar$ is the reduced Planck constant, and ω is the angular frequency.
- `PositionOperator(R, C)`: Returns the matrix representation of the position operator in the N-space. The elements are given by $X = \sqrt{\frac{\hbar}{2m\omega}}\left(\sqrt{n}\delta_{n',n-1} + \sqrt{n+1}\delta_{n',n+1}\right)$, where m is the particle's mass.
- `MomentumOperator(R, C)`: Returns the matrix representation of the momentum operator in the N-space. The elements are given by $P = i\sqrt{\frac{m\hbar\omega}{2}}\left(-\sqrt{n}\delta_{n',n-1} + \sqrt{n+1}\delta_{n',n+}\right.$
- `LoweringOperator(R, C)`: Returns the matrix elements of the lowering operator (annihilation operator a). The elements are given by $\langle n'|a|n\rangle = \sqrt{n}\delta_{n',n-1}$.
- `RisingOperator(R, C)`: Returns the matrix elements of the rising operator (creation operator $a^\dagger$). The elements are given by $\langle n'|a^\dagger|n\rangle = \sqrt{n+1}\delta_{n',n+1}$.

3.4.8 Eigenvalues and Eigenvectors

The library provides functions to calculate eigenvalues and eigenvectors of square matrices.

3.4.9 Physics Behind Eigenvalues and Eigenvectors

In quantum mechanics, the eigenvalues and eigenvectors of a matrix correspond to the energy levels and the associated states of a quantum system. These are crucial for solving quantum mechanical problems and understanding the system's behavior.

3.4.10 Functions for Eigenvalues and Eigenvectors

- `eval(A)`: Calculates the eigenvalues of the square matrix `A`.

- `evec(A)`: Calculates the eigenvectors of the square matrix `A`.
- `evec(A, B, coff)`: Overloaded function to calculate eigenvectors for a custom Hamiltonian `A + B` with coefficients provided in the row vector `coff`.

3.4.11 Wave Functions and Coefficients

The library provides functions to calculate wave functions and coefficients.

3.4.12 Physics Behind Wave Functions and Coefficients

Wave functions describe the probability amplitudes of finding a particle in a specific state. Coefficients are coefficients used to expand a wave function in terms of a basis set. In quantum mechanics, the wave function provides information about the state of a quantum system and allows us to calculate various observables.

3.4.13 Functions for Wave Functions and Coefficients

- `WaveFunction(x, n, omega)`: Calculates the wave function of a quantum harmonic oscillator in position space. The function takes the position `x`, the quantum number `n`, and the angular frequency `omega` as inputs and returns the value of the wave function at the given position.
- `Coefficients(n, omega)`: Calculates the coefficients for expanding the wave function of a quantum harmonic oscillator in the basis of energy eigenstates. The function takes the quantum number `n` and the angular frequency `omega` as inputs and returns the coefficients as a vector.
- `AngularMomentumJ(j1, j2, m1, m2, J, M)`: Calculates the Clebsch-Gordan coefficient $\langle j_1, j_2; m_1, m_2 | J, M \rangle$ using the generalized power series representation. The function takes the individual angular momenta j_1, j_2, magnetic quantum numbers m_1, m_2, and the total angular momentum J and magnetic quantum number M as inputs and returns the Clebsch-Gordan coefficient.
- `ListOfAllCGCs(j1, j2)`: Prints the list of all Clebsch-Gordan coefficients for the addition of two angular momenta j_1 and j_2. The function takes the individual angular momenta j_1 and j_2 as inputs and prints the Clebsch-Gordan coefficients for all possible J values.
- `CGCcoeffMap(j1, j2)`: Returns a map that stores Clebsch-Gordan coefficients as keys and their corresponding possible values as vectors. The function takes the individual angular momenta j_1 and j_2 as inputs and returns the map.

- `CoupledStates(j1, j2)`: Prints the states of the coupled system formed by combining angular momenta j_1 and j_2. The function takes the individual angular momenta j_1 and j_2 as inputs and prints the states of the coupled system along with their coefficients in terms of Clebsch-Gordan coefficients.

3.4.14 Conclusion

The QUANTAPLUS library provides useful functions and classes for performing various quantum mechanical calculations, including the matrix representation of operators, Clebsch-Gordan coefficients, eigenvalues, and eigenvectors. It is designed to assist researchers and students in the field of quantum mechanics in understanding and solving complex quantum mechanical problems.

Chapter 4

utilities

The "utilities.h" file is a part of the QUANTAPLUS library, which contains utility classes and functions to handle output operations and execution time calculations.

4.1 Output Class

Description

The `Output` class provides functionality to handle file output operations. It allows writing data to a file specified by the user.

Public Members

- `std::ofstream takeData`: Output stream class object to operate on files.

Constructor

- `Output(const std::string& fileName)`: Constructor that receives an input string representing the file name (without the extension). It opens the file with the extension ".dat".

Destructor

- `~Output()`: Destructor to close the output file when the object goes out of scope.

Member Functions

- `void writeOutput(const std::string& output)`: Function to write output data to the file. It takes a string (`output`) representing the data to be written to the file and appends a new line after writing the data.

4.2 ElapsedTime Class

Description

The `ElapsedTime` class provides functionality to calculate and display information about the execution time of a program or specific code section.

Public Members

- `std::clock_t start`: Represents the starting time of the execution.
- `double duration`: Represents the total duration of the execution in seconds.

Member Functions

- `void Start()`: Function to start the timer. It records the current clock time in `start` member variable.
- `void End()`: Function to end the timer and display the elapsed time. It calculates the duration of the execution by subtracting the starting time from the current time, and then displays the elapsed time in minutes and seconds.

4.3 Utility Functions

Description

The header file also contains several utility functions to assist in other operations.

Function List

- `bool IsNumber(const std::string& str)`: Function to check if the input string is a valid integer (contains only digits).
- `bool validInteger(const double& x)`: Function to check if the input floating-point number is a valid integer (e.g., 3.0, -5.0).

- `bool halfInteger(const double& x)`: Function to check if the input floating-point number is a half-integer (e.g., 1.5, -2.5).

Conclusion

The "utilities.h" file contains classes and utility functions that provide helpful functionalities for handling file output and measuring execution time. These utilities can be used in C++ projects to facilitate output operations and performance monitoring.

4.4 ToString Function

Template Function: `ToString`

Description: A template function to convert any type of input into a string representation.

Usage:

```
template <typename T>
std::string ToString(const T& input)
```

Parameters:

- `input` (generic): The input value of any type that needs to be converted into a string.

Return:

- `std::string`: The converted string representation of the input value.

4.5 ToFraction Function

Template Function: `ToFraction`

Description: A template function to print two numbers in literal fraction form (numerator/denominator).

Usage:

```
template <typename T1, typename T2>
std::string ToFraction(T1 numerator, T2 denominator)
```

Parameters:

- `numerator` (generic): The numerator of the fraction.

- `denominator` (generic): The denominator of the fraction.

Return:

- `std::string`: The string representation of the fraction in the form "numerator/denominator".

4.6 MatrixToString Function

Function: `MatrixToString`

Description: A function to print the elements of a dynamic-size matrix in fractions form.

Usage:

```
template <typename T>
void MatrixToString(T& matrix)
```

Parameters:

- `matrix` (Dynamic-size matrix): A matrix with numeric elements that needs to be printed in fraction form.

Note: The function directly prints the elements of the matrix in fractions format, separated by tabs.

4.7 DecimalToFraction Function

Function: `DecimalToFraction`

Description: A function to convert decimal numbers into fraction form, considering the square roots of the numerator and denominator if they are not integers.

Usage:

```
void DecimalToFraction(const double& decimal_number)
```

Parameters:

- `decimal_number` (double): The decimal number that needs to be converted into fraction form.

Note: The function directly prints the fraction representation of the decimal number, considering the square roots of numerator and denominator if applicable.

4.8 ComplexNumPrint Function

Function: `ComplexNumPrint`

Description: A function to print complex numbers in symbolic form, considering both real and imaginary parts.

Usage:

```
void ComplexNumPrint(const std::complex<double>& complex_number, bool numerical_flag
```

Parameters:

- `complex_number` (complex): The complex number that needs to be printed.
- `numerical_flag` (boolean): A flag to switch between displaying numerical or symbolic output.

Note: The function prints the complex number in symbolic form if `numerical_flag = 0`, and in numerical form if `numerical_flag = 1`.

4.9 ResultPrint Function

Function: `ResultPrint`

Description: A function to print complex numbers in symbolic form.

Usage:

```
void ResultPrint(const std::complex<double>& complex_num)
```

Parameters:

- `complex_num` (complex): The complex number that needs to be printed in symbolic form.

Note: The function directly prints the complex number in symbolic form.

4.10 NResultPrint Function

Function: `NResultPrint`

Description: A function to print complex numbers in numerical form.

Usage:

```
void NResultPrint(const std::complex<double>& complex_num)
```

Parameters:

- `complex_num` (complex): The complex number that needs to be printed in numerical form.

Note: The function directly prints the complex number in numerical form.

Summary

The provided C++ code defines a namespace named "QuantaPlus" that contains utility functions for working with fractions and complex numbers. The main functions include:

- `ToString`: Converts any type of input into a string representation.
- `ToFraction`: Prints two numbers in literal fraction form (numerator/denominator).
- `MatrixToString`: Prints the elements of a dynamic-size matrix in fractions form.
- `DecimalToFraction`: Converts decimal numbers into fraction form, considering the square roots of the numerator and denominator if applicable.
- `ComplexNumPrint`: Prints complex numbers in symbolic or numerical form, considering both real and imaginary parts.
- `ResultPrint`: Prints complex numbers in symbolic form.
- `NResultPrint`: Prints complex numbers in numerical form.

The code includes templates to handle various types of input, making the utility functions flexible and reusable across different data types.

4.10.1 Usage

To use these utility functions, include the provided header file "braket_print.h" in your C++ code. You can then use the functions within the "QuantaPlus" namespace.

For example, to convert a decimal number to its fraction form:

```
double decimal_number = 3.14159;
QuantaPlus::DecimalToFraction(decimal_number);
```

To print a complex number in symbolic form:

```
#include <complex>
std::complex<double> complex_number(2.0, -3.0);
QuantaPlus::ResultPrint(complex_number);
```

To print the elements of a matrix in fractions form:

```
#include <Eigen/Dense>
Eigen::MatrixXd matrix(2, 2);
matrix << 1.5, 2.25, 0.75, 3.0;
QuantaPlus::MatrixToString(matrix);
```

You can customize the output format by setting the `numerical_flag` parameter of the `ComplexNumPrint` function to 1, which will print complex numbers in numerical form. For example:

```
QuantaPlus::ComplexNumPrint(complex_number, 1);
```

This will output the complex number as "2.0 - 3.0i" instead of the symbolic representation.

Conclusion

The "QuantaPlus" namespace provides useful utility functions for working with fractions and complex numbers. These functions can be easily integrated into C++ projects, offering flexibility in handling different data types and providing the ability to print complex numbers in both symbolic and numerical forms. The code can be extended and adapted as needed to suit various mathematical and scientific applications.

4.11 Latex File generator

Introduction

The ‘latex.h‘ header file provides functionalities to generate LaTeX files for outputting results. It was developed by Mohammed Maher Abdelrahim Mohammed at the Università della Calabria, Dipartimento di Fisica, and INFN-Cosenza, Italy. The header file allows users to create LaTeX documents containing mathematical equations and various quantum mechanics objects.

Greek Letters Map

The header file defines a map named `GreekLatters`, which contains the Unicode escape characters for both capital and lower Greek letters. This map is used for converting the string names of Greek letters into their LaTeX representations when printing mathematical symbols.

Function: `Greek(letter_name)`

- Summary: This function takes a string `letter_name` as input, which should be the name of a Greek letter (e.g., "Alpha", "beta", "gamma"). It returns the corresponding LaTeX representation of the Greek letter using the `GreekLatters` map.
- Returns: A string containing the LaTeX representation of the given Greek letter.

LaTex Class

The `LaTex` class is the main class provided by the 'latex.h' header file. It allows users to create LaTeX files, set up LaTeX document settings, and write equations and text directly into the LaTeX file.

Constructor: `LaTex()` **and** `LaTex(filename)`

- `LaTex()`: Default constructor that sets the default file name and path for the LaTeX output file to "output/default.tex".
- `LaTex(filename)`: Constructor that allows users to specify the file name and path for the LaTeX output file.

Method: `BeginLaTex()`

- Summary: This method initiates the LaTeX typesetting document class and sets up the necessary LaTeX packages and settings. It opens the LaTeX output file for writing.
- Returns: None.

Method: `EndLaTex()`

- Summary: This method closes the LaTeX output file and adds the necessary LaTeX command to end the document.
- Returns: None.

Method: `Typing(text)`

- Summary: This method allows users to directly write normal text or even LaTeX commands into the LaTeX output file. It appends the provided `text` to the LaTeX file.

- Returns: None.

Method: `ToLaTex(fmt...)`

- Summary: This method allows users to convert multiple inputs into LaTeX format and write them as equations into the LaTeX output file. It uses the `fmt` argument as a format specifier to indicate the type of input (e.g., ket, bra, operator, text) and their corresponding LaTeX representation.
- Arguments:
  - `fmt`: Multiple variables with identifier format "fmt" to indicate the type of input and its corresponding LaTeX representation.
- Returns: None.

Method: `MathOperation(fmt...)`

- Summary: This method allows users to convert multiple mathematical operations into LaTeX equations and write them into the LaTeX output file. It uses the `fmt` argument as a format specifier to indicate the type of input (e.g., ket, bra, operator, complex number, text) and their corresponding LaTeX representation.
- Arguments:
  - `fmt`: Multiple variables with identifier format "fmt" to indicate the type of input and its corresponding LaTeX representation.
- Returns: None.

Method: `LaTexMath(function_name)`

- Summary: This method converts the string name of a math function into its equivalent LaTeX command. It uses the `MathFunctions` map to look up the corresponding LaTeX command for the given `function_name`.
- Arguments:
  - `function_name`: The string name of the math function (e.g., "Exp", "Sin", "Sqrt").
- Returns: A string containing the LaTeX command for the given math function.

Conclusion

The ‘latex.h‘ header file provides a convenient way to generate LaTeX files for displaying mathematical equations and quantum mechanics objects. Users can create LaTeX documents and include various quantum mechanics elements, such as kets, bras, operators, and complex numbers, in a visually appealing format for research papers, presentations, or educational materials.