Computer Physics Communications 196 (2015) 569–591

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

A Simple Quantum Integro-Differential Solver (SQuIDS)☆,☆☆

Carlos A. Argüelles Delgado, Jordi Salvado*, Christopher N. Weaver

Department of Physics, University of Wisconsin, 1150 University Ave., Madison, WI 53706-1390, USA
Wisconsin IceCube Particle Astrophysics Center, 222 W. Washington Ave., Suite 500, Madison, WI 53703, USA

ARTICLE INFO

Article history:
Received 19 January 2015
Received in revised form
13 May 2015
Accepted 29 June 2015
Available online 6 July 2015

Keywords:
Quantum mechanics
Ordinary differential equations
SU groups

ABSTRACT

Simple Quantum Integro-Differential Solver (SQuIDS) is a C++ code designed to solve semi-analytically the evolution of a set of density matrices and scalar functions. This is done efficiently by expressing all operators in an $SU(N)$ basis. SQuIDS provides a base class from which users can derive new classes to include new non-trivial terms from the right hand sides of density matrix equations. The code was designed in the context of solving neutrino oscillation problems, but can be applied to any problem that involves solving the quantum evolution of a collection of particles with Hilbert space of dimension up to six.

Program summary

Program title: SQuIDS

Catalogue identifier: AEXG_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEXG_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: GNU Lesser General Public License, version 3

No. of lines in distributed program, including test data, etc.: 18198

No. of bytes in distributed program, including test data, etc.: 137607

Distribution format: tar.gz

Programming language: C++11.

Computer: 32- and 64-bit x86.

Operating system: Linux, Mac OS X, FreeBSD.

RAM: Proportional to the number of nodes, the dimension of the Hilbert space, the number of scalar functions, and the number of density matrices used in the problem.

Classification: 11.1.

External routines: GNU Scientific Library (http://www.gnu.org/software/gsl/).

Nature of problem: Solve the evolution of open quantum systems of Hilbert space dimension N with self interactions and interaction with classical fields.

Solution method: The $SU(N)$ algebra is implemented as a C++ object and is embedded into the GSL ordinary differential equation solver.

Restrictions: The code is only implemented up to Hilbert spaces of dimension six, but a Mathematica notebook is provided in order to generate higher dimensional solutions. Furthermore, only ordinary differential equation solution methods that require only the first derivative can be used.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

☆☆ The code can be found in https://github.com/jsalvado/SQuIDS.

* Corresponding author at: Department of Physics, University of Wisconsin, 1150 University Ave., Madison, WI 53706-1390, USA.
E-mail addresses: carlos.arguelles@icecube.wisc.edu (C.A. Argüelles Delgado), jsalvado@icecube.wisc.edu (J. Salvado), cweaver@icecube.wisc.edu (C.N. Weaver).

http://dx.doi.org/10.1016/j.cpc.2015.06.022

Running time: Proportional to the number of nodes, the dimension of the Hilbert space, the number of scalar functions, the number of density matrices, and the numerical precision used in the problem.

1. Introduction

The evolution of an ensemble of neutrinos is a many body quantum mechanics problem, where every neutrino is represented by a state in a Hilbert space.

When all neutrinos are produced in the same quantum state and there are no interactions, a pure quantum state evolution is a good approximation. This can be implemented by solving the corresponding Schrödinger equation.

On the other hand, especially at high energies, there are interactions which can mix the states. In this case, the system is represented by a mixed state and therefore the evolution is more naturally described in the context of the density matrix formalism [1–3].

The possibility of using the symmetries of the problem in order to efficiently solve the evolution, i.e. differential equations, is a well studied topic in applied Mathematics [4]. In this paper, we apply this principle to make explicit the physical degrees of freedom, thus enabling us to solve analytically the evolution due to the time independent part of the Hamiltonian (H_0). This approach has been also used in the context of two neutrino flavor approximations [3].

The Simple Quantum Integro-Differential Solver (SQuIDS) is a code that implements the evolution of a set of density matrices and scalar functions using a semi-analytic approach. The code is written in object-oriented C++ and contains a base class which sets up the problem by means of virtual member functions. This allows the user to create a derived class which defines the right hand side terms of the differential equation via its implementation of these functions. Numerical integration is performed using the GNU Scientific Library [5].

The code works with a Hilbert space of dimension up to six and allows the inclusion of an arbitrary number of density matrices and scalar functions. The physical degrees of freedom in the problem can be represented in the basis of the generators of the $SU(N)$ group plus the identity, where N is the dimension of the Hilbert space, so we write all operators in terms of this basis. The evolution generated by the time independent part of the Hamiltonian and basis changes can be thought as the action of the unitary group on the operators. These transformations are included as analytic expressions in the code. Then, in order to make the numerical solution efficient we solve the evolution of the entire system in the interaction picture [6].

The paper is structured in the following sections: In Section 3 we introduce the density matrix formalism. In Section 2, we comment on other approaches to solve similar problems. In Section 4, we describe the evolution equations in the context of the $SU(N)$ algebra. In Section 5, we describe how this formulation is implemented in the code. Finally, in Section 7, we include three simple examples to illustrate some applications: neutrino oscillations in the, Rabi oscillations, and collective neutrino oscillations.

2. Related work

Solving quantum mechanics problems using computers has been a long standing topic of study. Examples of codes that solve the Schrödinger equation in the position representation have been published for both stationary [7] and time evolving [8–10] many body systems. On the other hand, representation of finite, quantum-mechanical, closed systems has been developed in order to perform quantum computing calculations (see e.g. [11]). In high energy particle physics the need for precise neutrino oscillation calculation has encouraged the development of ad hoc tools to solve two or three level closed quantum systems subject to time varying potentials (see e.g. [12–14]). This work goes beyond the pure state representation by using the density matrix formalism, which allows us to express in a natural way mixed states, as well as treating open quantum systems. To do this, we have developed a highly efficient representation of states and operators in terms of the generators of the $SU(N)$ group.

3. Density matrix formalism

3.1. Definition

In quantum mechanics the state of system is given by a vector in a Hilbert space, i.e. $|\phi_i\rangle \in \mathcal{H}_i$, where $\mathcal{H}_i$ is the Hilbert space for the ith particle. In general we can be interested in solving a system of many particles and for that the Hilbert space is constructed as $\mathcal{H} = \bigotimes_i \mathcal{H}_i$. For a large number of particles the dimension of this space grows exponentially. Nevertheless in the limit where the different particles do not have quantum correlations and $\mathcal{H}_i$ is the same for all of the single particles of the system, we can approximate the system as a statistical ensemble of single particle quantum states, which is known as a mixed state.

To describe a system in this limit it is convenient to introduce the density matrix formalism. For a given set of states $\{\langle\psi_i| \, | i = 1, \ldots, n\}$ and a set of positive real numbers $\{p_i | \sum_i p_i = 1\}$, that are physically interpreted as the probability of the system to be in the ith state, the density operator, which represents the mixed state, is constructed as

$$\hat{\rho} = \sum_i p_i \, |\psi_i\rangle \, \langle\psi_i| \, . \tag{1}$$

For a particular basis $\{|u_1\rangle \, , |u_2\rangle \, , \ldots , |u_n\rangle\}$ of the Hilbert space the components of the density matrix can be written as

$$\rho_{lm} = \sum_i p_i \, \langle u_l| \ \psi_i\rangle \, \langle\psi_i| \ u_m\rangle \, , \tag{2}$$

in other words, for a given basis the density operator is represented by a $n \times n$ Hermitian matrix ρ.

Is also useful to define the projectors to the i-state of a basis,

$$\hat{\Pi}^{(i)} = |u_i\rangle \langle u_i| . \tag{3}$$

The expectation value for any operator $\hat{O}$ when the system is described by $\hat{\rho}$ is given by the trace of the matrix product

$$\left\langle \hat{O} \right\rangle_\rho = \sum_{lm} O_{lm} \rho_{lm}, \tag{4}$$

where O_{lm} are the components that represent $\hat{O}$ in the same basis as ρ_{lm}.

In particular, the expectation value of the projector gives the probability of finding the system in the i-state, and is

$$\left\langle \hat{\Pi}^{(i)} \right\rangle_\rho = \sum_{lm} \Pi^{(i)}_{lm} \rho_{lm} = p_i. \tag{5}$$

3.2. Unitary transformations

In general in quantum mechanics, the transformation from one basis to another is always given by a unitary transformation. In this section we describe a way of writing a general unitary transformation in terms of a set of mixing angles and phases. This parametrization is widely used in the context of neutrino physics and is included in the code.

This parametrization can be written as product of two dimensional rotations, in particular,

$$U(\theta_{ij}\delta_{ij}) = R_{N-1\,N} R_{N-2\,N} \ldots R_{45} R_{35} R_{25} R_{15} R_{34} R_{24} R_{14} R_{23} R_{13} R_{12}, \tag{6}$$

where each matrix R_{ij} is a rotation in the ij plane with rotation angle θ_{ij} and complex phase δ_{ij}, namely

$$R_{ij} = \begin{pmatrix} \ddots & & & & \\ & \cos\theta_{ij} & \cdots & \sin\theta_{ij} e^{-i\delta_{ij}} & \\ & \vdots & & \vdots & \\ & -\sin\theta_{ij} e^{i\delta_{ij}} & \cdots & \cos\theta_{ij} & \\ & & & & \ddots \end{pmatrix}. \tag{7}$$

In a minimal description only a subset of the δ_{ij} are required to be allowed to be non-zero.

An operator in the Hilbert space is transformed using the relation

$$O_{lm} \to \sum_{kn} U^\dagger_{lk} O_{kn} U_{nm}. \tag{8}$$

3.3. System definition and time evolution

In this library, we choose to describe a quantum system as a collection of 'nodes', labeled by a parameter x, where a node is an object with a set of density matrices and scalar functions. For example, when studying a system of neutrinos and charged leptons, x can be chosen to correspond to particle energy, and each node can represent the state of the particles at a particular energy, where the state of the neutrinos is a density matrix, and lepton fluxes are scalars. This system might describe the neutrinos within a supernova, where scattering interactions relate the neutrinos and leptons at different energies (different nodes). For contrast, one might also consider atoms in a lattice subject to an external potential. Here, each node may simply correspond to a single lattice site, which contains a density matrix representing the spin state and a scalar representing the potential at the site. For notational convenience we define $\tilde{\rho}$ and $\tilde{S}$ as the set of all density matrices with elements ρ_J and set of all scalar functions with elements S_K, respectively, where J and K are general indices to label the elements of the sets, n_x is the number of nodes in the system, n_ρ is the number of density matrices contained in each node, n_s is the number of scalars contained in each node, $J \in \{(i,j)|i = 0, \ldots, n_x - 1; j = 0, \ldots, n_\rho - 1\}$ and $K \in \{(i,k)|i = 0, \ldots, n_x - 1; k = 0, \ldots, n_s - 1\}$.

The evolution of the system is given by two coupled equations: one for the density matrices and another for the scalar functions. The first one is the Von Neumann equation which contains terms for coherent, decoherence, and other interactions. The second one is a Boltzmann-like differential equation for the scalar functions

$$\frac{\partial \rho_J}{\partial t} = -i[H_J(t), \rho_J] + \{\Gamma_J(t), \rho_J\} + F_J(\tilde{\rho}, \tilde{S}; t), \tag{9}$$

$$\frac{\partial S_K}{\partial t} = -\Gamma_K(t) S_K + G_K(\tilde{\rho}, \tilde{S}; t), \tag{10}$$

where H_J is the Hamiltonian, Γ_J is the decoherence and attenuation term, Γ_K is the attenuation for the scalar functions, and finally the functions F_J and G_K are general functions of the full sets $\tilde{\rho}$ and $\tilde{S}$ that may contain non-linear and interaction terms.

The scalar functions can be used to solve the evolution of populations of particles where there is no coherent quantum dynamics. These parameters can also be used to implement any other first-order differential equation.

3.3.1. Interaction picture evolution

The Hamiltonian can always be decomposed into one part that does not depend on time H_{0J} and another that contains the time dependence, $H_{1J}(t)$

$$H_J(t) = H_{0J} + H_{1J}(t). \tag{11}$$

In a finite Hilbert space the evolution generated by H_{0J} can be solved analytically. This may dramatically speed up the numerical computation, particularly in the case where H_0 contains terms that produce very fast oscillations. This motivates expressing the problem in the interaction picture in which an operator $\hat{O}_J$ is transformed into $\bar{O}_J$ by

$$\bar{O}_J(t) = e^{-iH_{0J}t}\hat{O}_J e^{iH_{0J}t}. \tag{12}$$

Then, the evolution equations Eq. (9) are

$$\frac{\partial \bar{\rho}_J}{\partial t} = -i[\bar{H}_{1J}(t), \bar{\rho}_J] + \{\bar{\Gamma}_J(t), \bar{\rho}_J\} + F'_J(\tilde{\bar{\rho}}, \tilde{S}; t), \tag{13}$$

$$\frac{\partial S_K}{\partial t} = -\Gamma_K(t)S_K + G'_K(\tilde{\bar{\rho}}, \tilde{S}; t), \tag{14}$$

where $\bar{H}_{1J}(t)$ is known as the interaction Hamiltonian, and F'_J and G'_K are the corresponding F_J and G_K in the interaction picture.

4. Density matrix formalism using $SU(N)$ generators

Every ρ_J is a Hermitian matrix therefore has, by construction, real eigenvalues and can always be decomposed as a linear combination of the $SU(N)$ generators plus the identity with real coefficients, being N the dimension of the Hilbert space. The same applies to all the operators in the right hand side of Eq. (9). Using this decomposition we can write the operators as follows,

$$\rho_J = \rho_J^\alpha \lambda_\alpha, \tag{15}$$

$$H_J = H_J^\alpha \lambda_\alpha, \tag{16}$$

$$\Gamma_J = \Gamma_J^\alpha \lambda_\alpha, \tag{17}$$

where $\{\lambda_\alpha | \alpha = 1, \ldots, N^2 - 1\}$ are the $SU(N)$ generators, $\lambda_0 = \mathbb{I}$, and we have used the Einstein convention, i.e. a sum over repeated indices is implicit.

We can use the commutator and anti-commutator relations in the Lie algebra,

$$i[\lambda_\alpha, \lambda_\beta] = -f^\gamma_{\alpha\beta}\lambda_\gamma \tag{18}$$

$$\{\lambda_\alpha, \lambda_\beta\} = d^\gamma_{\alpha\beta}\lambda_\gamma, \tag{19}$$

in order to write down the evolution equations Eqs. (9) and (10)

$$\frac{\partial \rho_J^\alpha}{\partial t} = H_J^\beta(t)\rho_J^\gamma f^\alpha_{\beta\gamma} + \Gamma_J^\beta(t)\rho_J^\gamma d^\alpha_{\beta\gamma} + F_J^\alpha(\tilde{\rho}, \tilde{S}; t), \tag{20}$$

$$\frac{\partial S_K}{\partial t} = -\Gamma_K(t)S_K + G_K(\tilde{\rho}, \tilde{S}; t). \tag{21}$$

4.1. Unitary transformations and zero order time evolution

Since any Hermitian operator $\hat{O}$ can be expressed as linear combination of $\{\lambda_\alpha\}$, then the effect of any linear transformation on $\hat{O}$ only depends on the effect on every element on the basis $\{\lambda_\alpha\}$. In particular, transformations of the form given in Eqs. (6) and (12) fall into this category. For Eq. (6) we can write this explicitly

$$\lambda_\alpha \to U^\dagger(\theta_{ij}\delta_{ij})\lambda_\alpha U(\theta_{ij}\delta_{ij}), \tag{22}$$

given that Eq. (22) preserves hermiticity and that $\{\lambda_\alpha\}$ spans the Hermitian operator space then

$$U^\dagger(\theta_{ij}\delta_{ij})\lambda_\alpha U(\theta_{ij}\delta_{ij}) = u_\alpha^\beta(\theta_{ij}\delta_{ij})\lambda_\beta, \tag{23}$$

where $u_\alpha^\beta(\theta_{ij}\delta_{ij})$ are real functions that are solved analytically in the code.

The same procedure can be applied to the time evolution generated by H_0 given by Eq. (12), i.e.

$$\bar{\lambda}_\alpha(t) = e^{-iH_0t}\lambda_\alpha e^{iH_0t} = u_\alpha^\beta(t)\lambda_\beta, \tag{24}$$

as before the analytic expressions for $u_\alpha^\beta(t)$ are in the code.

4.2. Interaction picture evolution

Give the formalism in Section 4.1 a general Hermitian operator $\hat{O}$ in the interaction picture is given by

$$\bar{O}(t) = e^{-iH_0t}Oe^{iH_0t} = O^\alpha e^{-iH_0t}\lambda_\alpha e^{iH_0t} = O^\alpha \bar{\lambda}_\alpha(t). \tag{25}$$

The commutator and anti-commutator relations holds for the time dependent generators with the same structure constants, which implies that the evolution equations in the interactions picture are the same, but with the $\bar{H}_{1J}(t)$ instead of $H_J(t)$,

$$\frac{\partial \bar{\rho}_J^\alpha}{\partial t} = \bar{H}_{1J}^\beta(t)\bar{\rho}_J^\gamma f^\alpha_{\beta\gamma} + \Gamma_J^\beta(t)\bar{\rho}_J^\gamma d^\alpha_{\beta\gamma} + F'^\alpha_J(\tilde{\bar{\rho}}, \tilde{S}, t), \tag{26}$$

$$\frac{\partial S_K}{\partial t} = -\Gamma_K(t)S_K + G'_K(\tilde{\bar{\rho}}, \tilde{S}, t). \tag{27}$$

5. Description of the code

SQuIDS is a code written in C++ to solve the set of equations specified in the previous section, specifically Eqs. (26) and (27). In this section we will describe the classes, functions, and operations defined between the objects that compose the SQuIDS library. The objects in the library are embedded in the namespace `squids`.

The system structure is shown in Fig. 5.1, each i-node contains a set of density matrices $\{\rho_{(ij)}\}$ and scalar functions $\{S_{(ik)}\}$ as well as a real number x_i that can play the role of a parameter or label for the node. For example, when we are describing a system in which each node represents a given position x then $x_i \equiv x$, on the other hand when we are describing a system in which each state its characterized by is energy or momentum, then $x_i \equiv E$ in each node. Even though x_i can have a physical interpretation, in general it need not to be related to any physical quantity. Furthermore, each node can have an arbitrary number of scalars and density matrices, according to the problem at hand.

All of the classes are declared in the header files contained in the folder `SQuIDS/inc/`, and the corresponding source code that implements them is in `SQuIDS/src/`. In particular, the files that contain the analytic solutions for the evolution and rotation Eqs. (23) and (24) is included in `SQuIDS/inc/SU_inc/` up to dimension six. Finally, `SQuIDS/resources` contains a Mathematica notebook that can be used to generate higher dimension solutions.

5.1. Const

The `Const` class serves two purposes in the `SQuIDS` library. First, it contains transformation of physical units to natural units, which enable the user to work on a consistent unit system, as well as some fundamental constants such as Fermi constant, Avogadro number, proton mass, neutron mass, etc. These constants can be accessed as public members of the class. The full list of constants is listed in Tables 1 and 2. Second, it manages the mixing angles (θ_{ij}) and complex phases (δ_{ij}) that define the transformation between the `B0` and `B1` bases as defined in Eq. (6), which can be modified and obtained through public member functions. Finally, the class is declared in `SQuIDS/inc/const.h` and implemented in `SQuIDS/inc/const.cpp`.

5.1.1. Constructors

- Default constructor

```
Const();
```

  Initializes all units and arrays.
- Move constructor

```
Const(Const&&);
```

  Moves all contents from another `Const` object.

5.1.2. Functions

- Set energy eigenvalue difference

```
void SetEnergyDifference(unsigned int i, double Ediff);
```

  Sets the energy eigenvalue difference $\Delta E_{i0} = E_i - E_0$ to `Ediff`. We require $i > 0$.
- Get energy eigenvalue difference

```
double GetEnergyDifference(unsigned int i) const;
```

  Returns the energy eigenvalue difference ΔE_{i0}.
- Set Mixing angle

```
void SetMixingAngle(unsigned int i,unsigned int j,
                    double theta);
```

  Sets the rotation angle θ_{ij} to `theta`.
- Get Mixing angle

```
double GetMixingAngle(unsigned int i,unsigned int j) const;
```

  Returns the rotation angle θ_{ij}.
- Set complex phase

```
void SetPhase(unsigned int i,unsigned int j, double phase);
```

  Set the complex phase δ_{ij} to `phase`.
- Get complex phase

```
double GetPhase(unsigned int i,unsigned int j) const;
```

  Returns the complex phase δ_{ij}.
- Get transformation matrix

```
std::unique_ptr<gsl_matrix_complex,
                void (*)(gsl_matrix_complex)>
GetTransformationMatrix() const;
```

  Returns the complete matrix representing the transformation between the `B0` and `B1` bases.

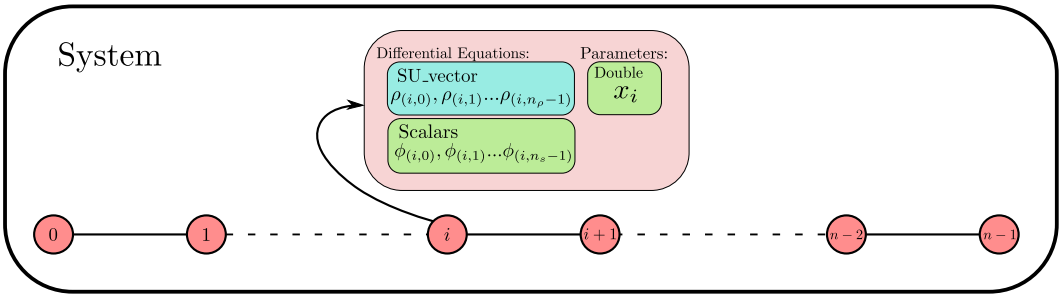

Fig. 5.1. Scheme of the distribution of density matrix and scalar functions on the nodes.

Table 1
Physics constants contained in `Const`.

Physical quantities	
`GF`	Fermi constant in natural units
`Na`	Avogadro number
`sw_sq`	$\sin^2(\theta_w)$ where θ_w is the weak mixing angle
`G`	Gravitational constant
`proton_mass`	Proton mass
`neutron_mass`	Neutron mass
`electron_mass`	e mass
`muon_mass`	μ mass
`muon_lifetime`	μ lifetime
`tau_mass`	τ mass
`tau_lifetime`	τ lifetime
`alpha`	Fine structure constant
`e_charge`	Unit of electric charge (Lorentz–Heaviside)

Table 2
Units contained in `Const`.

Units in natural units ($\hbar = c = k_b = 1$)	
`eV`	1 electron-volt
`keV`	10^3 electron-volt
`MeV`	10^6 electron-volt
`GeV`	10^9 electron-volt
`TeV`	10^{12} electron-volt
`Joule`	1 Joule
`kg`	1 kilogram
`gr`	1 gram
`sec`	1 second
`hour`	1 hour
`day`	1 day
`year`	1 year
`meter`	1 meter
`cm`	1 centimeter
`km`	1 kilometer
`fermi`	1 fermi
`angstrom`	1 angstrom
`AU`	1 astronomical unit
`ly`	1 light year
`parsec`	1 parsec
`picobarn`	1 picobarn
`femtobarn`	1 femtobarn
`Pascal`	1 Pascal
`atm`	1 atmosphere
`Kelvin`	1 Kelvin
`degree`	1 degree in radians
`C`	1 Coulomb (Lorentz–Heaviside)
`A`	1 Ampere (Lorentz–Heaviside)
`T`	1 T (Lorentz–Heaviside)

5.1.3. Operators

- Move assignment operator (=)

```
Const& operator=(Const&&);
```

Moves the contents of another `Const` object to the current object.

5.2. SU_vector

The `SU_vector` class is a type that represents an operator in a N-dimensional Hilbert space as a linear combination of the $SU(N)$ generators and is the building block of the SQuIDS library. The real coefficients of the generator plus the identity linear combination are stored as a private `double` pointer of size N^2. All the right hand side terms of the Von Neumann equation Eq. (26) must be constructed using `SU_vector` objects. In order to do this, the `SU_vector` class implements operations such as: addition, subtraction, scaling by a constant, rotation, time evolution, and traces. Of these the time evolution and rotation are the most computationally expensive. In order to improve the efficiency of the code we have implemented algebraic solutions of Eqs. (23) and (24) that implement rotation and time evolution respectively, and can be found in the files located in `SQuIDS/inc/SU_inc/`.

The headers that declare the class and operations are in the files `SQuIDS/inc/SUNalg.h` and the source code in `SQuIDS/inc/SUNalg.cpp`. Details of the operation implementation can be found in `SQuIDS/inc/detail/` and `SQuIDS/inc/SU_inc/`.

5.2.1. Constructors

Different constructors and initialization functions are added in order to make more flexible the object initialization.

- Default constructor.

```
SU_vector();
```

  Constructs an empty `SU_vector` with no size.
- Copy constructor.

```
SU_vector(const SU_vector& V);
```

  The newly constructed `SU_vector` will allocate its own storage which it will manage automatically.
- Move constructor.

```
SU_vector(SU_vector&& V);
```

  If V owned its own storage, it will be taken by the newly constructed `SU_vector`, and V will be left empty, as if default constructed.
- Constructor of a zero vector in an N-dimensional Hilbert space.

```
SU_vector(unsigned int dim);
```

  This constructor allocates the memory (which will be managed automatically) for the components of the `SU_vector` with size given by the argument and initializes it to zero.
- Constructor with a pointer to double.

```
SU_vector(int dim,double*);
```

  This constructor initializes the `SU_vector` with dimension given by the value of `dim` and uses as the `SU_vector` components the values in the `double*`. The newly constructed `SU_vector` will treat the specified data buffer as its backing storage. The user is responsible for ensuring that this buffer is large enough (at least $\texttt{dim}^2$), and has a lifetime at least as long as the constructed vector and any vectors which inherit the use of this buffer from it by move construction or assignment. The contents of the buffer will not be modified during construction.
- `gsl_matrix_complex` constructor

```
SU_vector(const gsl_matrix_complex*);
```

  Constructs a `SU_vector` from a complex Hermitian GSL matrix. The `gsl_matrix_complex*` contents will copied and not modified.
- Constructor with standard vector.

```
SU_vector(const std::vector<double>& vector);
```

  Constructs `SU_vector` of dimension equal to the square root of size of the `vector` and components given by the contents of the `vector`. The newly constructed `SU_vector` will allocate its own storage, but it will copy its component information from `vector`.

For convenience we also provide factory functions to construct operators that are commonly used such as the identity, projectors, and generators. The following are available:

- Identity.

```
SU_vector Identity(unsigned int dim);
```

  Returns a `SU_vector` which represents the identity of a Hilbert space of dimension `dim`.
- Generator.

```
SU_vector Generator(unsigned int dim, unsigned int i);
```

  Returns a `SU_vector` that represents the `i`th generator (λ_i) of a Hilbert space of dimension `dim`.
- Projector.

```
SU_vector Projector(unsigned int dim, unsigned int i);
```

  Returns a `SU_vector` that represents the projector operator into the subspace spanned by the `i`-state. Namely, $\texttt{Projector} = \text{diag}(0, \ldots, 0, 1, 0, \ldots, 0)$ where the one is in the `i`th entry.

- PosProjector.

```
SU_vector PosProjector(unsigned int dim, unsigned int i);
```

Returns a SU_vector that represents a projector to the upper subspace of dimension i, i.e. PosProjector = diag(1, . . . , 1, 0, . . . , 0) where the last one is in the i-1 entry.

- NegProjector.

```
SU_vector NegProjector(unsigned int dim, unsigned int i);
```

Returns a SU_vector that represents a projector to the lower subspace of dimension i, i.e. NegProjector = diag(0, . . . , 0, 1, . . . , 1) where the first one is in the d-i entry.

5.2.2. Functions

In this section we will describe the general functions that are used to manipulate the objects, access the values and do different operations.

- SetAllComponents.

```
void SetAllComponents(double v);
```

Sets all the components of the SU_vector to v.

- SetBackingStore.

```
void SetBackingStore(double * storage);
```

Sets the external storage used by the SU_vector. If the SU_vector had previously allocated automatically managed storage, that memory will be deallocated; but if it had previously used manually specified storage it will simply cease using that storage without attempting to deallocate it. All data previously stored in the SU_vector is lost after this function is called.

- Rotate function.

```
SU_vector Rotate(unsigned int i,unsigned int j,
                 double theta, double delta) const;
```

Returns the rotated SU_vector by a rotation R_{ij} in the ij-subspace by an angle θ_{ij} (in radians) and complex phase δ_{ij} (in radians) defined in Eq. (7). In order to make it efficient and general uses the analytic solution form Eq. (23) stored in SQuIDS/inc/SU_inc.

- Change of basis.

```
void RotateToB1(const Const& param);
void RotateToB0(const Const& param);
```

This functions uses Rotate to transform the SU_vector from a basis B0 to B1 or vice versa. The mixing matrix that defines the unitary transformation that relate B0 to B1, defined in Eqs. (7) and (8), are given by the parameters in param. In particular, RotateToB0 transform a SU_vector from the B1 basis to the B0 basis, whereas RotateToB1 does the opposite.

- Dimension.

```
unsigned int Dim() const;
```

Returns the dimension of the Hilbert space.

- Size of vector array.

```
unsigned int Size() const;
```

Returns the number of components of the SU_vector.

- Evolution by time independent Hamiltonian.

```
SU_vector Evolve(const SU_vector& h0,double t) const;
```

This function returns a SU_vector after applying the time evolution driven by the operator h0 argument during a time interval t. The analytic solutions of Eq. (24) stored in SQuIDS/inc/SU_inc/ are used. The SU_vector& operator must be a diagonal operator, therefore a linear combinations of the projector in the B0 basis. This choice of the basis makes the evolution more efficient since is not necessary to diagonalize the operator every time.

- Returns the GSL matrix

```
std::unique_ptr<gsl_matrix_complex,
                void (*)(gsl_matrix_complex)>
GetGSLMatrix() const;
```

Returns a unique_ptr that points to a gsl_matrix_complex object. Note that in order to interface with older GSL code one can use the get() member function to obtain a raw pointer.

- Returns the GSL matrix

```
SU_vector UTransform(const SU_vector& transform) const;
```

Returns a SU_vector that corresponds to

$$e^{iT} V e^{-iT} \tag{28}$$

where T is the algebra member represented by transform and V is algebra member that represents the current object.

5.2.3. Operators

Some of the standard C++ operators are overloaded to make more simple and natural writing mathematical expressions. Here we list the operators and how they are defined.

- Logical equality operator (==).

```
bool operator ==(const SU_vector&) const;
```

  Returns `true` or `false` if both `SU_vectors` are equal.
- Scalar product operator (*).

```
double operator*(const SU_vector&) const;
```

  Returns the scalar product of the two vectors, which is equivalent to the trace of the product of the operators that they represent. This operation is useful to compute expectation values, see Eq. (4).
- Product by scalar (*).

```
SU_vector operator*(const double) const;
```

  Returns the `SU_vector` re-scaled by the `double`.
- Assignation operator (=)

  Assigns the value of the `SU_vector` on the right to the one on the left. The dimensions of the `SU_vector` must be the same.
- Sum operator (+).

```
SU_vector operator +(const SU_vector&) const;
```

  Returns the sum of two `SU_vector` objects.
- Subtraction operator (-).

```
SU_vector operator -(const SU_vector&) const;
```

  Returns the subtraction of two `SU_vector` objects.
- Negation operator (-).

```
SU_vector operator -() const;
```

  Returns the additive inverse of an `SU_vector` object.
- Assignment and move assignment operators (=)

```
SU_vector & operator=(const SU_vector&);
SU_vector & operator=(SU_vector&&);
```

  The assignment operator as well as the move assignment operator are defined. In particular, for the move operator, if the vector is empty or owns its own storage it will switch to using whatever storage was used by other, causing other to relinquish any ownership of that storage. If, however, the vector is non-empty and uses external storage, it will copy other's data rather than shifting its storage. In this case if the dimensions of the two vectors differ the assignment will fail.
- Assignment addition and subtraction operators (-= +=)

```
SU_vector & operator+=(const SU_vector&);
SU_vector & operator-=(const SU_vector&);
```

  These operations combine the addition and subtraction with the assignment as usual in C++.
- Assignment multiplication and division operators (*= /=)

```
SU_vector & operator*=(double);
SU_vector & operator/=(double);
```

  These operations combine the multiplication and division by scalars with the `SU_vector` assignment operation.
- Array like component access.

```
double& operator[](int);
const double& operator[](int) const;
```

  Returns the component given by the `int` argument.
- Ostream operator (<<)

```
friend ostream& operator<<(ostream&, const SU_vector&);
```

  Writes the components of the `SU_vector` into the `ostream` object as human-readable text.

5.2.4. External functions

We have also defined quantum mechanical operations between two SU_vector. Furthermore, to optimize the code we have implemented the iCommutator and Anticommutator analytically and stored them in SQuIDS/inc/SU_inc.

- iCommutator.

```
SU_vector iCommutator(const SU_vector&,const SU_vector&);
```

  Returns the SU_vector result of i times the commutator of the SU_vector objects given as an argument Eq. (18).
- Anticommutator.

```
SU_vector ACommutator(const SU_vector&,const SU_vector&);
```

  Returns the SU_vector result of the anti-commutator of the SU_vectors objects given as an argument Eq. (19).
- Trace function.

```
double SUTrace(const SU_vector&, const SU_vector&);
```

  Returns the trace of the product of the two operators represented by the SU_vectors given in the arguments. It is the same as the scalar product.

5.2.5. Usage and optimization

A SU_vector of dimension N has N^2 components that are stored as a private double*. By default SU_vector will automatically allocate sufficient 'backing storage' to contain these. It is, however, possible to specify that an SU_vector should treat some externally provided buffer as its backing storage. In this case the size and lifetime of that buffer are the responsibility of the user, so users are encouraged to avoid using this mode unless it is required by their application, as its use is more difficult and requires much greater care. The external storage mode is primarily useful because it allows interfacing with other, low-level numerical codes efficiently.

As described in the previous sections SU_vector provides overloaded mathematical operators so that algebra can be written in a natural way. Furthermore, the SQuIDS library has a limited ability to optimize away temporary objects (via a partial expression template system). That is, all operations of the forms

```
v1 [Op1]= v2 [Op2] v3;
v1 [Op1]= s * v2;
```

where v1, v2, and v3 are pre-existing objects of type SU_vector (and s is a scalar) are performed without allocating memory. Op1 may +,-, or nothing (normal assignment), and Op2 may be +, -, time evolution, a commutator or an anticommutator.

This optimization is inhibited when v1 aliases v2 or v3 (they are the same objects or they otherwise refer to the same backing storage) and the operation being performed involves components in the input and output vectors with different indices. This has no influence on the correctness of writing complex expressions in terms of subexpressions: These will still be correctly evaluated, but memory will be allocated for the results of the subexpressions, making the calculation slower than if this can be avoided. It is expected that users will write expressions in the form they find most natural, and only if performance optimization is required consider restructuring code to take deliberate advantage of this optimization. In that case, the following techniques may be useful:

- If a calculation involving subexpressions is performed in a loop, it is advantageous to manually create sufficient temporaries for all subexpressions outside of the loop and then split the complex expression into a series of basic operations whose results are stored to the temporaries. This ensures that allocation will be performed only once per temporary before entering the loop, rather than once per temporary, per loop iteration. For example:

```
1 //assuming size N arrays of SU_vector state, v1, v2, v3, and v4
2 //and a floating-point t
3 for(unsigned int i=0; i<N; i++)
4     state[i] += v1[i].Evolve(v2[i],t)
5               + v3[i].Evolve(v4[i],t);
```

  In this code the addition on the right hand side can be performed without allocation, but each of the evolution operations must allocate a temporary, so 2*N allocations and deallocations must occur. This can be reduced to 2 allocations and deallocations by rewriting in this form:

```
1 SU_vector temp1, temp2;
2 for(unsigned int i=0; i<N; i++){
3   temp1 = v1[i].Evolve(v2[i],t);
4   temp2 = v3[i].Evolve(v4[i],t)
5   state[i] += temp1 + temp2;
6 }
```

- If a calculation has an SU_vector calculation as a subexpression, but otherwise operates on scalars, it can be useful to rewrite the expression so that the vector calculation forms the top level if possible:

```
1 //assuming SU_vectors v1 and v2 and scalars s1 and s2
2 v1 = s1*(s2*v2);
3 //can be better reassociated as:
4 v1 = (s1*s2)*v2;
```

Table 3
Arguments description of SQuIDS constructor and initialization function from left to right.

Argument	Type	Description
`nx`	`unsigned int`	Number of nodes in the problem, each node has a set of density matrices and scalars as defined by the other arguments.
`dim`	`unsigned int`	Dimension of the Hilbert space of the density matrices.
`nrho`	`unsigned int`	Number of density matrices in each node.
`nscalar`	`unsigned int`	Number of scalars functions in each node.
`ti`	`double`	Initial time of the system. (Defaults to zero.)

Table 4
Arguments of the `Set_xrange` function.

Argument	Type	Description
`xini`	`double`	Smaller value of `x`.
`xend`	`double`	Largest value of `x`.
`scale`	`string`	Either "`lin`" or "`log`" and sets the scale as a linear or logarithmic.

5.3. SQuIDS

This object implements the numerical solution for a set of density matrices plus a set of scalar functions in the interaction picture, where the evolution given by H_0 is solved analytically. The numerical calculation is done using the GNU Scientific Library, and different parameters for the numerical precision and integrator can be set through the SQuIDS interface.

As we described before in Section 3.3 the system consists of a set of nodes where every i-node contains n_ρ density matrix expressed as `SU_vector`, n_s scalar functions which are `double`, and a `double` parameter x_i. The scheme is shown in Fig. 5.1.

The object is defined so that all of the terms in the right hand side of the differential equation are defined as virtual functions that by default return zero, and can be overridden by the user.

If the user does not activate the terms that are numerically solved, then the evolution will be done analytically.

5.3.1. Constructors and initializing functions

- Default constructor.

```
SQuIDS();
```

Constructs an uninitialized SQuIDS object.

- Constructor and initializing function.

```
SQuIDS(unsigned int nx,unsigned int dim,unsigned int nrho,
       unsigned int nscalar, double ti = 0);
void ini(unsigned int nx,unsigned int dim,unsigned int nrho,
       unsigned int nscalar, double ti = 0);
```

Initializes the object, allocating all necessary memory for the density matrices and scalars. The arguments are described in Table 3.

- Move constructor

```
SQuIDS(SQuIDS&&);
```

Moves all contents from another SQuIDS object.

5.3.2. Functions

- Set the range for the array `x`.

```
int Set_xrange(double xini, double xend, string scale);
```

This function sets the values on the array `x`. The array will start at `xini` and end at `xend` (inclusive) with either a uniform linear or logarithmic spacing (see Table 4).

- Set the range for the array `x`.

```
void Set_xrange(const std::vector<double>& xs);
```

This function sets the values on the array `x` to be the contents of `xs` and checks that the number of set nodes is equal to the size of the vector.

- Get value of `x`

```
double Get_x(unsigned int i) const;
```

Returns the `i`th value of the `x` array.

- Get the bin in `a`

```
int Get_i(double x) const;
```

Returns the index of the `x` array whose value is closest to `a`. `a` must be between the values previously specified for `xini` and `xend` in the most recent call to `Set_xrange`.

Table 5
Set functions that control the use of virtual functions.

Set function	Description
`Set_CoherentRhoTerms`	Activate the use of `HI`
`Set_NonCoherentRhoTerms`	Activate the use of `GammaRho`
`Set_OtherRhoTerms`	Activate the use of `InteractionsRho`
`Set_GammaScalarTerms`	Activate the use of `GammaScalar`
`Set_OtherScalarTerms`	Activate the use of `InteractionsScalar`

- Get current system time

```
double Get_t() const;
```

Returns the current time of the system.

- Get `params` object

```
double Get_t_initial() const;
```

Returns the initial time of the system.

- Get initial time

```
const * Const Get_params() const;
```

Returns a const reference to the `params` SQuIDS protected member.

- Get number of nodes

```
unsigned int Get_nx() const;
```

Returns the number of nodes in the system.

- Get number of density matrices

```
unsigned int Get_nrhos() const;
```

Returns the number of density matrices per nodes.

- Get number of scalar functions

```
unsigned int Get_nscalars() const;
```

Returns the number of scalar functions per node.

- Derivative.

```
void Derive(double t);
```

Computes the derivative, r.h.s. of Eqs. (26) and (27), of the system at a time `t`, including all of the terms defined by the user by overriding the virtual functions specified in Section 5.3.3. Note that for each user-supplied interaction term the appropriate flag must be set `true` for that term to be included (see Section 5.3.4 and Table 5).

- Evolution function.

```
void Evolve(double dt);
```

Evolves the system by a time interval `dt` given in natural units[1] (see `Const` class in Section 5.1 and, in particular, Table 2).

- Get expectation value.

```
double GetExpectationValue(SU_vector op,unsigned int irho,
                           unsigned int ix) const;
double GetExpectationValueD(SU_vector op,unsigned int irho,
                           double x) const;
```

The first function returns the expectation value of the operator represented by `op` for the state in the `irho` density matrix in that `ix` node at the current time `t`. Notice that `op` is evolved by the $H_{0(ix,irho)}$ Hamiltonian in order to go the interaction picture at time `t`.

In general H_{0l} may depend continuously in the parameter x, in that case in order to compute the expectation value it is very useful perform the evolution of the operator `op` driven by H_{0l} for the exact value of x. The second function uses this method together with a linear interpolation in ρ over the parameter x in order to give the expectation value.

[1] We set $c = \hbar = k_b = 1$.

5.3.3. Virtual functions

All of these virtual functions return a zero `SU_vector` by default and may be overridden by the user in a derived class to define a problem. When implemented, each function's return value must be in natural units (see Table 2).

- Time independent Hamiltonian H_{0J}.

```
virtual SU_vector H0(double x,unsigned int irho) const;
```

  This function returns the time independent Hamiltonian that will be solved analytically, for a particular value of the parameter `x` and density matrix `irho`. It is important to note that the analytic solution is implemented assuming that this operator is represented by a diagonal matrix; therefore the problem basis should be chosen to satisfy this condition. For example, in the case of neutrino oscillations this means that the operators are defined in the mass basis.

  The `double x` gives the parameter corresponding to the node of the `x` array. Notice that H_{0J} must be defined as a continuous function of `x` since H_{0J} is solved analytically and thus its independent of the discrete nodes, as this allows `GetExpectationValueD` to calculate observables at arbitrary `x`.

- Time dependent Hamiltonian H_{1J}.

```
virtual SU_vector HI(unsigned int ix,unsigned int irho,
                     double t) const;
```

  Returns the time dependent part of the Hamiltonian at the `ix` node for the density matrix `irho` at time `t`. The result of this function is used to calculate the commutator that drives the quantum evolution, the first term in Eq. (26).

  This function is used only if `Set_CoherentRhoTerms` has been called with a `true` argument.

- Non coherent terms Γ_J.

```
virtual SU_vector GammaRho(unsigned int ix,
                           unsigned int irho, double t) const;
```

  Returns the non-coherent interaction at the `ix` node for the density matrix `irho` at time `t`. The result of this function is used to calculate the anticommutator, the second term in Eq. (26).

  This function is used only if `Set_NonCoherentRhoTerms` has been called with a `true` argument.

- Other ρ interactions F_J.

```
virtual SU_vector InteractionsRho(unsigned int ix,
                           unsigned int irho,double t) const;
```

  Returns the third term on the right hand side of Eq. (26) at the `ix` node for the density matrix `irho` at time `t`. For example, terms mixing different density matrices and scalar functions of different nodes can be included here.

  This function is used only if `Set_OtherRhoTerms` has been called with a `true` argument.

- Scalar function attenuation Γ_K.

```
virtual double GammaScalar(unsigned int ix,
                           unsigned int iscalar, double t) const;
```

  It returns the Boltzmann attenuation term (first term) for the scalar function Eq. (26) at the `ix` node for the scalar function `iscalar` at time `t`.

  This function is used only if `Set_GammaScalarTerms` has been called with a `true` argument.

- Other scalar interactions G_K.

```
virtual double InteractionsScalar(unsigned int ix,
                           unsigned int iscalar,double t) const;
```

  Returns any necessary second term on the right hand side of Eq. (26) at the `ix` node for the scalar function `iscalar` at time `t`. This may include terms that depend on the other scalars and density matrices.

  This function is used only if `Set_OtherScalarTerms` has been called with a `true` argument.

- Pre-Derivative Function.

```
virtual void PreDerive(double t);
```

  This function is called every time before computing the derivatives at time `t`, i.e. before evaluating the virtual functions described above, and can be used to pre-calculate variables that will be used in the derivative or time evolve projectors in order to more easily and efficiently compute the preceding functions. In general, any update of the parameters in the preceding functions can be included. For example, in the case of neutrino oscillations, this can include the evolution of the flavor projectors that can be then used to define the H_{1J} Hamiltonian.

Table 6
Available GSL stepper functions.

Available GSL stepper functions
gsl_odeiv2_step_rk2
gsl_odeiv2_step_rk4
gsl_odeiv2_step_rkf45
gsl_odeiv2_step_rkck
gsl_odeiv2_step_rk8pd
gsl_odeiv2_step_msadams

5.3.4. Set functions

The set functions configure different parameters in the SQuIDS object.

- Set GSL stepper function.

```
void Set_GSL_step(gsl_odeiv2_step_type const * opt);
```

Sets the GSL stepper function for the differential numerical algorithm, see Table 6. Note that this is a subset of the methods supported by GSL and only includes those that do not require second derivative. By default gsl_odeiv2_step_rkf45 is set. For more details see the GSL website [5].

- Set absolute error

```
void Set_abs_error(double error);
```

Sets the GSL algorithm absolute error to error.

- Set relative error

```
void Set_rel_error(double error);
```

Sets the GSL algorithm relative error to error.

- Set initial step

```
void Set_h(double h);
```

Sets the GSL algorithm initial step to h.

- Set minimum step

```
void Set_h_min(double h);
```

Sets the GSL algorithm minimum step to h.

- Set maximum step

```
void Set_h_max(double h);
```

Sets the GSL algorithm maximum step to h.

- Switch adaptive stepping

```
void Set_AdaptiveStep(bool opt);
```

If opt is true adaptive stepping will be used, otherwise a fixed step size will be used. In the fixed step case the number of steps can be set by Set_NumSteps.

- Set number of steps (for fixed stepping)

```
void Set_NumSteps(int steps);
```

Sets the number of steps used when not using adaptive stepping.

- Switch Coherent Interactions

```
void Set_CoherentRhoTerms(bool opt);
```

If opt is true the implemented HI will be used, otherwise it will be ignored and treated as zero.

- Switch Non Coherent Interactions

```
void Set_NonCoherentRhoTerms(bool opt);
```

If opt is true the implemented GammaRho will be used, otherwise it will be ignored and treated as zero.

- Switch extra ρ Interactions

```
void Set_OtherRhoTerms(bool opt);
```

If opt is true the implemented InteractionsRho will be used, otherwise it will be ignored and treated as zero.

- Switch Scalar Attenuation.

```
void Set_GammaScalarTerms(bool opt);
```

If opt is true the implemented GammaScalar will be used, otherwise it will be ignored and treated as zero.

- Switch Scalar Interactions

```
void Set_OtherScalarTerms(bool opt);
```

If opt is true the implemented InteractionsScalar will be used, otherwise it will be ignored and treated as zero.

5.3.5. Protected members

In this section we describe some useful protected member of the SQuIDS derive which derive classes may want to use.

- `Const` object

```
Const params;
```

`Const` object that contains useful units and angles that define the transformation between basis.

- State of a node

```
struct SU_state {
    std::unique_ptr<SU_vector[]> rho;
    double* scalar;
};
```

Contains the state of a node consisting of density matrices contained in `rho` and scalar functions in `scalar`.

- System state

```
std::unique_ptr<SU_state[]> state;
```

Contains the state of the system at the current time, namely the density matrices and scalars at all nodes.

- Number of nodes

```
unsigned int nx;
```

Number of nodes in the system.

- Number of scalars

```
unsigned int nscalars;
```

Number of scalar functions per node.

- Number of density matrices.

```
unsigned int nrhos;
```

Number of density matrices per node.

- Hilbert space dimension

```
unsigned int nsun;
```

Dimension of the Hilbert space of the density matrices.

5.3.6. Protected functions

- Set the current `SQuIDS` time

```
void Set_t(double t);
```

Sets the current time to `t`. The only valid argument of this function is the current time. It should only be used by experienced users.

6. Performance details

Solving couple quantum integro differential equations is not a trivial numerical problem. In this section, we compare different ODE integrators in a simple case scenario, as well as the performance of compilers in optimizing our code.

6.1. GSL integrator comparison

Most of the numerical problems consist of solving the differential equations, for which we use the GSL library, which provides a variety of methods whose speed and precision might differ substantially for different problems. In order to compare the different GSL integration routines we test a simple two-level quantum mechanical system with states labeled $|A\rangle$ and $|B\rangle$ as a benchmark, similar to the example in Section 7.2. In this scenario, the evolution of the quantum system has to be unitary and the expected value of the system in each state has to be between 0 and 1; thus we define the error as deviations from unitarity and valid probabilities, i.e.

$$\epsilon(t) = \left|1 - |\langle A|\ \psi(t)\rangle|^2 - |\langle B|\ \psi(t)\rangle|^2\right| + f(|\langle A|\ \psi(t)\rangle|^2) + f(|\langle B|\ \psi(t)\rangle|^2), \tag{29}$$

where $|\psi(t)\rangle$ is the state at time t and we have introduced

$$f(x) = \begin{cases} -x & \text{if } x < 0, \\ 0 & \text{if } 0 \leq x \leq 1, \\ x - 1 & \text{if } x > 1. \end{cases} \tag{30}$$

We have evolved the system for some number of steps accumulating the error at each step and repeated the exercise for each available GSL stepper function, listed in Table 6. The result can be seen in Fig. 6.1 where the total execution time in seconds is given in parentheses in the plot label. It should be noted that different algorithms provide significantly different trade-offs between speed and correctness. The precision of results using this library depends on the details of the ODE integration, so users will want to choose the stepper function carefully depending on the problem at hand.

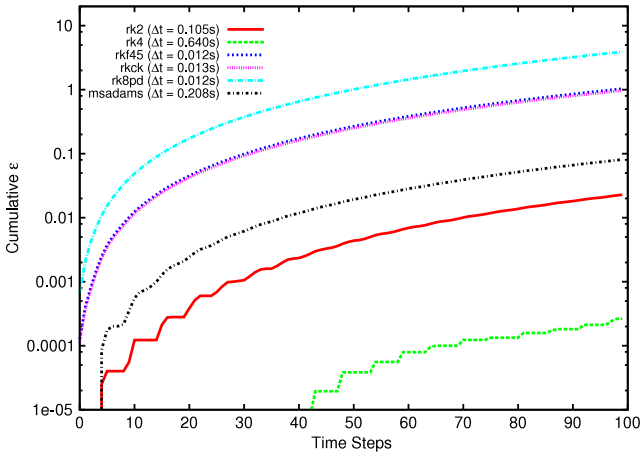

Fig. 6.1. Cumulative error for a simple two-level system using different GSL stepper implementations. Total execution time is shown in parentheses in the legend.

Table 7
Relative times for each compiler on the three examples given with the code, normalized to the fastest.

Compiler	Rabi	Vacuum	Collective neutrinos
gcc 4.8	1.2	1	1
clang 3.6	1.36	1.05	1.14
icc 15.0	1	1.05	1.29

6.2. Compiler comparison

Currently three generally available compilers support the C++11 features used in this code, namely, GCC [15], Clang [16], and ICC [17]. We performed a test of the speed of the code generated from this library by these compilers. This consisted of compiling and running the three examples described in Section 7. The median execution time for each compiler with respect to the fastest compiler is reported in Table 7. No drastic differences are observed, but it is clear that some compiler perform better than other in different tasks. Thus no compiler appears to be strongly favored over the others.

7. Included examples

7.1. Vacuum neutrino oscillations

This is a basic example in which vacuum neutrino oscillations are implemented. In this case, the code does not use any numerical integration, just the analytic solutions given by the `SU_vector` class.

The evolution is defined by H_0, which in the mass basis has the following form

$$H_0 = \frac{1}{2E}\begin{pmatrix} 0 & 0 & 0 \\ 0 & \Delta m^2_{21} & 0 \\ 0 & 0 & \Delta m^2_{31} \end{pmatrix}, \tag{31}$$

where the mixing matrix equation (6), which relates the mass and flavor bases, depends on three mixing angles and one complex phase: $\{\theta_{12}, \theta_{23}, \theta_{13}, \delta\}$.

In the following, we describe the derived class and the functions that implement this example.

7.1.1. Derived class (`vacuum`)

This class is defined in `SQuIDS/examples/VacuumNeutrinoOscillations/vacuum.h` and implemented in `SQuIDS/examples/VacuumNeutrinoOscillations/vacuum.cpp`.

The `vacuum` class constructor has the following signature

```
vacuum(unsigned int nbins,unsigned int nflavor,
       double Eini, double Efin);
```

where `nbins` is the number of energy bins in a logarithm scale from a minimum `Eini` to a maximum `Efin` and `nflavor` is the number of neutrino flavors. On the constructor it calls the SQuIDS `ini` function in the following way

```
ini(nbins,nflavor,1,0,0.);
```

which initializes `nbins` x nodes, which refer to the neutrino energy, with one `SU_vector` of dimension `nflavor` and no scalar functions. The final parameter sets the initial time of the system to zero. Furthermore, in the constructor, it is useful to define the projectors in the flavor and mass basis given by Eq. (3), since H_0 is a linear combination of the mass projectors and the flavor projectors are needed to

evaluate flavor expectation values. The projectors are stored in the following arrays

```
std::unique_ptr<squids::SU_vector[]> b0_proj;
std::unique_ptr<squids::SU_vector[]> b1_proj;
```

where the b0 label corresponds to the mass basis and the b1 to the flavor basis. In order to define the transformation between the mass and flavor basis we use `params.Set_MixingAngle`.

```
params.SetMixingAngle(0,1,33.48*params.degree); //theta 1,2
params.SetMixingAngle(0,2,8.55*params.degree); //theta 1,3
params.SetMixingAngle(1,2,42.3*params.degree); //theta 2,3
```

Next we construct the `SU_vector DM2` that represent the matrix in Eq. (31).

```
const double ev2=params.eV*params.eV;
params.SetEnergyDifference(1,7.5e-5*ev2); //delta m^2 2,1
params.SetEnergyDifference(2,2.45e-3*ev2); //delta m^2 3,1

for(int i = 1; i < nsun; i++)
      DM2 += (b0_proj[i])*params.GetEnergyDifference(i);
```

Finally, we set the initial state of system to the first flavor (ν_e) by means of the flavor projectors

```
for(int ei = 0; ei < nx; ei++)
  state[ei].rho[0]=b1_proj[0];
```

The next member function is

```
SU_vector H0(double E, unsigned int irho) const;
```

which returns the value of the time independent Hamiltonian H_0. The last member function is defined to get the flux of a given flavor

```
double Get_flux(unsigned int flavor,double enu);
```

where `flavor` specifies the neutrino flavor and `enu` the neutrino energy.

7.1.2. Main file

The main file declares the object and propagates the three standard neutrino states in a 1000 km baseline. The final flavor content is saved in the file `oscillations.dat`. The output is shown in Fig. 7.1.

```
1  int main(){
2    squids::Const units;
3
4    //Number of energy bins
5    unsigned int Nenergy=1000;
6    //Number of flavors
7    unsigned int Nflavor=3;
8    //Energy Range
9    double Emin=10*units.MeV, Emax=10*units.GeV;
10   //declaration of the object
11   vacuum V0(Nenergy,Nflavor,Emin,Emax);
12
13   V0.Evolve(1000*units.km);
14
15   std::ofstream file("oscillations.dat");
16
17   const int nu_e=0, nu_mu=1, nu_tau=2;
18   for(double lE=log(Emin); lE<log(Emax); lE+=0.0001){
19     double E=exp(lE);
20     file << E/units.GeV << "␣␣" << V0.Get_flux(nu_e,E) << "␣␣" <<
21       V0.Get_flux(nu_mu,E) << "␣␣" << V0.Get_flux(nu_tau,E) << std::endl;
22   }
23
24   std::cout << std::endl <<  "Done!␣" << std::endl <<  "Do␣you␣want␣to␣run␣the␣gnuplot␣script
       ?␣yes/no" << std::endl;
25   std::string plt;
26   std::cin >> plt;
27
28   if(plt=="yes" || plt=="y"){
29     return system("./plot.plt");
30   }
31
32   return 0;
33 }
```

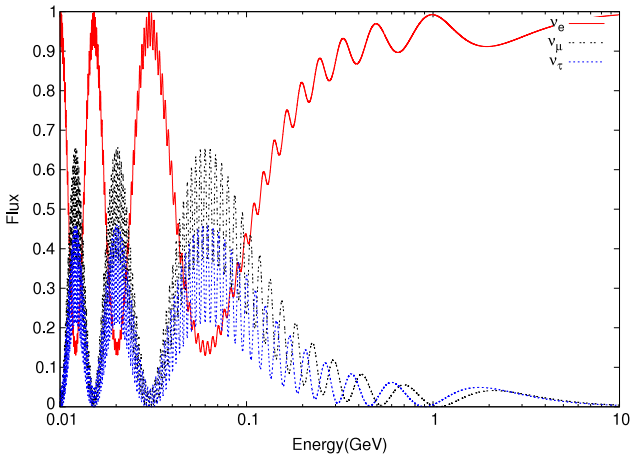

Fig. 7.1. Probability for a neutrino to interact as a particular flavor as a function of energy after propagating 1000 km, starting from a pure ν_e flux.

7.2. Rabi oscillations

This example illustrates the numerical time dependent Hamiltonian solution. The code solves for the population of a two level system as a function of time under the influence of an external oscillating potential (e.g. a laser).

We consider two cases: in the first the frequency of the laser is resonant with the energy difference of the two level system, and in the second the laser has a small de-tuning.

The evolution of the Rabi system is driven by the Hamiltonian

$$H(t) = H_0 + H_1(t) = \begin{pmatrix} \epsilon_1 & 0 \\ 0 & \epsilon_2 \end{pmatrix} + \begin{pmatrix} 0 & d \\ d & 0 \end{pmatrix} \xi(t), \tag{32}$$

where ϵ_i is the energy of the i-state, d is the dipole expected value, and the function $\xi(t)$ plays the role of an external laser acting on the system, which is given by

$$\xi(t) = A\cos(\omega t). \tag{33}$$

In terms of the H_0 eigenstates projectors, Π_i, the full Hamiltonian has the form

$$H(t) = \Pi_2(\epsilon_2 - \epsilon_1) + (U^\dagger(\pi/4)\Pi_1 U(\pi/4) - U^\dagger(\pi/4)\Pi_2 U(\pi/4))\xi(t), \tag{34}$$

where the dipole operator is constructed by a linear combination of rotated projectors.

7.2.1. Derived class (`rabi`)

The class is declared in `SQuIDS/examples/RabiOscillations/rabi.h` and implemented in `SQuIDS/examples/RabiOscillations/rabi.cpp`.

As in Section 7.1 the projectors are included, where `b0` and `b1` label the projectors to the H_0 and dipole eigenstates respectively.

```
std::unique_ptr<squids::SU_vector[]> b0_proj;
std::unique_ptr<squids::SU_vector[]> b1_proj;
```

It is also useful to define

```
squids::SU_vector suH0;
squids::SU_vector d0;
squids::SU_vector d;
```

where `d0` is the dipole operator, `d` is the time evolved dipole operator, and `suH0` is the time independent Hamiltonian.

The initialization and constructor depend on the energy difference of the two levels, `D_E`, the frequency of the external potential, `w_i`, and the amplitude of the external interaction, `Am`.

```
rabi(double D_E, double wi, double Am);
void init(double D_E, double wi, double Am);
```

First we initialize the base SQuIDS object for this problem

```
ini(1/*nodes*/,2/*SU(2)*/,1/*density matrices*/,0/*scalars*/);
```

Next, we set the physical parameters of the problem: We use `SetEnergyDifference` to set the levels' energy difference and `SetMixingAngle` to construct the transformation between the two bases, which is U in Eq. (34).

```
params.SetEnergyDifference(1,D_E);
params.SetMixingAngle(0,1,params.pi/4);
```

We turn on the treatment of coherent terms (`HI`):

```
Set_CoherentRhoTerms(true);
```

We construct the time independent Hamiltonian using energy difference, the first term in Eq. (34):

```
suH0 = b0_proj[1]*params.GetEnergyDifference(1);
```

We construct the dipole operators: `d0` is the initial dipole operator and `d` the time evolved one, which are defined initially to be

```
d0=(b1_proj[0]-b1_proj[1]);
d=d0;
```

Finally, the system is initialized to the ground state.

We also defined the `PreDerive` function

```
void rabi::PreDerive(double t){
  d=d0.Evolve(suH0,t-Get_t_initial());
}
```

which evolves the dipole operator to the necessary time. In this problem, `H0` is just defined to return the constant `SU_vector suH0` which was computed at initialization. Finally, `HI` is returns the second term in Eq. (34)

```
squids::SU_vector rabi::HI(unsigned int ix,
                           unsigned int irho, double t) const{
  return (A*cos(w*t))*d;
}
```

7.2.2. Main file

The main file declares two `rabi` objects `R0` and `R1`, the first with two matching frequencies and the second with two frequencies de-tuned by the value given by the user.

```
int main(){
  // Declaration of the objects
  rabi R0,Rd;
  // de-tuning
  double del;

  // delta time for the prints
  double dt=0.01;
  // Final time
  double tf=120;

  // Tuned Rabi system
  R0.init(10,10,0.1);

  // Setting the errors
  R0.Set_rel_error(1e-5);
  R0.Set_abs_error(1e-5);

  std::cout << "Rabi␣system␣with␣frequency␣of␣10␣initialized." << std::endl;
  std::cout << "give␣the␣value␣for␣the␣detuning:␣" << std::endl;
  std::cin >> del;

  // un-tuned Rabi system
  Rd.init(10,10+del,0.1);
  // Setting the errors
  Rd.Set_rel_error(1e-5);
  Rd.Set_abs_error(1e-5);

  std::cout << "Computing␣rabi" << std::endl;
  std::ofstream file("rabi.dat");

  // Evolve and save the evolution
  for(double t=0;t<tf;t+=dt){
    progressbar(100*t/tf);
    R0.Evolve(dt);
    file << t << "\t" << R0.GetExpectationValue(R0.d0,0,0) << "␣␣"
         << R0.GetExpectationValue(R0.b0_proj[0],0,0) << "␣␣"
         << R0.GetExpectationValue(R0.b0_proj[1],0,0) << std::endl;
  }
  file.close();
  file.open("rabi_detuned.dat");
  std::cout << std::endl << "Computing␣detuned␣rabi" << std::endl;
```

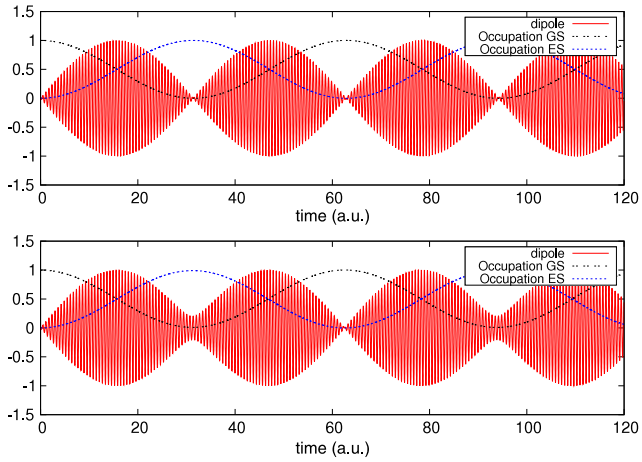

Fig. 7.2. Time dependence of the expected value of the dipole and occupation number for the ground-state (GS) and excited state (ES) for exact tuning (upper panel) and a system with a frequency de-tuned by 0.01 (lower panel).

```
43    for(double t=0;t<tf;t+=dt){
44      progressbar(100*t/tf);
45      Rd.Evolve(dt);
46      file << t << "\t" << Rd.GetExpectationValue(Rd.d0,0,0) << "␣␣"
47           << Rd.GetExpectationValue(Rd.b0_proj[0],0,0) << "␣␣"
48           << Rd.GetExpectationValue(Rd.b0_proj[1],0,0) << std::endl;
49    }
50    file.close();
51
52    //Ask whether to run the gnuplot script
53    std::string plt;
54    std::cout << std::endl <<  "Done!␣" << std::endl <<
55     "␣␣Do␣you␣want␣to␣run␣the␣gnuplot␣script?␣yes/no" << std::endl;
56    std::cin >> plt;
57    if(plt=="yes" || plt=="y")
58      return system("./plot.plt");
59    return 0;
60  }
```

As before a gnuplot script is added in order to plot the result Fig. 7.2. This result is comparable to the one obtained in [18].

7.3. Collective neutrino oscillation

In this example we implement a simple version of the collective neutrino oscillation phenomena. We will follow the notation given in [19].

For a two level system and in the absence of non-coherent interactions, the operators can be interpreted as a vectors in a three dimensional space, in particular the state of the system is

$$\rho_w = P_w^i \lambda_i, \tag{35}$$

and the time independent Hamiltonian is

$$H_0 = B_w^i \lambda_i \tag{36}$$

where $i \in \{1, 2, 3\}$ and $w = T/E$, with T being the temperature and E the energy of the state.

The evolution of the system is given by the density matrix equation

$$\dot{\rho}_w = i[H_0 + \mu\rho, \rho_w], \tag{37}$$

where $\rho = \int \rho_w dw$ and $\mu = \sqrt{2} G_F n_\nu$.

In order to get a geometrical description is useful to use the vector notation: In the vector notation the commutator of the operators is equivalent to the cross product of the vectors. Using this the equation for a system with self interacting terms is given by

$$\dot{P}_w = (wB + \mu P) \times P_w, \tag{38}$$

where

$$P = \int P_w dw, \tag{39}$$

and $\mu = \sqrt{2} G_F n_\nu$ is the self-interaction strength, and B is the vacuum Hamiltonian.

The problem we solve is the evolution of the density matrix for the case where the parameter μ is varying until it reaches 0 at some time T. As the initial condition we set all the vectors P_w aligned together, with a small angle θ from B.

7.3.1. Derived object (`collective`)

The object is declared in SQuIDS/examples/CollectiveNeutrinoOscillations/collective.h and implemented in SQuIDS/examples/CollectiveNeutrinoOscillations/collective.cpp.

In this example, working in the interaction picture does not bring any advantage for the case of having a large number of w nodes. Therefore, we do not need to define the evolved projectors. Instead, and following the analogy of the vectors in three dimensional space, we define the $SU(2)$ generators, which are equivalent the unit vectors in the three perpendicular directions.

```
SU_vector ex,ey,ez;
ex=SU_vector::Generator(nsun,1);
ey=SU_vector::Generator(nsun,2);
ez=SU_vector::Generator(nsun,3);
```

The constructor sets up the value of μ (`mu`), the angle between B and P (`th`), the range for w (`wmin` to `wmax`) and the number of w-bins (`Nbins`).

```
collective(double mu,double th, double wmin, double wmax, int Nbins);
void init(double mu,double th, double wmin, double wmax, int Nbins);
```

Next we define

```
void collective::PreDerive(double t){
  if(bar)
    progressbar(100*t/period, mu);
  //compute the sum of 'polarizations' of all nodes
  P=state[0].rho[0];
  for(int ei = 1; ei < nx; ei++)
    P+=state[ei].rho[0];
  //update the strength of self-interactions
  mu = mu_f+(mu_i-mu_f)*(1.0-t/period);
}
```

in which we implement Eq. (39) and update the value of μ.

We write `HI` to implement Eq. (38):

```
SU_vector collective::HI(unsigned int ix,
                         unsigned int irho, double t) const{
  //the following is equivalent to
  //return Get_x(ix)*B+P*(mu*(w_max-w_min)/(double)nx);

  //make temporary vectors which use the preallocated buffers
  SU_vector t1(nsun,buf1.get());
  SU_vector t2(nsun,buf2.get());
  //evaluate the subexpressions into the temporaries
  t1=Get_x(ix)*B;
  t2=P*(mu*(w_max-w_min)/(double)nx);
  //return the sum of the temporaries, noting that t1 is no
  //longer needed so its storage may be overwritten
  return(std::move(t1)+t2);
}
```

Note that in the commented line we directly write Eq. (38), but for efficiency reasons we write equivalent code in a somewhat more verbose manner. Instead of using unnamed temporary vectors we create temporary vectors using preallocated memory buffers. This avoids the need to allocate memory on each call to `HI`, although it comes at the cost that the function is neither reentrant nor thread-safe. In particular, to use this style one must ensure that no caller of `HI` ever keeps the result across a second call as it would be overwritten when the buffer is reused.

The main function in the collective object is

```
void Adiabatic_mu(double mu_i, double mu_f, double period, bool bar);
```

This function evolves the system, changing the parameter μ from mu_i to mu_f linearly during the period of time given by `period`. The parameter `bar` controls whether the progress bar for the evolution is shown.

7.3.2. Main file

The main file declares two collective objects: one in order to do the time evolution and the other as a reference.

```
1 int main(){
2   //Parameters
3   double mu=10.0;
4   double mu2=10;
5   double wmin=-2;
6   double wmax=2;
7   double th=0.01;
8   double th2=0.01;
9   int Nbins=200;
```

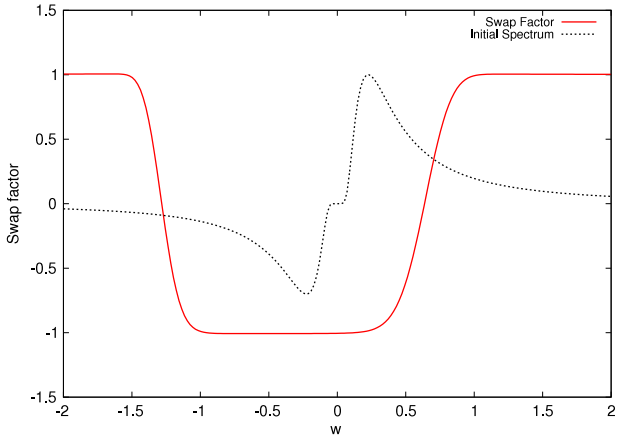

Fig. 7.3. Swap factor (solid line) of the system after going from $\mu = 10$ to $\mu = 0$ in a time period of 100 time units. The dotted line shows the initial spectra.

```
10
11    collective ColNus(mu,th,wmin,wmax,Nbins);
12    collective ColNus_notevolved(mu2,th2,wmin,wmax,Nbins);
13
14    //Evolution from mu=10 to mu=0 in a time period of 100
15    ColNus.Adiabatic_mu(10,0,100,true);
16
17    SU_vector o=ColNus.ez;
18    double max=0;
19    //find the maximum of the initial spectrum
20    for(int w=0;w<Nbins;w++){
21      if(max<ColNus_notevolved.GetExpectationValue(o,0,w))
22        max=ColNus_notevolved.GetExpectationValue(o,0,w);
23    }
24
25    //write the output in the file
26    //col 1 value of w
27    //col 2 expectation value of ez for the evolved system normalized to the maximum
28    //col 3 expectation value of ez for the non evolved system normalized to the maximum
29    std::ofstream file("collective.dat");
30    for(int w=0;w<Nbins;w++){
31      file << std::scientific << ColNus.Get_x(w) << "\t"
32           << ColNus.GetExpectationValue(o,0,w)/max <<"␣␣" <<
33        ColNus_notevolved.GetExpectationValue(o,0,w)/max << std::endl;
34    }
35
36    //runs the gnuplot script if yes
37    std::string plt;
38    std::cout << std::endl <<  "Done!␣" << std::endl <<  "Do␣you␣want␣to␣run␣the␣gnuplot␣script
        ?␣yes/no" << std::endl;
39    std::cin >> plt;
40
41    if(plt=="yes" || plt=="y"){
42      return system("./plot.plt");
43    }
44
45    return 0;
46  }
```

After declaring the objects we evolve ConNus from $\mu = 10$ to $\mu = 0$ in a time period of 100 time units. As before a gnuplot script is added in order to plot the result, which is shown in Fig. 7.3.

Acknowledgments

The authors acknowledge support from the Wisconsin IceCube Particle Astrophysics Center (WIPAC). C.A. and C.W. were supported in part by the US National Science Foundation under Grant Nos. OPP-0236449 and PHY-0969061 and by the University of Wisconsin Research Committee with funds granted by the Wisconsin Alumni Research Foundation. J.S. was funded under contract DE-FG-02-95ER40896 from the US Department of Energy.

References

[1] M.C. Gonzalez-Garcia, F. Halzen, M. Maltoni, Physics reach of high-energy and high-statistics icecube atmospheric neutrino data, Phys. Rev. D 71 (2005) 093010.
[2] L. Wolfenstein, Neutrino oscillations in matter, Phys. Rev. D 17 (1978) 2369–2374.
[3] G. Sigl, G. Raffelt, General kinetic description of relativistic mixed neutrinos, Nuclear Phys. B 406 (1993) 423–451.
[4] Robert Gilmore, Lie Groups, Physics, and Geometry, Cambridge University Press, 2008.
[5] GNU Scientific Library. http://www.gnu.org/software/gsl/.
[6] Carlos A. Argüelles, Joachim Kopp, Sterile neutrinos and indirect dark matter searches in IceCube, J. Cosmol. Astropart. Phys. 1207 (2012) 016.
[7] J. Wensch, M. Däne, W. Hergert, A. Ernst, The solution of stationary ode problems in quantum mechanics by magnus methods with stepsize control, Comput. Phys. Comm. 160 (2) (2004) 129–139.
[8] Frederick Ira Moxley III, Tim Byrnes, Fumitaka Fujiwara, Weizhong Dai, A generalized finite-difference time-domain quantum method for the-body interacting hamiltonian, Comput. Phys. Comm. 183 (11) (2012) 2434–2440.
[9] Tomasz Dziubak, Jacek Matulewski, An object-oriented implementation of a solver of the time-dependent schrödinger equation using the cuda technology, Comput. Phys. Comm. 183 (3) (2012) 800–812.
[10] Miguel A.L. Marques, Alberto Castro, George F. Bertsch, Angel Rubio, octopus: a first-principles tool for excited electron-ion dynamics, Comput. Phys. Comm. 151 (1) (2003) 60–78.
[11] H. Weimer, libquantum. http://www.libquantum.de/.
[12] Super-Kamiokande Collaboration. Prob3++. http://www.phy.duke.edu/raw22/public/prob3++/.
[13] Marius Wallraff, Christopher Wiebusch, Calculation of oscillation probabilities of atmospheric neutrinos using nucraft. 09 2014.
[14] Patrick Huber, Joachim Kopp, Manfred Lindner, Mark Rolinec, Walter Winter, New features in the simulation of neutrino oscillation experiments with GLoBES 3.0: General Long Baseline Experiment Simulator, Comput. Phys. Comm. 177 (2007) 432–438.
[15] Free Software Foundation. Gnu compiler collection. https://gcc.gnu.org, 2015.
[16] LLVM Project. Llvm/clang compiler. http://clang.llvm.org, 2015.
[17] Intel Corporation. Intel c/c++ compilers (https://software.intel.com/en-us/c-compilers), 2015.
[18] J.I. Fuks, N. Helbig, I.V. Tokatly, A. Rubio, Nonlinear phenomena in time-dependent density-functional theory: What Rabi oscillations can teach us, Phys. Rev. B 84 (7) (2011).
[19] Georg G. Raffelt, N-mode coherence in collective neutrino oscillations, Phys. Rev. D 83 (2011) 105022.