Quantum Simulator (Matlab)

Michael Schilling[1]

[1] *Humboldt Universität zu Berlin*

February 5, 2020

The Quantum Simulator was written in matlab 2016a, in order to verify and benchmark quantum computer algorithms in matlab. It was written by Michael Schilling for his masters thesis "Quantum Simulation of Bosons". It is currently capable of the calculation of the unitary matrices of an ideal quantum circuit (without errors) whereas the evolution of density matrices via a Markovian error model is still under development. Furthermore, the simulator provides a variety of tools to represent circuits and the data that has been generated. Currently most tools, while being general purpose, have been designed with the purpose of the simulation of coupled bosonic modes, but its development remains an ongoing project.

1 Installation & Requirements

The *Quantum Simulator* runs on Windows machines, but should also work under Linux, although the required folders, won't be added to the matlab path variable automatically. In order for the *Quantum Simulator* to work, add `poppler` from the Folder `Required_Ressources` to your system variables, in order to use *.pdf* to *.png* transformations (these are required for previews of Representation scripts that create LaTeX files.

2 Folder Structure

An overview of the main folders is given in figure 1. Examples for algorithms (used in my Master's thesis) can be found in the folders `0_2_Two_WG_Convergence` & `0_3_Multi_WG_Convergence`. These folders contain the Simulation results of $M = 2$ and $M \geq 2$ coupled bosonic modes, each using the XY-Model, Ladder coding, Gray coding and the Gauss-Jordan method. The other folders will be explained in the following sections.

3 Basic Procedure

A quantum simulation will typically consist of the following steps

- Defining the Gates A quantum circuit and the gates it consists of are treated as composite gates. Composite gates can consist of composite gates or elementary gates. Both the set of elementary gates and the composite gates are represented in a struct and can be defined or loaded by the user.
  - Initial state In case the simulation is run with a specific initial state, this initial state has to be defined as a wave vector.
- Calculating the Unitary Matrix of the Circuit The gates are expanded to determine the unitary transformation performed by the circuit. This can be done independently of a specific initial state. For an ideal quantum computer, this furthermore supports the use of gates based on function handles and symbolic gates.
- Analysis The results of a simulation can be analysed, by comparing with an analytical result and the results represented as different plots.

These will be expanded upon in the following sections.

4 Defining Gates

Gates in the quantum simulator are the array elements of a (gate) structure (from now on these array elements will be referred to as gates). Basic composite and elementary gate set are defined in `Gates_Table`, they can be generated via `quantum_gates.m`. An

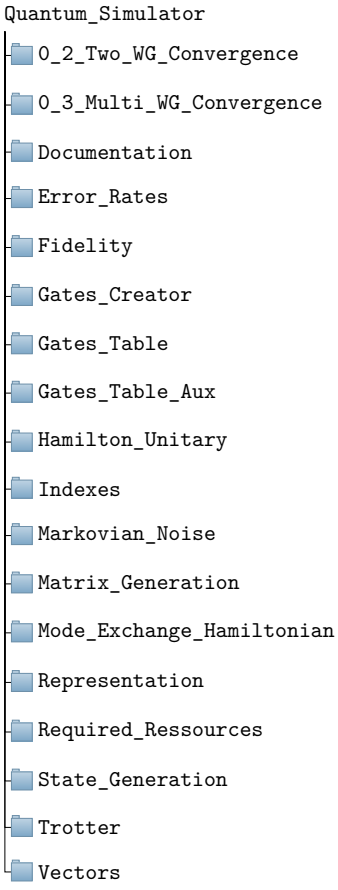

Figure 1: *Folder Structure.*

introduction into some of the parameters, that can be specified for a gate follows. For comparison, see the elementary gates in `Gates_Table/elem_gates.m` or the composite gates in `Gates_Table/comp_gates.m`.

`names` contains either a string, if only one possible name is specified

```
elem_gates(11).names='CZ',
```

or a cell array of strings, if multiple names are possible

```
elem_gates(12).names=···
 ···{'CZ_phi','CPHASE1','C1PHASE1','CPHASE'}.
```

`circuit` Definition of the circuit diagram associated with a gate. Can be specified manually, or if merely a box diagram with (or without) conditions is required, it can be generated via the program `Generate_Gate_Circuit.m`. The circuit elements are defined via the syntax of the LaTeX package `qcircuit`.

Single qubit gates can be defined via a string, defining the gate string in the `qcircuit`. For example `'\gate{H}'` for the Hadamard gate.

Multi qubit gates have to be defined (and single qubit gates can be defined) via cell arrays. The first row of the cell array represents the default circuit, whereas the second row allows for an alternative circuit if the gate isn't applied in normal ordering. If an alternative gate hasn't been specified, but the gate uses `\multigate` (see the `qcircuit` manual), the circuit creation algorithms automatically substitute this by multiple smaller gates, connected by vertical wires. For each row, the first column element is a matrix specifying, the relative position of the different elements (and therefore referred to as the position matrix), which are then defined via strings in the subsequent columns (via strings). There have to be as many columns in the position matrix, as there are strings, for the elements in the following columns of the `circuit` cell array. The first row of the positions matrix specifies the index of the qubit (of the gate in normal ordering) at which the element is going to be positioned. The second row specifies the corresponding position along the time axis of the quantum circuit (this is 1 for most gate circuits, but a gate could be wider this way). As an example, see the `circuit` of the Pauli-ZZ gate defined in `Gates_Table/comp_gates.m`

```
comp_gates(1).circuit{1,1}=[1,2;1,1]

comp_gates(1).circuit{1,2}=···
   ···'\multigate{1}{e^{-i\frac{\phi}{2}zz}}'

comp_gates(1).circuit{1,3}=···
          ···'\ghost{e^{-i\frac{\phi}{2}zz}}'
```

If the strings contain a substring of the form `_#_`, where `#` is an integer number, this substring is replaced by the number of the qubit of number `#`. This allows conditions to always be connected to the correct qubit. For example the circuit string `'\ctrl{_1_}'` connects the condition to the first qubit of the gate (in normal ordering).

`twirling` Specifies the error model used for the (elementary) gate. Is determined by a string, with 3 possibilities `''` for the single qubit error, that was specified for the Markov calculation, `'0'` if the gate is performed without an error (for example a global phase rotation, that is merely used to make the resulting unitary evolution look more familiar) and `'cz'` for the controlled-Z rotation gate.

`size` The number of qubits, that the gate acts upon.

`anc_size` The number of ancilla qubits that the gate acts on directly (via the gates that it is composed of). The ancilla gates, that the gates in the gate definition apply to are not specified here, these are determined automatically and addressed automatically.

`matrix` The unitary matrix of the gate, can be specified as a symbolic or as a numerical matrix. The matrix has to be defined for the elementary gates, but

can be calculated via the methods in 5.

fun_mat Additional definition of the matrix (of an elementary gate) as a function handle. This allows a program called `Gates_Table_Prepare.m` to optimize the circuits, and determine the necessary substitutions. The function handle is given as a string, that can be constructed via the script `matlabFunction.m`.

fun_vars Additional definition of the variables of `fun_mat` for an elementary gate. These are also defined as strings.

global_phase Defines a gloabl phase factor to present the gate in a more conventional or more desired notation.

The composite gates are constructed from other gates. These other gates are specified by the `steps`

steps.gates A cell array of strings specifying the names of the gates, in the sequence that they are applied. For example for the *CNOT* gate this results in `{'H','CZ','H'}`

steps.index A cell array of vectors specifying the indexes of the qubits to which the gates, specified in `steps.gates` are applied. For example for the *CNOT* gate this results in `{[1],[1,2],[1]}`

steps.param The gates defined in `steps.gates` can have parameters (like the continuous Pauli rotation gates, or the CZ_ϕ gate). For every gate, parameter substitutions can be defined, that either set them to a specific parameter, or substitute the parameter by another parameter. The parameters of the gate can be set to a specific numeric value via an array of the numeric values. These then get applied to the parameters in alphabetic order. Alternatively the parameters can be substituted by either symbolic variables or by numeric values using cell arrays. In this case, every gate defined in `steps.gates` corresponds to a cell array in `steps.param`. In these cell arrays, the entries are tuples, so that the first (uneven) entry of a tuple defines what terms are to be substituted (this does not have to correspond to a variable, but can also be a function of variables - although this is not advisable), and the second (even) entry corresponds to the term, that the first is to be substituted by.

step_num The number of gates in `steps`.

circuit_subs Substitutions for the strings in a circuit. A typical example is the substitution of the parameters, so that `'\phi'` is substituted by `'\frac{\pi}{2}'`. The circuit subs are defined for each of the gates in step.

Scripts that help with concatenation of gates, generation of new gates and the construction of conditional $C^n\hat{U}$ gates can be found in `Gates_Creator`. Particle exchange rotation gates (for the different encodings), the construction of unitary rotation gates from angles (as well as the extraction of angles from unitaries), and the generation of the quantum Gauss-Jordan decomposition are defined in the corresponding subfolders of `Gates_Creator`.

Circuits (composite gates) can be preprocessed via `Gates_Tables_Prepare.m` in the folder `Gates_Table_Aux`. This determines the hierarchy of composite gates, expands them into the elementary components, determines the necessary amount of ancilla qubits and the necessary substitutions for the gate parameters (when using continuous sets of elementary gates). Notably, the two programs `Gate_by_Name.m` & `Gate_Index_by_Name.m` (in `Gates_Table_Aux`), return a gate specified by one of it's names, if it resides within a gate list or it's gate index within the gate list, respectively.

5 Calculating the Unitary Matrix

The folder `Matrix_Generation` offers programs for different purposes.

Embed_Gate.m Embeds gates in a qubit register, expanding the gates matrix representation onto the larger Hilbert space, based on the Embedding approach in my thesis. This offers advantages over the decomposition of an n-qubit gate into n-qubit Pauli matrices, that are then expanded onto the register via the Kronecker product, as the function `Embed_Gate.m` offers similar speed for an embedding as the Kronecker product *kron(A,B)* requires for the same matrix size, but doesn't require the construction of a sum of Kronecker expanded terms to construct the matrix. The speed-up becomes significant for larger gates (as the number of Kronecker expanded Pauli matrices generically rises exponentially with the number of qubits, that a gate acts on). This function is both capable of expanding symbolic matrices, and numeric matrices.

Gate2Matrix.m Creates a matrix from a gate, by expanding the gates to a certain `depth`. This means, that the gates defined in `steps.gates` (if they are not elementary), get expanded further into their constituent gates until the desired depth has been reached. Returns the resulting matrix, the matrix including the ancilla qubits, verifies if all ancillas are back in their original position, and if desired, a list of the constituent gates (and their indexes) expanded to the specified depth.

Expansion2Matrix.m Creates a matrix from a gate, that has been preprocessed with

`Gates_Tables_Prepare.m`. Preprocessed gates contain additional parameters (`exp_steps.index`, `exp_steps.matrix`, `exp_steps.param` & `exp_steps.global_phase`), of an expansion into the elementary gates. This way, the substitutions have already been carried out ahead of time, and repetitive substitutions are optimized. Remaining parameters can be specified.

There are variants on the two previous programs, that require a wavevector as an input, and that return the time evolution of the wavevector for every gate in the expansion (either for a specified depth or the elementary gate decomposition).

Furthermore the programs `RandH.m` & `RandU.m` create random hermitian and unitary matrices of specified dimension n.

6 Representing Results

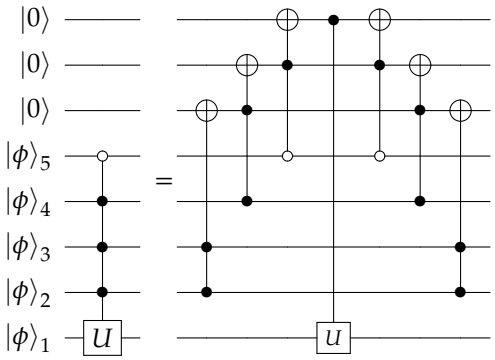

Figure 2: *Circuit representation of the gate* $C^4_{1110}\hat{U}$.

The Folder `Representation` contains many scripts for generating representations of data generated by the Quantum Simulator. The main subfolders are indicated by the prefix `0_`.

6.1 Quantum Circuit Representation

A first example can be created via the program `\Representation\0_Circuits\Test\test_Gate_2_tex.m`, which (with the default settings) creates a circuit preresentation (of depth 1) for the $C^4_{1110}\hat{U}$ gate. This circuit is constructed by using the program `Gates2Tex.m`, to create a LaTeX file for the circuit. Afterwards it gets automatically compiled into a .pdf and previewed in matlab by using the program `tex2pdf2preview.m`. The result can be seen in figure 2.

The program `Gates2Circ2Preview.m` combines the two previously mentioned functions into one, offering quick and easy previews of gate circuits. The mode of a gate construction can be specified via a string. The possible strings are `'gate'`, which merely shows the gate representation of the input gate, `'expansion'`, which shows the expansion of the gate (in normal ordering) up to a specified depth and finally `'gate=expansion'`, showing both, and equating them (see fig. 2).

6.2 Matlab2Tikz

This subfolder contains scripts for converting LaTeX documents to .pdf and then to .png's that can be previewed from within matlab.

6.3 Matrices

The subfolder `0_Matrices` contains a variety of programs to represent matrices.

`FockPrint.m` Prints a reduced matrix to the console (the `extra` command `'full'` changes this from reduced to full matrix). The reduced matrix consists only of the dimensions, that have non trivial elements either in the column or the row for said dimension. The option `'uni'` changes this to dimensions, that have non trivial elements besides the diagonal. As the matrices of many gates are very large, but often sparse, this reduces the matrices to the relevant entries. The dimensions are indicated by bra and ket vectors at the left and top of the matrix (there are multiple bra and fock options like `'dense'` for horizontal bra vectors, please see `Sparse2Square.m` for all the definition of the options). An example follows below. For the Toffoli gate $\hat{T}$ the command `FockPrint(T,'unit')` outputs

```
1                  <1,1,0|   <1,1,1|
2  |1,1,0>           0         1
3  |1,1,1>           1         0
```

are the non trivial elements of the Toffoli matrix. This is an easier to use version of `Sparse2Square.m`, requiring less inputs. The matrices support substitutions and via the option `'def_subs'` the elements of $\hat{U}(2)$ are substituted, for better readability. As an example see the $C\hat{U}(2)$ gate, for which the command `FockPrint(CU,'def_subs')` returns

	<0,0|	<0,1|	<1,0|	<1,1|
|0,0>	1	0	0	0
|0,1>	0	1	0	0
|1,0>	0	0	U11	U12
|1,1>	0	0	U21	U22

`FockTexPreview.m` Creates a LaTeX file, of a matrix with bra and ket vectors, and preview of the compiled file. The procedure is equivalent to that of `FockPrint.m`, but requires the addition of a filename. For an example constructed via the command `FockTexPreview(CU,'name','fock_dense_def_subs')`, see fig. 3.

$$\begin{array}{c} \\ \langle \mathbf{0,0}| \\ \langle \mathbf{0,1}| \\ \langle \mathbf{1,0}| \\ \langle \mathbf{1,1}| \end{array} \begin{array}{c} \begin{array}{cccc} \widehat{|\mathbf{0,0}\rangle} & \widehat{|\mathbf{0,1}\rangle} & \widehat{|\mathbf{1,0}\rangle} & \widehat{|\mathbf{1,1}\rangle} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \hat{U}_{11} & \hat{U}_{12} \\ 0 & 0 & \hat{U}_{21} & \hat{U}_{22} \end{pmatrix} \end{array}$$

Figure 3: *Matrix representation of the gate $C\hat{U}(2)$.*

6.4 Plotting

This folder contains many different scripts to plot density matrices via 3D bar plots (see `DensityBar3.m`, and fig. 4 for an example) or via PColorPlots (see for example `PColorTikz.m`), to animate unitary matrices over time (see `Unitary_Anim.m`), to plot which method vonverges the fastest to a desired target Fidelity (see the subfolder `Best_Method_Plots`), or the time evolution of waveguides (see `WG_Plot.m`).

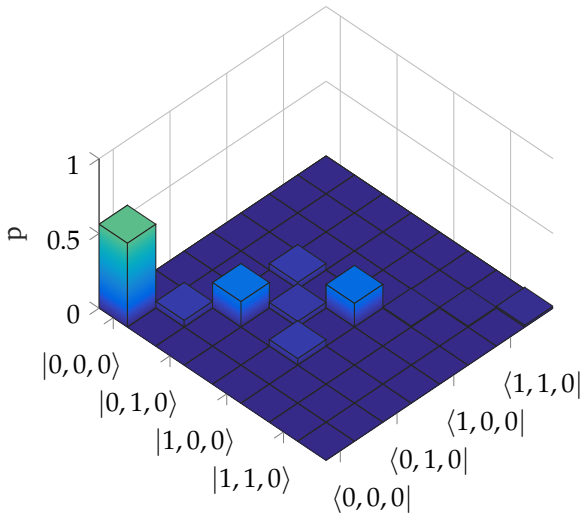

Figure 4: *Density matrix ρ of a state that has mostly decohered.*