SPINE Manual

J.P.G. van Dijk

January 10, 2020

Contents

1 Introduction

SPINE – SPIN Emulator – is a tool for the co-simulation of classical electrical signals with spin-based quantum processors, and was originally introduced in [1, 2]. The tool was extensively used in deriving the impact of classical control electronics on the fidelity of a single-electron spin qubit [3]. However, the toolset can be directly extended to other qubit technologies.

1.1 Simulation Platforms

The simulator is implemented in different platforms, with different features/limitations as discussed in the following.

1.1.1 MATLAB

As SPINE was originally written in MATLAB, this implementation contains all features present in the other platforms. Leveraging the power of MATLAB, it has better plotting functions (e.g., 3D plot) and additional solvers (`solver_expm`, `solver_taylor_sparse_approx`).

This implementation was tested on Windows 10 Pro 64-bit (1903) running MATLAB R2018a 64-bit.

1.1.2 C++

The C++ implementation of the simulator can in principle be used on any host OS, without the need for MATLAB to be installed on the system. Additionally, by defining `MKL`, the Intel© Math Kernel Library (with or without multi-threading [requires MPI]) is used to optimize most calculations. Moreover, there is explicit control over the used precision, with single precision floating point arithmetic the default, unless `DOUBLE_PRECISION` is defined (advised). Finally, when running on Microsoft Windows, plotting functions are available, which can be enabled by defining `PLOT`. All matrices are stored in row-major order.

This implementation was tested on Windows 10 Pro 64-bit (1903) with MKL version 2019.5.281, compiled using Visual Studio 2017 (v141) and Windows SDK Version 10.0.17763.0.

1.1.3 Verilog-A

The Verilog-A implementation is meant for the simulation of the quantum processor in an electrical circuit simulator supporting Verilog-A models. Because of the limitations of the Verilog-A language, only the following modules are available:

- `spine_qubit1`: a port of the simulator for one single-electron spin qubit in the lab frame (`system_1_spin`) solved using the available analytical solution (`solver_analytical_xz`).
- `spine_qubit2`: a port of the simulator for two single-electron spin qubits, each with singlet state (`system_2_spin_2_singlet`), solved using a Taylor series expansion (`solver_taylor`).

This implementation was tested on a CentOS 5.11 server (linux kernel 2.6.18-410.el5) running Cadence® IC6.1.5.500.6 (32-bit) with Spectre version 7.2.0.307.isr10.

More details of the Verilog-A implementation can be found in Section 8.

2 SPINE

2.1 Hamiltonian Simulation

The evolution of the quantum processor's state, captured as a vector $|\psi\rangle$, can be described by the multiplication with a unitary matrix U: $|\psi(t)\rangle = U \cdot |\psi(0)\rangle$. In general, finding this operation U involves solving the time-dependent Schrödinger equation:

$$\mathrm{i}\hbar \cdot \frac{\partial|\psi(t)\rangle}{\partial t} = H(t) \cdot |\psi(t)\rangle, \tag{1}$$

where $H(t)$ is the Hamiltonian describing the system. In general, an approximate solution can be more easily found by simulation. Such a Hamiltonian simulation relies on the fact that the solution to a time-independent Hamiltonian H_n is trivially:

$$U_n = \mathrm{e}^{-\mathrm{i}/\hbar \cdot H_n \cdot t}. \tag{2}$$

Then, by approximating the time-dependent Hamiltonian as many time-independent Hamiltonians H_n that are valid only for a short duration t, which are all applied subsequently, the overall operation can be found:

$$U_{\text{overall}} = \prod_{n=N}^{1} U_n = U_N \cdot \cdots \cdot U_2 \cdot U_1. \tag{3}$$

This process is called trotterization. Note that the order of the matrix multiplication matters. Moreover, for simplicity $\hbar = 1$ throughout SPINE.

2.2 SPINE Simulator Core

The pseudo-code of the core of the SPINE simulator is shown below:

```
function SIMULATE($inHamiltonian$, $outOperation$, $solver$)
    $U \leftarrow I$
    while $inHamiltonian(H, dt)$ do
        $dU \leftarrow solver(H \cdot dt)$
        $U \leftarrow dU \cdot U$
        $outOperation(U)$
    end while
end function
```

This function calculates the overall operation U at any time, given the user-provided Hamiltonian H and time step dt at that time (through the callback function *inHamiltonian*) and passes the resulting U, at that time, back to the user (through the callback function *outOperation*). The user needs to provide a *solver*(arg) that calculates $\mathrm{e}^{-\mathrm{i} \cdot \mathrm{arg}}$.

If instead of a simulation of the full quantum operation, only a simulation of the quantum state is desired, the faster simulation as shown below is executed:

```
function SIMULATE($inHamiltonian$, $outOperation$, $solver$, $state$)
    while $inHamiltonian(H, dt)$ do
        $dU \leftarrow solver(H \cdot dt)$
        $state \leftarrow dU \cdot state$
        $outOperation(state)$
    end while
end function
```

This function calculates the quantum state *state* at any time, given the user-provided Hamiltonian H and time step dt at that time (through the callback function *inHamiltonian*) and passes the resulting *state*, at that time, back to the user (through the callback function *outOperation*). The user needs to provide a *solver*(arg) that calculates $\mathrm{e}^{-\mathrm{i}\cdot\mathrm{arg}}$.

The advantages of such a generic setup using callbacks are as follows:

- Allows for dynamic time step size.
- There is no requirement for a predetermined number of points or simulation time.
- There is no requirement for a predetermined signal; it can be generated based on previous simulation points if desired.
- There is freedom in what to do with the simulated operation/state at any time instance, e.g. plot or save every N^{th} point.

2.3 Functions

2.3.1 simulate

The function prototype of *simulate* in MATLAB and C++, respectively, is:

```
function simulate(dim, inHamiltonian, outOperation, solver,
    varargin)
```

```
void simulate(unsigned int dim,
        bool(*inHamiltonian)(complex * H, realnum * dt),
        void(*outOperation)(complex * U),
        void(*solver)(unsigned int dim, complex * H, complex * dU))
            ;
void simulate(unsigned int dim,
        bool(*inHamiltonian)(complex * H, realnum * dt),
        void(*outOperation)(complex * U),
        void(*solver)(unsigned int dim, complex * H, complex * dU),
        complex * state);
```

The dimension of the Hamiltonian should be passed in `dim`. Additionally, handles to the functions `inHamiltonian`, `outOperation` and `solver` should be passed. For the C++ implementation, additional optimized functions are available when using real-valued Hamiltonians (`arg` is real). In both the MATLAB and C++ implementation, an additional optional argument `state` can be passed in which case only the quantum state evolution, instead of the full quantum operation, is simulated, assuming initial state `state`.

2.3.2 `inHamiltonian`

The function prototype of *inHamiltonian* in MATLAB and C++, respectively, is:

```
function [run, H, dt] = inHamiltonian()
```

```
bool inHamiltonian(complex * H, realnum * dt);
```

For the MATLAB implementation, `run`, specifies whether the simulation should continue or not, whereas this is returned as boolean variable in the C++ implementation. For the C++ implementation, the memory regions passed as arguments (`H` and `dt`) are filled by the function `inHamiltonian`. For the C++ implementation, an additional optimized function is available when using real-valued Hamiltonians.

Several Hamiltonians for different spin systems are provided in the MATLAB package `spine.systems` and the C++ namespace `spine::systems`. For more information, see Section 3.

2.3.3 `outOperation`

The function prototype of *outOperation* in MATLAB and C++, respectively, is:

```
function outOperation(U)
```

```
void outOperation(complex * U);
```

Several helper functions for plotting the operation/state and calculating the fidelity of the operation are provided in MATLAB as `spine.plot*()` and `spine.fidelity()`), respectively, and in C++ as `spine::Plot()` and `spine::fidelity()`, respectively. For more information, see Section 4.

2.3.4 `solver`

The function prototype of *solver* in MATLAB and C++, respectively, is:

```
function dU = solver(dim, arg)
```

```
void solver(unsigned int dim, complex * arg, complex * dU);
```

The dimension of the argument `arg` should be passed in `dim`. For the C++ implementation, the memory region passed as argument (`dU`) is filled by the function `solver`. For the C++ implementation, additional optimized functions are available when using real-valued Hamiltonians (`arg` is real).

Several solvers are provided in the MATLAB package `spine.solvers` and the C++ namespace `spine::solvers`. For more information, see Section 5.

3 Implemented Spin Systems

For the simulation of single-qubit gates on an isolated quantum dot, a good approximation is obtained when only considering the spin-up and spin-down states of the electron, i.e.: $|\psi\rangle = \alpha_0 \cdot |\uparrow\rangle + \alpha_1 \cdot |\downarrow\rangle$, where $|\uparrow\rangle$ and $|\downarrow\rangle$ can be considered the qubit states $|0\rangle$ and $|1\rangle$ respectively. The state vector only contains 2 complex numbers, and the matrices involved in the simulation are 2×2.

The Hamiltonian of the spin qubit with the energy levels split by the Zeeman energy $\hbar\omega_0$ under excitation by a signal $x(t)$ in a perpendicular magnetic field is given by (setting $\hbar = 1$):

$$H(t) = -\omega_0 \frac{\sigma_z}{2} + x(t)\frac{\sigma_x}{2} \tag{4}$$

where σ_x and σ_z are the X and Z Pauli matrices. In general, the signal $x(t)$ is a sinusoidal signal with frequency ω_0 and varying amplitude.

For the simulation of two qubits, it would seem sufficient to simply take the qubit state as $|\psi\rangle = \alpha_{00} \cdot |\uparrow\uparrow\rangle + \alpha_{01} \cdot |\uparrow\downarrow\rangle + \alpha_{10} \cdot |\downarrow\uparrow\rangle + \alpha_{11} \cdot |\downarrow\downarrow\rangle$ with a Hamiltonian $H = H_A \oplus H_B$ where H_i describes the physics of a single quantum dot. This is indeed sufficient for the simulation of single-qubit operations on multiple isolated qubits. However, for the simulation of two-qubit gates at least one more energy level should be included in the simulation that is responsible for the qubit interactions in the physical system: $|\psi\rangle = \alpha_{00} \cdot |\uparrow\uparrow\rangle + \alpha_{01} \cdot |\uparrow\downarrow\rangle + \alpha_{10} \cdot |\downarrow\uparrow\rangle + \alpha_{11} \cdot |\downarrow\downarrow\rangle + \alpha_{S0} \cdot |S0\rangle$ where $|S0\rangle$ describes the lowest energy state (a Singlet state) where both electrons have moved into one of the quantum dots. The corresponding system Hamiltonian, now also of size 5×5, is given by [4]:

$$H = \begin{bmatrix} -\frac{\omega_{0,A}+\omega_{0,B}}{2} & 0 & 0 & 0 & 0 \\ 0 & -\frac{\omega_{0,A}-\omega_{0,B}}{2} & 0 & 0 & t_0 \\ 0 & 0 & \frac{\omega_{0,A}-\omega_{0,B}}{2} & 0 & -t_0 \\ 0 & 0 & 0 & \frac{\omega_{0,A}+\omega_{0,B}}{2} & 0 \\ 0 & t_0 & -t_0 & 0 & U-\epsilon \end{bmatrix} \tag{5}$$

where for simplicity the driving term $x(t)$ has been left out. The tunnel coupling t_0 and detuning ϵ are generally time-varying signals. This Hamiltonian allows for the simulation of the controlled-Z gate at a detuning ($\epsilon \approx U$).

However, for the simulation of e.g. a controlled-Z gate at no detuning, or a SWAP gate, the state vector has to be expanded with the $|0S\rangle$ state, describing that both electrons can also go into the other quantum dot. For more accurate simulations, as also required for the simulation of a qubit measurement by pauli-spin blockade, even higher energy levels have to be included.

Due to the fact that it is not sufficient to only simulate two energy levels for a single quantum bit, the simulation complexity grows very rapidly for larger qubit systems. Considering an isolated system (the number of electrons does not change), where only the lowest states with 2 electrons in a single dot are considered (the Singlet states), the length of the state vector grows as:

Qubits	Dimension
2	6
3	20
4	70
5	252
6	924

In MATLAB the following systems are currently available for simulation (in the package `spine.systems`):

- `spine.systems.system_spin`
  - `spine.systems.system_1_spin`
  - `spine.systems.system_1_spin_rwa`
  - `spine.systems.system_2_spin_1_singlet`
  - `spine.systems.system_2_spin_2_singlet`
  - `spine.systems.system_2_spin_1_singlet_triplet`
  - `spine.systems.system_2_spin_2_singlet_triplet`
  - `spine.systems.system_n_spin_n_singlet`
- `spine.systems.system_1_singlet_triplet`
- `spine.systems.system_dispersive_readout`

In C++ the following systems are currently available for simulation (in the namespace `spine::systems`):

- `spine::systems::system_spin`
  - `spine::systems::system_1_spin`
  - `spine::systems::system_1_spin_rwa`
  - `spine::systems::system_2_spin_1_singlet`
  - `spine::systems::system_2_spin_2_singlet`
  - `spine::systems::system_2_spin_1_singlet_triplet`
  - `spine::systems::system_2_spin_2_singlet_triplet`
  - `spine::systems::system_n_spin_n_singlet`
- `spine::systems::system_1_singlet_triplet`
- `spine::systems::system_dispersive_readout`

3.1 `system_spin`

Each of the `system_*_spin` classes is derived from the `system_spin` base class that contains most of the functionality for simulating any number of *single-electron spin qubits* with any number of energy levels per dot considered (the 0 and 1 computational states, with optional singlet and triplet energy levels).

This base class contains the storage and general getters/setters for the Hamiltonian properties (Larmor frequency, Rabi frequency, charging energy, singlet-triplet energy splitting) and control variables (microwave signal, detuning signal, tunnel control signal) for every dot.

Additionally, it contains helper functions `initialize` and `measure` (`measureST`) to initialize a state vector to the ground state of the system and to measure the state of every qubit in the X,Y,Z basis (or the singlet and triplet occupancy probability of the dot), respectively. Optionally, a time t can be passed to the function `measure` to measure the state in the rotating frame at time t instead of the lab frame. These functions, however, rely on an implementation of the following functions in the derived class:

- `getIndex`: given the desired state of every quantum dot, passed as argument, returns the corresponding index in the state vector/Hamiltonian.
- `getIndexMeasurement`: given a dot and the state of interest, passed as argument, returns all indices in the state vector/Hamiltonian where that dot is in the desired state.
- `getDimension`: returns the dimension of the state vector/Hamiltonian.

On top of the `measure` functions, additional `plot` functions are provided to plot the X,Y,Z measurement probability for every dot in a Bloch sphere (MATLAB) or a simple 2D plot (C++, Windows only), see Section 4.

3.1.1 `system_1_spin`

This class contains the 2×2 real-valued Hamiltonian of 1 single-electron spin-qubit in a single quantum dot, considering only the energy levels of the 0 and 1 states.

3.1.2 `system_1_spin_rwa`

This class contains the 2×2 complex Hamiltonian of 1 single-electron spin-qubit in a single quantum dot, in the rotating frame with rotating wave approximation, considering only the energy levels of the 0 and 1 states.

3.1.3 `system_2_spin_1_singlet`

This class contains the 5×5 real-valued Hamiltonian of 2 single-electron spin-qubits in a double quantum dot, considering only the energy levels of the 0 and 1 states. Additionally, the singlet state energy level of one of the dots is included to allow for basic 2-qubit operations. For simplicity, a single microwave drive line and a common Rabi frequency is assumed for the two dots.

3.1.4 `system_2_spin_2_singlet`

This class contains the 6×6 real-valued Hamiltonian of 2 single-electron spin-qubits in a double quantum dot, considering the energy levels of the 0 and 1 states, and the singlet state for both quantum dots. For simplicity, a single microwave drive line and a common Rabi frequency and charging energy are assumed for the two dots. Moreover, only the relative detuning of the two dots is considered.

3.1.5 `system_2_spin_1_singlet_triplet`

This class contains the 8×8 real-valued Hamiltonian of 2 single-electron spin-qubits in a double quantum dot, considering only the energy levels of the 0 and 1 states. Additionally, the energy levels of the singlet and 3 triplet states of one of the dots is included to allow for basic 2-qubit operations and simulation of Pauli-spin blockade readout. For simplicity, a single microwave drive line and a common Rabi frequency is assumed for the two dots.

3.1.6 `system_2_spin_2_singlet_triplet`

This class contains the 12×12 real-valued Hamiltonian of 2 single-electron spin-qubits in a double quantum dot, considering the energy levels of the 0 and 1 states, singlet state and 3 triplet states for both quantum dots. For simplicity, a single microwave drive line and a common Rabi frequency, charging energy and singlet-triplet energy splitting are assumed for the two dots. Moreover, only the relative detuning of the two dots is considered.

3.1.7 `system_n_spin_n_singlet`

This class contains the real-valued Hamiltonian of N single-electron spin-qubits in N quantum dots ($N \geq 2$). For each quantum dot the energy levels of the 0 and 1 states, and the singlet state is simulated.

3.2 `system_1_singlet_triplet`

This class contains the 2×2 real-valued Hamiltonian of 1 singlet-triplet qubit in a double quantum dot, considering only the energy levels of the 0 and 1 states.

Similar as for the `system_spin` class, this class contains the storage and general getters/setters for the Hamiltonian properties (magnetic field gradient) and control variables (exchange interaction).

Additionally, it contains helper functions `initialize` and `measure` to initialize a state vector to the ground state of the system and to measure the state of the qubit in the X,Y,Z basis, respectively. On top of the `measure` function, additional `plot` functions are provided to plot the X,Y,Z measurement probability for the dot in a Bloch sphere (MATLAB) or a simple 2D plot (C++, Windows only).

3.3 `system_dispersive_readout`

Finally, the master equation governing dispersive readout has been rewritten in the form of a complex 2×2 non-Hermitian ‘Hamiltonian’ (not a true Hamiltonian as the resulting operation is not unitary). The implemented master equation is given by [5]:

$$\frac{\mathrm{d}P_1(t)}{\mathrm{d}t} + \Gamma_0 P_1(t) = \Gamma_+(t) \tag{6}$$

where P_1 is the probability of the electron being in the dot and Γ_+ is the tunnel rate:

$$\Gamma_+(t) = \frac{\Gamma_0}{1 + \mathrm{e}^{\Delta E(t)/k_B/T}} \tag{7}$$

where k_B is the Boltzmann constant, T the temperature, $\Delta E(t)$ the time-dependent energy difference and Γ_0 is the constant tunnel rate away from the degeneracy.

Similar as for the `system_spin` class, this class contains the storage and general getters/setters for the Hamiltonian properties (electron temperature, tunnel rate and lever arm) and control variables (gate voltage).

Additionally, it contains helper functions `initialize` and `measure` to set the initial probability and to measure the probability respectively. On top of the `measure` function, additional `plot` functions are provided to plot the measurement probability of the electron being in the dot in a simple 2D plot (MATLAB, Windows only for C++).

4 Helper Functions

For calculating the fidelity of a 2×2 unitary operation, the following functions are available in MATLAB and C++ respectively:

```
function F = fidelity(dim, U, varargin)
```

```
realnum fidelity(unsigned int dim, complex * U, complex * Uideal);
realnum fidelity(unsigned int dim, complex * U, realnum theta,
    realnum phi);
```

where the dimension of the unitary (i.e. 2) needs to be passed in the argument `dim`, and the unitary operation in `U`. Next, either a single argument must be passed containing the ideal unitary operation `Uideal`, or two arguments follow: the rotation angle `theta` and rotation axis `phi` of the ideal rotation.

4.1 Plotting in MATLAB

For plotting in MATLAB, the following function draws a simple 3D Bloch sphere:

```
function plotBlochSphere()
```

This function is for instance used in the base class `system_spin` (see Section 3.1) in the `plot` function, which takes the following arguments:

```
function plot(obj, state_or_U, t, varargin)
```

where `state_or_U` is either the quantum state or the unitary operation (in which case an initial ground state initialization is assumed to determine the state to visualize). The argument `t` is used to determine the measurement probability in the lab frame, which is shown by default. By passing a first optional argument `plot_lab_style`, the measurement probability in the lab frame is also shown as either an arrow (`plot_lab_style` = 1) or a full trace (`plot_lab_style` = 2). By passing a second optional argument `plot_st`, additional plots of the singlet-triplet state occupancy of the dot are generated. When `plot_st` = 1, only the singlet state occupancy is plotted; when `plot_st` = 2, the expected number of electrons in the dot is plotted.

Finally, the following function plots the eigenenergy diagram of a double dot spin system (i.e. the energy levels of the various states versus detuning):

```
function plotEigenEnergies(system, varargin)
```

where the system is passed in the argument `system`. By default the detuning is swept from -1.25 to 1.25 times the charging energy with 1001 points. By passing a second optional argument `points` this number can be changed.

4.2 Plotting in C++

Note: these plotting functions are only available under Windows and require PLOT *to be defined!*

The class `Plot` is used for plotting and has the following constructors:

```
Plot(LPCWSTR name, unsigned int points = 100);
Plot(LPCWSTR name, unsigned int points, double * xdata, double *
    ydata, BYTE r, BYTE g, BYTE b, double xmin, double xmax, double
     ymin, double ymax);
Plot(LPCWSTR name, unsigned int points, double * ydata, BYTE r,
    BYTE g, BYTE b, double xmin, double xmax, double ymin, double
    ymax);
```

where `name` is the title of the plotting window, `points` is the number of points (expected) to be plotted. Optionally, a curve can directly be plotted by passing the `ydata` and optionally the `xdata` along with the desired plotting color (`r`, `g`, `b`) and plot window limits (`xmin`, `xmax`, `ymin`, `ymax`).

An additional curve can be added through the method:

```
void add(BYTE r = 0, BYTE g = 0, BYTE b = 0, double xmin = 0,
    double xmax = 0, double ymin = 0, double ymax = 1);
```

which again takes the desired plotting color and plot window limits. Next, a handle to the curve can be obtained through the method:

```
PlotSeries * get(unsigned int n);
```

where `n` is the curve number (starting from 0 for the first curve that was added). For each of these curves, points can later be added, and the window limits can be changed, through the following methods of `PlotSeries`:

```
void add(double x, double y);
void xlim(double xmin, double xmax);
void ylim(double ymin, double ymax);
```

When using the latter 3 functions, a redraw must be forced by calling the `redraw()` method of the `Plot` class.

These plotting functions are for instance used in the base class `system_spin` (see Section 3.1) in the `plot*` methods:

```cpp
void plotSetup(unsigned int points = 100, double xmin = 0, double
    xmax = 0);
void plotAddU(complex * U, realnum t = 0, bool plot_lab = false);
void plotAdd(complex * state, realnum t = 0, bool plot_lab = false)
    ;
void plot();
```

From these functions, `plotSetup` can optionally be called before starting a simulation to setup up the number of points that will be plotted and the x-scale (alternatively, the x-scale will be adjusted with every point added), which can significantly speed-up the plotting. The functions `plotAddU` and `plotAdd` are used to add a plot point for the simulated unitary operation (again assuming an initial ground state initialization) or quantum state, equivalent to the MATLAB function `plot(obj, state_or_U, t, varargin)` (see Section 4.1). The argument `t` is used to determine the measurement probability in the rotating frame, which is shown by default. By passing setting the optional argument `plot_lab` to `true`, the measurement probability in the lab frame is also shown as a full trace. Unlike for the MATLAB implementation, an additional call to the `plot` method is required to update the actual graphics with the added points when desired.

Note: After all simulations/calculations are finished, the endless loop Plot::run() must be called, which handles the Windows GUI message loop that takes care of the GUI actions, such as plot window resizing/closing.

5 Implemented Solvers

In MATLAB the following solvers are currently available (in the package `spine.solvers`):

- `spine.solvers.solver_analytical_xz`
- `spine.solvers.solver_expm`
- `spine.solvers.solver_diagonalization`
- `spine.solvers.solver_taylor`
- `spine.solvers.solver_taylor_sparse_approx`

In C++ the following solvers are currently available (in the namespace `spine::solvers`):

- `spine::solvers::solver_analytical_xz`
- `spine::solvers::solver_diagonalization`
- `spine::solvers::solver_taylor`

5.1 `solver_analytical_xz`

This solver uses the analytical solution available in the case of a real-valued 2×2 Hamiltonian of the following form:

$$H = a \cdot \sigma_x + b \cdot \sigma_z \tag{8}$$

where σ_x and σ_z are the Pauli-X and Pauli-Z matrices, respectively, and a and b are real numbers.

5.2 `solver_expm`

This solver uses the `expm` function available in MATLAB to evaluate the matrix exponential.

5.3 `solver_diagonalization`

This solver calculates the matrix exponential using the eigenvalue decomposition. The Hamiltonian is assumed to be Hermitian (symmetric in case of a real-valued Hamiltonian).

5.4 `solver_taylor`

This solver approximates the matrix exponential using the Taylor series expansion upto an order that is set by the MATLAB global variable `solver_taylor_accuracy` or the C++ local variable `accuracy`. For increased accuracy, it uses the scaling-and-squaring method with scaling factor 2^N where N is set by the MATLAB global variable `solver_taylor_scaling` or the C++ local variable `scaling`. Default values are provided.

5.5 `solver_taylor_sparse_approx`

Same as the `solver_taylor`, but uses sparse matrices (available in MATLAB only). To ensure efficient use of sparse matrices, during the squaring step, all elements with magnitude below a certain level (set by the MATLAB global variable `solver_taylor_tolerance`) are discarded. While this reduces the simulation accuracy, it allows for significantly faster simulation of larger spin qubit systems.

6 Examples

Currently, equivalent examples are provided for the MATLAB and C++ implementation demonstrating each of the currently implemented spin systems:

- `example_1_spin`
- `example_1_spin_rwa`
- `example_2_spin_1_singlet`
- `example_2_spin_2_singlet`
- `example_2_spin_1_singlet_triplet`
- `example_2_spin_2_singlet_triplet`
- `example_n_spin_n_singlet`
- `example_1_singlet_triplet`
- `example_dispersive_readout`

6.1 `example_1_spin`

This example demonstrates a π-rotation on a single-electron spin qubit using a rectangular envelope. Every `Nplot`$^{\text{th}}$ unitary operation is plotted in the Bloch sphere, in both the lab frame and the rotating frame. Finally, the fidelity of the operation is calculated from the ideal rotation angle/axis and printed.

6.2 `example_1_spin_rwa`

This example demonstrates a $\pi/2$-rotation along the X-axis followed by a $\pi/2$-rotation along the Y-axis on a single-electron spin qubit in the rotating frame using a rectangular envelope. Every `Nplot`$^{\text{th}}$ unitary operation is plotted in the Bloch sphere, in the rotating frame (with additional arrow). Finally, the fidelity of the operation is calculated from the ideal unitary operation and printed.

6.3 `example_2_spin_1_singlet`

In this example, a controlled-NOT (CNOT), based on a 2-qubit CZ-gate at detuning, is demonstrated, where for the first CNOT the control qubit is in the ground state, and for the second CNOT the control qubit is in the excited state. Gaussian envelopes are used for the microwave control signal, which is frequency multiplexed over the 2 qubits. Every `Nplot`$^{\text{th}}$ unitary operation is plotted in the Bloch sphere, in both the lab frame (shown as an arrow) and the rotating frame.

6.4 `example_2_spin_2_singlet`

In this example, a SWAP gate is demonstrated at detuning for a double dot. Additionally, a simulation of the quantum state, instead of the unitary operation, is used, as the initial state is chosen differently for both qubits. Every `Nplot`$^{\text{th}}$ state is plotted in the Bloch sphere, in the rotating frame.

6.5 `example_2_spin_1_singlet_triplet` and `example_2_spin_2_singlet_triplet`

These examples demonstrate Pauli-spin blockade readout of a double quantum dot. Initially both qubits are in the ground state, and no charge is transferred when adiabatically detuning the quantum dots, whereas later charge is transferred as one qubit is rotated into the excited state. Every `Nplot`$^{\text{th}}$ unitary operation is plotted in the Bloch sphere, in the rotating frame.

6.6 `example_n_spin_n_singlet`

In this example 4 qubits are simulated and every `Nplot`$^{\text{th}}$ state is plotted, with the following instructions executed in sequence:

1. Rotate qubit 1 from 0 to 1 using a microwave signal
2. SWAP between qubit 1 and qubit 2 using the tunnel coupling
3. Rotate qubit 3 and qubit 4 in plane using a microwave signal
4. CZ-gate between qubit 1/qubit 3 and qubit 2/qubit 4

5. Rotate qubit 3 and qubit 4 out plane using a microwave signal, out of phase
6. Pauli-spin blockade measurement of qubit 3 with respect to qubit 4
7. Rotate qubit 1 using a microwave signal

6.7 `example_1_singlet_triplet`

In this example, a π-rotation along the Y-axis is demonstrated on a singlet-triplet qubit using a composite pulse comprising 2 rectangular pulses with different amplitude and duration. Additionally, this example demonstrates the use of a variable timestep, as both rectangular pulses are simulated with a different timestep. Every `Nplot`$^{\text{th}}$ unitary operation is plotted in the Bloch sphere, in the lab frame. Finally, the fidelity is calculated and printed given the ideal rotation angle and axis.

6.8 `example_dispersive_readout`

In this example, dispersive gate readout is demonstrated. A sinusoidal gate voltage is applied and the resulting dot occupancy is simulated. Finally, from this occupancy, the expected gate current is calculated and plotted. The resulting waveforms depend on the drive frequency relative to the tunnel coupling (capacitive vs. resistive regimes).

7 Templates

Below are the templates for a typical SPINE program written in MATLAB and C++. In these templates, `<SYSTEM>`, `<SET>`, `<VALUE>`, `<POINTS>` and `<SOLVER>` should be adjusted to the needs.

7.1 MATLAB

```
% Global variables
global nsim;
global Nsim;
global system;

% Create the system
system = spine.systems.<SYSTEM>();
system.<SET>(<VALUE>);

% Simulate the system
nsim = 0;
Nsim = <POINTS>;
spine.simulate(system.getDimension(), @inHamiltonian, @outOperation
    , @spine.solvers.<SOLVER>);

function [run, H, timestep] = inHamiltonian()
global nsim;
global Nsim;
global system;

if (nsim < Nsim)

```

```
% Set the signal at this time instance
system.<SET>(<VALUE>);

% Update the Hamiltonian accordingly
H = system.updateHamiltonian();

% Provide the current timestep, and continue the simulation
timestep = <TIMESTEP>;
run = true;

else
run = false;
H = [];
timestep = [];
end
end

function outOperation(U)
global nsim;
global Nsim;

% Plot, print, store
...

% Continue the simulation
nsim = nsim + 1;
if (nsim == Nsim)

% Last point
...

end

end
```

7.2 C++

```
// Includes
#include "spine/simulate.h"
#include "spine/math.h"
#include "spine/systems/<SYSTEM>.h"
#include "spine/solvers/<SOLVER>.h"

// Namespaces
using namespace spine::math;

// Function prototypes
bool inHamiltonian(realnum * H, realnum * dt);
void outOperation(complex * U);

// Private global variables/constants
static spine::systems::<SYSTEM> spin_system = spine::systems::<SYSTEM>();
static const realnum dt = (realnum) 10e-12;
static int Nsim, nsim;

int main(void)
{
// Create the system
spin_system.<SET>(<VALUE>);

// Simulate the system
nsim = 0;
Nsim = <POINTS>;
spine::simulate(spin_system.getDimension(), inHamiltonian,
    outOperation, spine::solvers::<SOLVER>);
}

bool inHamiltonian(realnum * H, realnum * dt)
{
if (nsim < Nsim)
{
// Set the signal at this time instance
spin_system.<SET>(<VALUE>);

// Update the Hamiltonian accordingly
spin_system.updateHamiltonian(H);

// Provide the current timestep, and continue the simulation
*dt = ::dt;
return true;
}
return false;
}

void outOperation(complex * U)
{
// Plot, print, store
...

// Continue the simulation
nsim++;
if (nsim == Nsim)
{
// Last point
...
}
}
```

8 Verilog-A

As advanced electrical circuit simulators use quasi-static time-domain solvers, they provide a favorable environment for the inclusion of a time-discrete Hamiltonian simulation. Using Cadence® as a framework, the quantum physical system is included in the electrical simulation as a module that takes as input the control signals for the quantum system and outputs the quantum operation (Figs. 1 and 2).

In the Verilog-A implementation, only modules emulating either one single-electron spin qubit (Fig. 1) or a system of two coupled single-electron spin qubits (Fig. 2) are currently available:

- `spine_qubit1`: a port of the simulator for one single-electron spin qubit in the lab frame (`system_1_spin`) solved using the available analytical solution (`solver_analytical_xz`).
- `spine_qubit2`: a port of the simulator for two single-electron spin qubits, each with singlet state (`system_2_spin_2_singlet`), solved using a Taylor series expansion (`solver_taylor`).

The inputs to the Verilog-A blocks are the electrical signals applied to the quan-

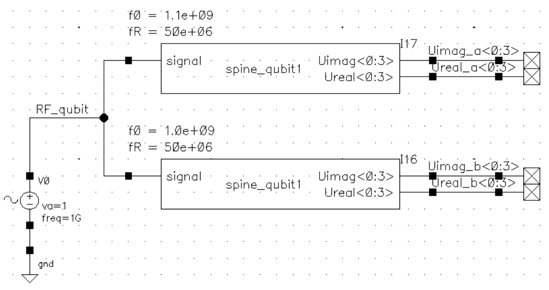

Figure 1: Two modules, each emulating one single-electron spin qubit, have been instantiated as a verilog-A module in the electrical circuit simulator; the two qubits are uncoupled and cannot be entangled.

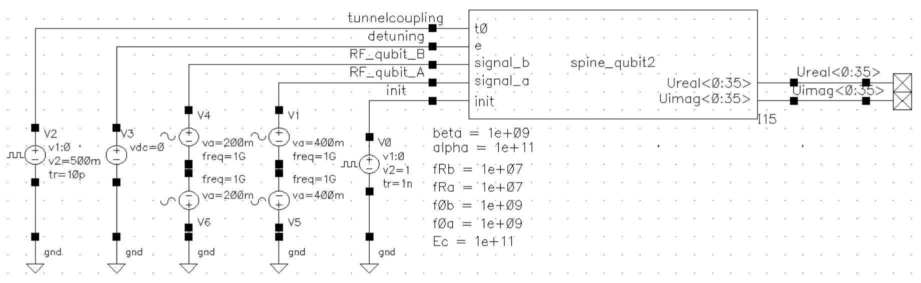

Figure 2: A system of two coupled single-electron spin qubits is included as a verilog-A module in the electrical circuit simulator.

tum processor (Figs. 1 and 2; `signal` is the RF-signal, `e` for detuning the quantum dots and `t0` to control the tunnel coupling) and `init` to reset the operation to the identity matrix. The resulting complex operation U, which can be used to calculate the operation fidelity, is available at the output with separated real (`Ureal<>`) and imaginary (`Uimag<>`) parts in row-major order. Parameters of the physical system are set as a module parameter when instantiating the module in the circuit schematic (Figs. 1 and 2; `Ec` is the charging energy of the quantum dot, `f0` the spin precession frequency, `fR` the rotation frequency at 1-V RF-signal, `beta` the tunnel coupling at 1 V and `alpha` the detuning energy at 1 V).

While the time step control of a transient simulation is managed by the circuit simulator, which relaxes the time step when tolerable, a maximum time step is set by the Verilog-A module to ensure accurate simulation of the quantum physics.

References

[1] J. van Dijk, A. Vladimirescu, M. Babaie, E. Charbon, and F. Sebastiano, "A co-design methodology for scalable quantum processors and their classical electronic interface," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE. IEEE, mar 2018, pp. 573–576. [Online]. Available: https://ieeexplore.ieee.org/document/8342072

[2] J. van Dijk, A. Vladimirescu, M. Babaie, E. Charbon, and F. Sebastiano, "SPINE (SPIN emulator) - a quantum-electronics interface simulator," in *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, IEEE. IEEE, jun 2019, pp. 23–28. [Online]. Available: https://ieeexplore.ieee.org/document/8791334

[3] J. van Dijk, E. Kawakami, R. Schouten, M. Veldhorst, L. Vandersypen, M. Babaie, E. Charbon, and F. Sebastiano, "Impact of classical control electronics on qubit fidelity," *Physical Review Applied*, vol. 12, no. 4, p. 044054, oct 2019. [Online]. Available: https://journals.aps.org/prapplied/abstract/10.1103/PhysRevApplied.12.044054

[4] M. Veldhorst, C. H. Yang, J. C. C. Hwang, W. Huang, J. P. Dehollain, J. T. Muhonen, S. Simmons, A. Laucht, F. E. Hudson, K. M. Itoh, A. Morello, and A. S. Dzurak, "A two-qubit logic gate in silicon," *Nature*, vol. 526, no. 7573, pp. 410–414, oct 2015, letter.

[5] M. Gonzalez-Zalba, S. Barraud, A. Ferguson, and A. Betz, "Probing the limits of gate-based charge sensing," *Nature communications*, vol. 6, p. 6084, 2015.