Software paper for submission to the Journal of Open Research Software

Please submit the completed paper to: editor.jors@ubiquitypress.com

(1) Overview

Title
spinsim: a GPU optimised simulator of spin half and spin one quantum systems

Paper Authors
1. Tritt, Alex;
2. Morris, Joshua;
3. Hochstetter, Joel;
4. Anderson, R. P.;
5. Saunderson, James;
6. Turner, L. D.;

Paper Author Roles and Affiliations
1. School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Primary author of the released packages.
2. School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Present address: Faculty of Physics, University of Vienna, 1010 Vienna, Austria.
Author of first version of code.
3. School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Present address: School of Physics, University of Sydney, NSW 2006, Australia.
Optimization and extension to spin one of first version of code.
4. School of Molecular Sciences, La Trobe University, PO box 199, Bendigo, Victoria 3552, Australia.
Original conception of first version of code.
5. Department of Electrical and Computer Systems Engineering, Monash University, Victoria 3800, Australia.
Advice on numerical analysis.
6. School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Original conception of released version of algorithm.

Abstract
`spinsim` is a *python* package that simulates spin half and spin one quantum mechanical systems following a time dependent Shroedinger equation. It makes use of `numba.cuda` [1], which is an *LLVM* (Low Level Virtual Machine) [2] compiler, and other optimisations, to allow for fast and accurate evaluation on *Nvidia Cuda* [3] compatible systems using GPU parallelisation. `spinsim` is available for installation on *PyPI*, and the source code is available on *github*. The initial use case for the package will be to simulate quantum sensing-based Bose Einstein condensate (BEC) experiments for the Monash University School of Physics and Astronomy spinor BEC lab, but we anticipate it will be useful in simulating any range of spin half or spin

one quantum systems with time dependent Hamiltonians that cannot be solved analytically. These appear in the fields of nuclear magnetic resonance (NMR), nuclear quadrupole resonance (NQR) and magnetic resonance imaging (MRI) experiments and quantum sensing, and with the spin one systems of nitrogen vacancy centres (NVCs) and BECs.

Keywords
Time dependent Schroedinger equation; Spin one; Spin half; Integrator; GPU; Solver; python; numba;

Introduction

Motivation

Ultracold rubidium atoms have proven their effectiveness in state of the art technologies in quantum sensing [4], the use of quantum mechanics to make precise measurements of small signals. The rotation of these atoms can be modelled as quantum spin systems, which is the quantum mechanical model for objects with angular momentum. The simplest spin system, spin half (ie spin quantum number of $\frac{1}{2}$, also referred to as a qubit), is quantised into just two quantum spin levels, and this describes the motion of some fundamental particles such as electrons. However, systems more practical for sensing, such as ultracold rubidium atoms, are more accurately described as a spin one quantum system (ie spin quantum number of 1, also referred to as a quitrit), which is quantised into three quantum spin levels.
The design of sensing protocols requires many steps of verification, including simulation. This is especially important, since running real experiments can be expensive and time consuming, and thus it is more practical to debug such protocols quickly and cheaply on a computer. In general, any design of experiments using spin systems could benefit from a fast, accurate method of simulation.
In the past, the spinor Bose Einstein condensate (spinor BEC) lab at Monash University used an in-house, *cython* based script, on which this package is based, and standard differential equation solvers (such as *Mathematica*'s function `NDSolve`) to solve the Schroedinger equation for quantum sensing spin systems. Spin one systems are sometimes approximated to spin half for a faster execution time, at the cost of modelling all effects of the system. However, these methods are not completely optimised for our use case, and therefore come with some issues.
First, while the execution time for these solvers is acceptable for running a small number of experiments, for certain experiments involving large arrays of independent atom clouds (which require many thousands of simulations to be run), this time accumulates to the order of many hours.
Second, the Schroedinger equation has the geometric property of being norm persevering. In other words, the time evolution operator for a system between two points in time must be unitary. As such, numerical solutions to the Schroedinger equation should also preserve this property. For many numerical methods like those in the Runge Kutta family, the approximations used might not be norm preserving, and the evaluated quantum state may diverge towards an infinite norm, or converge to zero if run for many iterations.

Third, our system (and similar spin systems) can be very oscillatory. In standard conditions for our application, the expected spin projection of a system that we want to solve for can rotate in physical space (alternatively viewed as a point rotating around an abstract object known as a *Bloch sphere*) at a rate of 700kHz. Standard integration methods require very small time steps in order to accurately depict these oscillations.

Implementation and architecture

Mathematical methods

Background

In general, `spinsim` solves the Schroedinger equation,

$$\frac{\mathrm{d}}{\mathrm{d}t}\psi(t) = -iH(t)\psi(t), \tag{1}$$

where $i^2 = -1$, the quantum state $\psi(t) \in \mathbb{C}^N$ a time dependent, N dimensional, unit complex vector, and the Hamiltonian $H(t) \in \mathbb{C}^{N\times N}$ is a time dependent $N \times N$ complex Hermitian matrix. Here N is the number spin levels in the quantum system, so spin half is the $N = 2$ case, and spin one refers to the $N = 3$ case. Rather than being represented by standard coordinates in $\mathbb{C}^{N\times N}$, in `spinsim` the Hamiltonian $H(t)$ is instead represented with respect to a choice of basis for the corresponding Lie Algebra, $\mathfrak{su}(N)$. For example, when set to spin half mode, the `spinsim` package solves the time dependent Schroedinger equations of the form

$$\frac{\mathrm{d}}{\mathrm{d}t}\psi(t) = -i2\pi(f_x(t)J_x + f_y(t)J_y + f_z(t)J_z)\psi(t), \tag{2}$$

where $i^2 = -1$, $\psi(t) \in \mathbb{C}^2$, and the spin half spin projection operators are given by

$$J_x = \frac{1}{2}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, \qquad J_y = \frac{1}{2}\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, \qquad \text{and } J_z = \frac{1}{2}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}. \tag{3}$$

The energy source f that represents the time dependent Hamiltonian $H(t)$, is the collection of energy functions $f_x(t), f_y(t), f_z(t)$, with t in units of s and f in units of Hz that control the dynamics of the system. The user must define a method that returns a sample of these source functions when a sampling time is input. In physical terms, these functions could represent the x, y, z components of a magnetic field applied to a magnetically sensitive.
Similarly, when `spinsim` is set to spin one mode, it can solve the Schroedinger equation of the form

$$\frac{\mathrm{d}}{\mathrm{d}t}\psi(t) = -i2\pi(f_x(t)J_x + f_y(t)J_y + f_z(t)J_z + f_q(t)Q)\psi(t). \tag{4}$$

where now $\psi(t) \in \mathbb{C}^3$, and the spin one operators are given by

$$J_x = \frac{1}{\sqrt{2}}\begin{pmatrix}0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 1 & 0\end{pmatrix}, \qquad J_y = \frac{1}{\sqrt{2}}\begin{pmatrix}0 & -i & 0\\ i & 0 & -i\\ 0 & i & 0\end{pmatrix},$$
$$J_z = \begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & -1\end{pmatrix}, \qquad \text{and } Q = \frac{1}{3}\begin{pmatrix}1 & 0 & 0\\ 0 & -2 & 0\\ 0 & 0 & 1\end{pmatrix}. \tag{5}$$

The matrices J_x, J_y, J_z are regular spin operators, and Q is a quadrupole operator. Note that Q is proportional to Q_{zz} as defined by Hamley et al [5], and Q_0 as defined by Di et al [6].
Frequently when one calculates the times series of the quantum state $\psi(t)$, they are also interested in its expected spin projection $\langle J\rangle$(t). It is given by the vector

$$\langle J\rangle(t) = \begin{pmatrix}\psi(t)^\dagger J_x \psi(t)\\ \psi(t)^\dagger J_y \psi(t)\\ \psi(t)^\dagger J_z \psi(t)\end{pmatrix}, \tag{6}$$

where $\cdot^\dagger$ is the adjoint operator (also referred to as Hermitian conjugate, and is implemented as complex conjugate transpose). The expected spin projection can be interpreted as the average direction that the spin system is oriented in space when many systems of the equivalent state are measured (due to the Heisenberg uncertainty principle, only one component of the exact orientation of a quantum state can be known at any point in time, which is why we need to deal with expected values rather than absolute values). However, it can also be used to model the overall orientation of an ensemble of many quantum systems, like a BEC, for instance. `spinsim` has the functionality to calculate the expected spin projection of a system from its state.

Parallelisation

Given that $\psi(t)$ is a unit vector, it is possible to write $\psi(t)$ in terms of a unitary transformation $U(t, t_0)$ of the state $\psi(t_0)$, for any time t_0. In other words, for any $t_0, t \in \mathbb{R}$, there is a unitary transformation $U(t, t_0)$ such that

$$\psi(t) = U(t, t_0)\psi(t_0). \tag{7}$$

This means that a time series for the state of the system can be evaluated by evaluating the time evolution operator between each of the sample times. Consider quantising time with

$$t_k = t_0 + \mathrm{D}t \cdot k, \tag{8}$$

where $\mathrm{D}t$ is the time step of the time series (in contrast to $\mathrm{d}t$, a smaller time step for integration). If we define

$$\psi_k = \psi(t_k) \text{ and} \tag{9}$$
$$U_k = U(t_k, t_{k-1}), \tag{10}$$

then the time series of states ψ_k and time evolution operators U_k satisfies

$$\psi_k = U_k \psi_{k-1} \tag{11}$$

This presents an opportunity for parallelism. While each of the ψ_k must be evaluated sequentially, the value of the U_k is independent of the value of any ψ_{k_0}, or any other U_{k_0}. This means that the time evolution operators U_k can all be calculated in parallel, and it allows `spinsim` to use GPU parallelisation on the level of time sample points, so a speed up is achieved even if just a single simulation is run.
In summary, `spinsim` splits the time evolution of the full simulation into time evolution U_k within small time intervals $[t_{k-1}, t_k]$, which are each calculated massively in parallel on a GPU. When all the U_k are calculated, the CPU then multiplies the U_k together (a comparatively less demanding job than calculating them) using Equation (11) to determine the ψ_k.

Rotating frame

If the rotating frame option is selected, the U_k are first calculated within a rotating frame of reference as U_k^r, which, in some situations, reduces the size of the source functions used in the calculation, increasing accuracy. The rotation speed of the rotating frame is calculated locally for each parallel time step U_k, and only for rotations around the z axis. The rotating from source functions f_x^r, f_y^r, f_z^r, and f_q^r are related to the source function from the user input via

$$f_x^r(t) + i f_y^r(t) = e^{-i2\pi f_r t}(f_x(t) + i f_y(t)), \tag{12}$$
$$f_z^r(t) = f_z(t) - f_r, \text{ and} \tag{13}$$
$$f_q^r(t) = f_q(t), \text{ for spin one.} \tag{14}$$

Where $f_r = f_z(t_k + \frac{1}{2}\mathrm{D}t)$ is sampled the midpoint value of the source over the interval $[t_{k-1}, t_k]$. This, assuming that a midpoint sample is representative of an average value over the time interval, decreases the magnitude of $f_z(t)$, while leaving the other source components at an equivalent magnitude. The rotation is then applied to obtain the lab frame time evolution operator U_k via

$$U_k = \exp(-i2\pi f_r J_z \mathrm{D}t) U_k^r. \tag{15}$$

Specifically, this relationship is

$$U_k = \begin{pmatrix} e^{-i2\pi f_r \mathrm{D}t} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & e^{i2\pi f_r \mathrm{D}t} \end{pmatrix} U_k^r, \text{ for spin one, and} \tag{16}$$

$$U_k = \begin{pmatrix} e^{-i\pi f_r \mathrm{D}t} & 0 \\ 0 & e^{i\pi f_r \mathrm{D}t} \end{pmatrix} U_k^r, \text{ for spin half.} \tag{17}$$

It is a common technique in solving quantum mechanical problems to enter rotating frames, and their more abstract counterparts of interaction pictures [7]. Note that this is typically done in conjunction with the Rotating Wave Approximation (RWA), which is an assumption that the oscillatory components of f_x^r, f_y^r, f_z^r, and f_q^r on an average of many cycles do not have a large contribution to time evolution of the solution and can be ignored. In some cases, this allows for analytic solutions to the approximate quantum system to be obtained. Note that the RWA is *not* invoked in `spinsim`, as doing this would reduce the accuracy of simulation results, defeating our purpose of using a rotating frame.

Magnus based integration method

The integration method used in `spinsim` is the CF4 method from Auer et al [8]. Each of the U_k are split into products of time evolution operators between times separated by a smaller time step, that is,

$$U(t_k, t_{k-1}) = U(t_k, t_k - \mathrm{d}t) \cdots U(t_{k-1} + 2\mathrm{d}t, t_{k-1} + \mathrm{d}t) U(t_{k-1} + \mathrm{d}t, t_{k-1}) \tag{18}$$

$$U_k = u_{L-1}^k \cdots u_0^k, \text{ where} \tag{19}$$

$$u_{L-1}^k = U(t_0 + (k-1)\mathrm{D}t + (l+1)\mathrm{d}t, t_0 + (k-1)\mathrm{D}t + l\mathrm{d}t) \tag{20}$$

with dt being the integration level time step. Note that the time steps are related to each other by D$t = L$dt, where $L \in \mathbb{N}$.
The CF4 method is used to calculate each individual u_l^k. Let the fine sample time be given by $t_f = l\mathrm{d}t + t_k$. Then as part of the CF4 method, the source functions are sampled at particular times based on the second order Gauss-Legendre quadrature, given by

$$t_1 = t_f + \frac{1}{2}\mathrm{d}t \left(1 - \frac{1}{\sqrt{3}}\right), \text{ and} \tag{21}$$

$$t_2 = t_f + \frac{1}{2}\mathrm{d}t \left(1 + \frac{1}{\sqrt{3}}\right). \tag{22}$$

Now, let

$$f(t_1) = (f_x(t_1), f_y(t_1), f_z(t_1), f_q(t_1)), \text{ and} f(t_2) = (f_x(t_2), f_y(t_2), f_z(t_2), f_q(t_2)). \tag{23}$$

The integration time evolution operator can then be calculated using

$$
\begin{aligned}
g_1 =& (g_{1,x}, g_{1,y}, g_{1,z}, g_{1,q}) && (24)\\
=& 2\pi \mathrm{d}t \left(\frac{3+2\sqrt{3}}{12} f(t_1) + \frac{3-2\sqrt{3}}{12} f(t_2) \right), \text{ and} && (25)\\
g_2 =& (g_{2,x}, g_{2,y}, g_{2,z}, g_{2,q}) && (26)\\
=& 2\pi \mathrm{d}t \left(\frac{3-2\sqrt{3}}{12} f(t_1) + \frac{3+2\sqrt{3}}{12} f(t_2) \right), \text{ so} && (27)\\
u =& \exp(-i\,(g_{2,x}J_x + g_{2,y}J_y + g_{2,z}J_z + g_{2,q}Q)) && (28)\\
& \cdot \exp(-i\,(g_{1,x}J_x + g_{1,y}J_y + g_{1,z}J_z + g_{1,q}Q)). && (29)
\end{aligned}
$$

Exponentiator

For all exponentiation, the exponentiator computes the matrix exponential

$$
\begin{aligned}
E(g) &= E(g_x, g_y, g_z, g_q) && (30)\\
&= \exp(-i(g_x J_x + g_y J_y + g_z J_z + g_q Q)), \text{ with} && (31)
\end{aligned}
$$

For spin half, the default exponentiator is in an analytic form. For spin one, a simple analytic form cannot be written, so instead an exponentiator based on the Lie Trotter product formula [9] is used. Explicitly, this formula is

$$
\exp(A+B) = \lim_{n\to\infty} \left(\exp\left(\frac{A}{n}\right) \exp\left(\frac{B}{n}\right) \right)^n \qquad (32)
$$

Where $A, B \in \mathbb{C}^{N\times N}$ are matrix operators, and $n \in \mathbb{N}$. The equality holds for without the limit for all n if the operators commute, like it does for real and complex numbers. This tells us that we can approximate a matrix exponential of a linear combination of spin operators by reducing the size of the coefficients of these operators, taking the analytically known exponentials of each of the operators separately, multiplying each individual result together, and then raising the combination T to the power of the original reduction. For our spin one case, the exponential $E(g)$ can be approximated as, for large 2^τ,

$$
\begin{aligned}
E(g) =& \exp\left(-ig_xJ_x - ig_yJ_y - ig_zJ_z - ig_qQ\right) && (33)\\
=& \exp\left(2^{-\tau}\left(-ig_xJ_x - ig_yJ_y - ig_zJ_z - ig_qQ\right)\right)^{2^\tau} && (34)\\
\approx& \Bigg(\exp\left(-i\left(2^{-\tau}\frac{1}{2}g_zJ_z + \left(2^{-\tau}\frac{1}{2}g_q\right)Q\right)\right)\\
&\cdot \exp\left(-i\left(2^{-\tau}g_\phi J_\phi\right)\right)\\
&\cdot \exp\left(-i\left(2^{-\tau}\frac{1}{2}g_zJ_z + \left(2^{-\tau}\frac{1}{2}g_q\right)Q\right)\right)\Bigg)^{2^\tau} && (35)\\
=& \begin{pmatrix} \left(\frac{\Gamma}{PZ}\right)^2 & \frac{SP}{Z\Phi} & -\left(\frac{\Sigma P}{\Phi}\right)^2 \\ \frac{SP\Phi}{Z} & CP^4 & \frac{SPZ}{\Phi} \\ -\left(\frac{\Sigma\Phi}{P}\right)^2 & \frac{SPZ\Phi}{1} & \left(\frac{\Gamma Z}{P}\right)^2 \end{pmatrix}^{2^\tau} && (36)\\
=& T^{2^\tau}. && (37)
\end{aligned}
$$

where

$$
\begin{aligned}
g_\phi &= \sqrt{g_x^2 + g_y^2}\\
J_\phi &= \frac{g_x}{g_\phi}J_x + \frac{g_y}{g_\phi}J_y\\
\Gamma &= \cos\left(2^{-\tau}\frac{g_\phi}{2}\right) & \Phi &= e^{i2^{-\tau}g_\phi}\\
\Sigma &= \sin\left(2^{-\tau}\frac{g_\phi}{2}\right) & Z &= e^{i2^{-\tau}\frac{g_z}{2}}\\
C &= \cos(2^{-\tau}g_\phi) & P &= e^{i2^{-\tau}\frac{g_q}{6}}\\
S &= \frac{-i}{\sqrt{2}}\sin(2^{-\tau}g_\phi) && (38)
\end{aligned}
$$

Once T is calculated, it is then recursively squared τ times to obtain $E(g)$. The approach used for spin one exponentiation means that the package cannot solve arbitrary spin one quantum systems, as that would require the ability to exponentiate a point in the full, 8 dimensional Lie algebra of $\mathfrak{su}(3)$, rather than just the four dimensional subspace spanned by the subalgebra $\mathfrak{su}(2)$ spanned by $\{J_x, J_y, J_z\}$, and the single quadratic operator Q. Including the full algebra could be possible as a feature update if there is demand for it, though just including this subspace is sufficient for our application, and many others, and has the advantage of being able to use this faster, more specialised method of matrix exponentiation.
Note that, the methods for both spin half and spin one use analytic forms of exponentials to construct the result, meaning that all calculated time evolution operators are unitary. This guarantees that the results of `spinsim` maintain unitary.

Software architecture

Integrator architecture

The integrator in the `spinsim` package calls a `numba.cuda.jit()`ed kernel to be run on a *Cuda* capable *Nvidia* GPU in parallel, with a different thread being allocated to each of the U_k. This returns when each of the U_k have been evaluated.
The thread starts by calculating t_k and, if the rotating frame is being used, f_r. The latter is done by sampling a (`numba.cuda.jit()`ed version of a) user provided *python* function f describing how to sample the source Hamiltonian. The code then loops over each integration time step $\mathrm{d}t$ to calculate the integration time evolution operators u_l^k.
Within the loop, the integrator enters a device function (ie a GPU subroutine, which is inline for speed) to sample $f(t)$, as well as calculate $e^{-i2\pi f_r t}$, at the sample times needed for the integration method. After this, it enters a second device function, which makes a rotating wave transformation as needed in a third device function, before calculating g values, and finally taking the matrix exponentiation in a fourth device function. u_l^k is premultiplied to U_k^r (which is initialised to 1), and the loop continues.
When the loop has finished, if the rotating frame is being used, U_k^r is transformed to U_k as in Equation (11), and this is returned. Once all threads have executed, the state ψ_k is calculated in a (CPU) `numba.jit()`ed function from the U_k and an initial condition ψ_{init}.

Compilation of integrator

The `spinsim` integrator is constructed and compiled just in time, using `numba.cuda.jit()`. The particular device functions used are not predetermined, but are instead chosen based on user input to decide a closure. This structure has multiple advantages. First, the source function f is provided by the user as a plain python function (that must be `numba.cuda.jit()` compatible). This allows users to define f in a way that compiles and executes fast, does not put many restrictions on the form of the function, and returns the accurate results of analytic functions (compared to the errors seen in interpolation). Compiling the simulator also allows the user to set meta parameters, and choose the features they want to use, in a way that does not require experience with the `numba.cuda` library. This was especially useful for running benchmarks comparing old integration methods to the new ones, like CF4. The default settings should be optimal for most users, although tuning the values of *Cuda* meta parameters `max_registers` and `threads_per_block` could improve performance for GPUs with a differing number of registers and *Cuda* cores to the mobile GTX1070 mainly used in testing here. Finally, just in time compilation also allows the user to select a target device other than *Cuda* for compilation, so the simulator can run, using the same algorithm, on a multicore CPU in parallel instead of a GPU, if the user so chooses.
This functionality is interfaced through an object of class `spinsim.Simulator`. The *Cuda* kernel is defined as per the user's instructions on construction of the instance, and it is used by calling the method `spinsim.Simulator.evaluate()`,

which returns a results object including the time, state, time evolution operator, and expected spin projection (that is, Bloch vector). Note that the expected spin projection is calculated as a lazy parameter if needed, rather than returned by the simulator object.

Quality control

Benchmarks

Speed

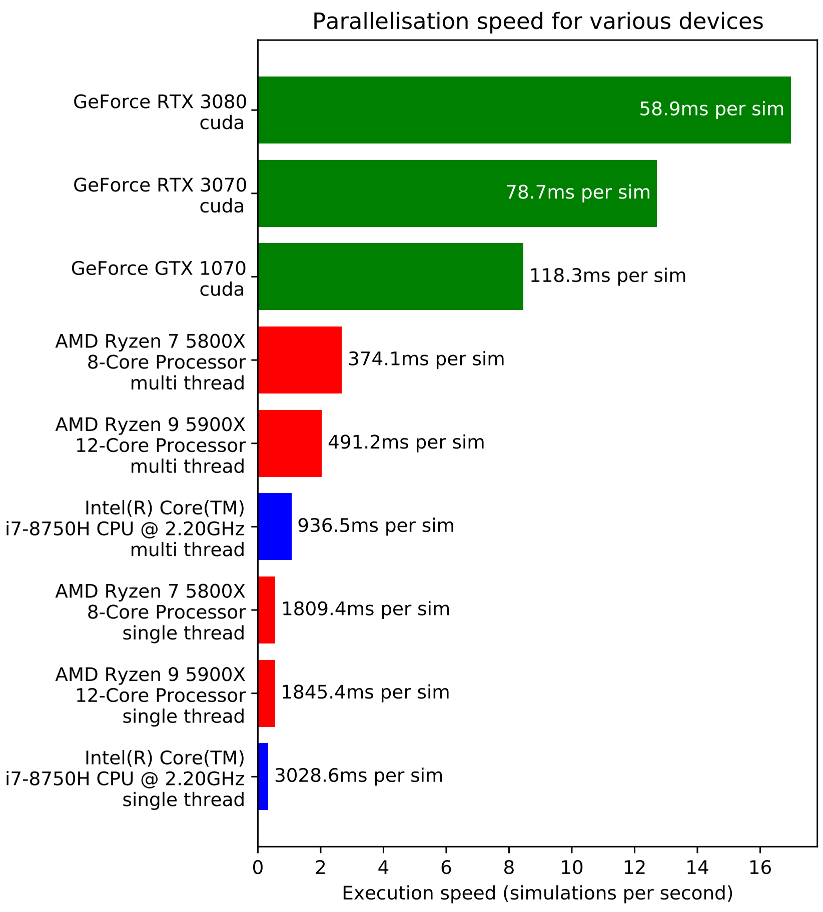


Figure 1: Evaluation speed of a typical spin one sensing experiment. Fine time step is 100ns, as determined to be ideal by the accuracy experiments. Experiments run for a duration of 100ms. Evaluation time is determined by an average of 100 similar experiments for each device.

Benchmarks were performed using `sense.sim.benchmark`, by comparing evaluation speed of typical spin one sensing experiments on different devices. This is shown in Figure 1. The integration code was compiled by `numba` for single core CPUs, multicore CPUs, and *Nvidia Cuda*, and run on different models of each of them. These test devices are,

- Computer A
  - Intel Core i7-8750H, a 6 core laptop processor released in 2018. Run with 16GiB of RAM. Air cooled (laptop fan). Base clock speed of 2.2GHz.
  - Nvidia GeForce GTX 1070, a 2048 *Cuda* core laptop graphics processor released in 2016. Run with 8GiB of VRAM. Air cooled (laptop fan).
- Computer B
  - AMD Ryzen 9 5900X, a 12 core desktop processor released in 2020. Run with 32GiB of RAM. Air cooled. Base clock speed of 3.7GHz.
  - Nvidia GeForce RTX 3070, a 5888 *Cuda* core desktop graphics processor released in 2020. Run with 8GiB of VRAM. Air cooled.
- Computer C
  - AMD Ryzen 7 5800X, an 8 core desktop processor released in 2020. Run with 32GiB of RAM. Liquid cooled. Base clock speed of 3.8GHz.
  - Nvidia GeForce RTX 3080, an 8704 *Cuda* core desktop graphics processor released in 2020. Run with 10GiB of VRAM. Air cooled.

This benchmark shows the benefit to using parallelisation when solving this problem. Moving from a 6 core processor to a 12 core processor doubles the execution speed. Moving from a single core processor to a GPU increases performance by well over an order of magnitude. As an aside, liquid cooling allows the 8 core processor to increase its boost clock and outperform the 12 core processor.

Evaluation of techniques

All subsequent benchmarks were run on computer C, which from Figure 1 has the fastest CPU (liquid cooled AMD Ryzen 7 5800X) and GPU (Nvidia GeForce RTX 3080). Benchmarks were performed using `neural-sense.sim.benchmark` (where `neural-sense` [10] is the quantum sensing package that `spinsim` was written for). We first wanted to test the accuracy of the different integration techniques for various integration time steps. This accuracy was calculated by taking state evaluations of a typical quantum sensing experiment and finding the Root Mean Squared (RMS) to a baseline simulation run by `scipy.integrate.ivp_solve()` as part of the *SciPy* python package, via

$$\epsilon = \frac{1}{K}\sqrt{\sum_{k=0}^{K-1}\sum_{m_j=-j}^{j}|\psi_{k,(m_j)} - \psi^{\text{baseline}}_{k,(m_j)}|^2}, \tag{39}$$

where $j \in \{\frac{1}{2}, 1\}$ is the spin quantum number of the system. This quantum sensing experiment involves continuously driving transitions in the system, while exposing

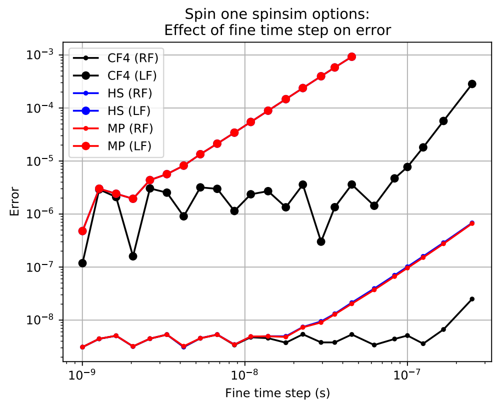
it to a short pulsed signal that the system should be able to sense. It runs over a duration of 1ms.
This baseline was computed in 3.8 hours, and was also used for comparisons to other software packages. These are shown in Figures 2a, 3a, and 3c. For each of these simulations we measured the time of execution, as although it is more accurate compared to Euler integration, the CF4 method is slower for any fixed integration time step. These are shown in Figures 2b, 3b, and 3d. In all of these comparisons, errors above 10^{-3} were counted as a failed simulation, as the maximum possible error for a quantum state saturates given that it is a point on a unit complex sphere.
The integration techniques tested were

- The Magnus based method Commutator Free 4. Abbreviated as CF4.
- The Euler based half step method used in the previous version of our code, where two Euler steps are made in each iteration, one sampled from the left, and one from the right. Abbreviated as HS.
- The Euler based midpoint method, where a single sample is made from the midpoint of the interval being stepped over. Abbreviated as MP.

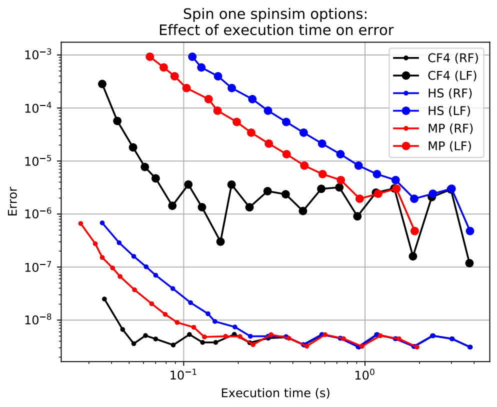
We benchmarked these methods both while using the rotating frame option (abbreviated as RF), and not (abbreviated as LF, for lab frame). The spin half simulations were benchmarked using both the analytic based exponentiator,and the Lie Trotter based exponentiator. This was done separately for spin one and spin half systems, to ensure they all yield accurate results.
From Figures 2 and 3, we find that overall, the results that `spinsim` gives are accurate to those of *SciPy*. Figures 2a, 3a, and 3c show that using the Magnus based integration method is up to 3 orders of magnitude more accurate when compared the Euler based methods. Also, using the rotating frame increased the accuracy here by 4 orders of magnitude for any individual integration method. From Figures 2b, 3b, and 3d, although the Magnus based method is slower than the midpoint Euler based method, it makes up for this in terms of its accuracy. Thus, by default `spinsim` sets the integrator to CF4, and uses the rotating frame. These can be modified using optional arguments when instantiating the `spinsim.Simulator` object.
The that the accuracy of spin half simulations using the Lie Trotter based exponentiator in Figure 3c plateaus below 10^{-8}, whereas an accuracy below 10^{-12} is able to be obtained when when the analytic exponentiator in Figure 3a is used. This also explains the plateau in accuracy from the spin one integrator, where the Lie Trotter based method is the only method implemented. Furthermore, the analytic exponentiator is much faster in Figure 3b, as compared to the Lie Trotter based integrator in Figure 3d. Hence, the analytic exponentiator is the default option for `spinsim` simulations when in spin half mode.

Comparison to alternatives

We ran the same error (using Equation (39)) and execution time benchmarks on some alternative packages to compare `spinsim`'s performance to theirs. The packages compared were:



(a) Accuracy of the spin one options of `spinsim`.

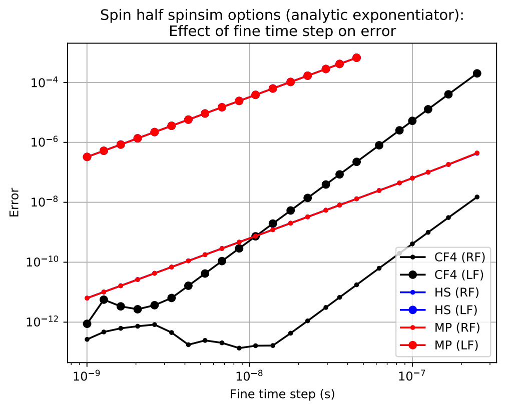


(b) Speed vs accuracy of the spin one options of `spinsim`.

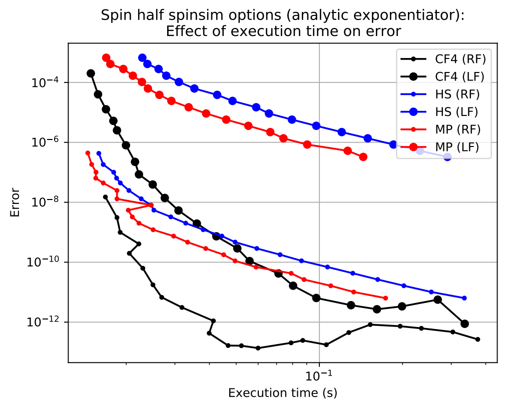
Figure 2: Speed and accuracy of the spin one options of `spinsim`.

- `spinsim` running on the *Cuda* device, using the CF4 integrator and the rotating frame mode, which were the highest performing options when comparing `spinsim` integration methods. Abbreviated as ss.
- `NDSolve` from the *Mathematica* [11] software. This was chosen as it is popular with our lab group for simulating magnetometry experiments. Abbreviated as mm.
- `scipy.integrate.ivp_solve()` from the *python* library `SciPy` [12]. This was chosen as a generic solver from within the python ecosystem. Abbreviated as sp.
- We had also planned to benchmark against `qutip.sesolve()`, a solver in the popular quantum mechanics *python* library, `QuTip` [13]. However, due to a known bug with the library's dependencies, this was not installable on Windows 10, the operating system being used for testing, and so benchmarks for it could not be run.

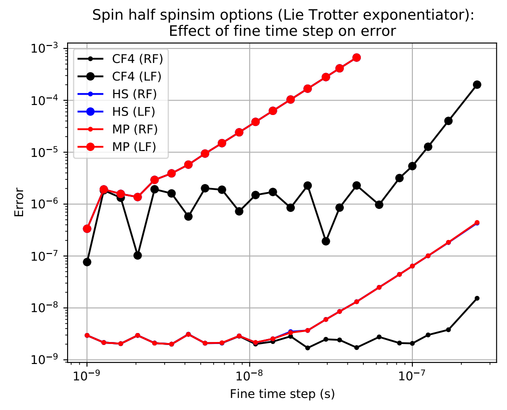
In each case, the step sizes of the alternative integrators were limited to a maximum value obtain simulation results of different accuracies. Apart from that, the integrator settings were left untouched from the default values, as a representation of what a user would experience using a generic solver for spin system problems. Similarly to with the internal `spinsim` benchmarks, the expected spin projection was evaluated in each case, but the states were compared to calculate a relative error. Also like with the internal benchmarks, we used the longest running *SciPy* simulation as a ground truth for comparison, as both *Mathematica* and `spinsim` plateau in accuracy at small time steps.
Each benchmark was run one simulation at a time, however it might be possible to increase the average speed of many benchmarks from alternative packages using multithreading to run multiple benchmarks at a time. When this was attempted using *Mathematica*, the kernels crashed as the 32GiB of RAM was not enough to



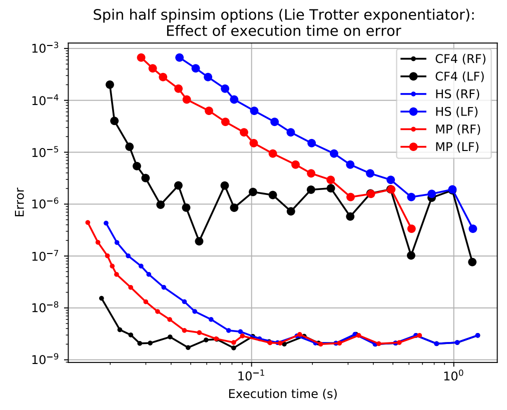
(a) Accuracy of the spin half options of `spinsim`, using the analytic exponentiator.



(b) Speed vs accuracy of the spin half options of `spinsim`, using the analytic exponentiator.



(c) Accuracy of the spin half options of `spinsim`, using the Lie Trotter exponentiator.



(d) Speed vs accuracy of the spin half options of `spinsim`, using the Lie Trotter exponentiator.

Figure 3: Speed and accuracy of the spin half options of `spinsim`.

run them all at once. Multithreading was also not attempted using *SciPy*, due to the fact that running the benchmark only single threaded uses up a day of computational time. But to be fair, both *Mathematica* and *SciPy* benchmarks are plotted with an artificial reduction in execution time by a factor of 4 and 8, which is an upper bound for the speed increase that could be obtained by running them parallel on a 4 and 8 core processor, respectively. These are respectively shown in plots as disconnected $+$ and $\times$.

From Figure 4, if an error of 10^{-8} is acceptable, `spinsim` is 3 orders of magnitude faster than *Mathematica*, and 4 orders of magnitude faster than *SciPy*. In practice, this means that an 8 minute *SciPy* simulation is reduced to 50ms, and a full week long batch simulation of 1000 separate systems (a realistic situation for quantum sensing protocols) would take less than one minute. With that said, if a smaller error is needed, it also plateaus at a greater minimum error than its alternatives, with *Mathematica* having a minimum error less than 10^{-10}, and *SciPy* not plateauing during the benchmark. However, for the vast majority of use cases, an error of 10^{-8}

is acceptable, and `spinsim` has vastly better performance in terms of speed and accuracy.

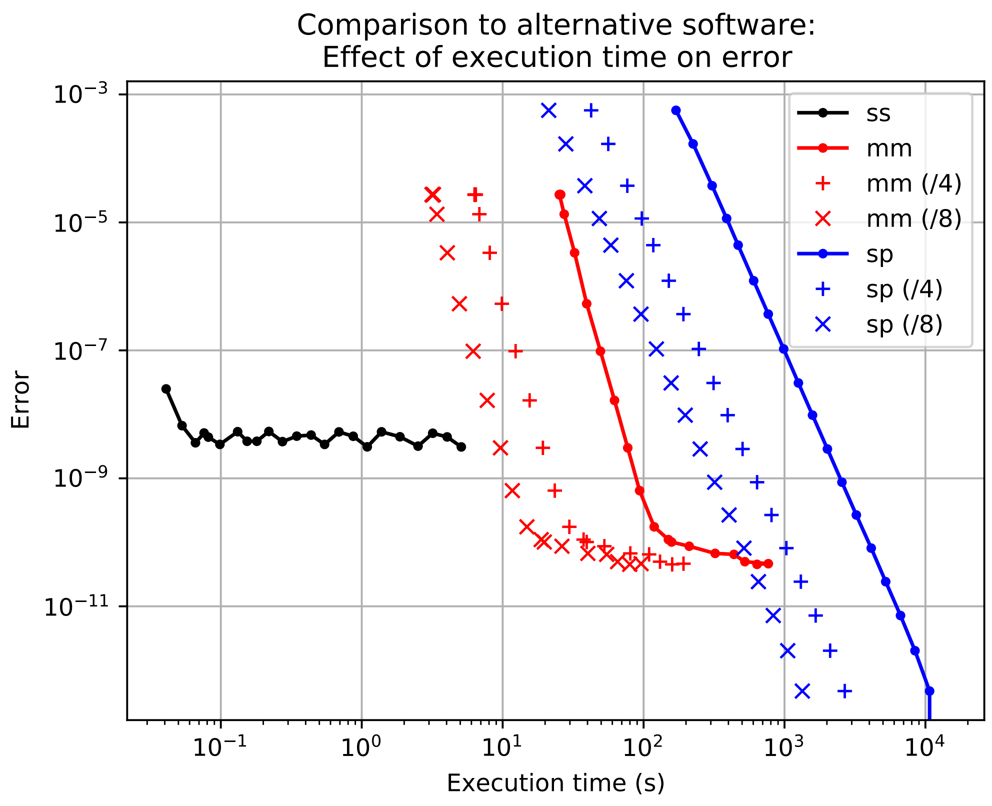


Figure 4: Speed vs accuracy of various integration packages.

Testing

During the accuracy tests, it was confirmed that all possible modes of `spinsim` agree with a standard *SciPy* simulation up to a small error. The Lie Trotter matrix exponentiator was tested separately from the full system, as well as benchmarked separately. These tests and benchmarks were run as part of the `neural_sense` package. The simulator has also been used as part of the measurement protocol being developed there, and it has been tested as part of those algorithms as well.
The kernel execution was profiled thoroughly, and changes were made to optimise VRAM and register usage and transfer. This was done specifically for the development hardware of computer A with an *Nvidia* GTX1070, so one may get some performance increases by changing some GPU specific meta parameters when instantiating the `spinsim.Simulator` object.
A good way to confirm that `spinsim` is functioning properly after an installation is to run the tutorial code provided and compare the outputs. Otherwise, one can reproduce the benchmarks shown here using `neural_sense.sim.benchmark`.

(2) Availability

Operating system

Developed and tested on Windows 10. CPU functionality tested on MacOS Big Sur (note that modern Mac computers are not compatible with *Cuda* software). All packages referenced in `spinsim` are compatible with Linux, but functionality has not been tested.

Programming language

Python (3.7 or greater)

Additional system requirements

To use the (default) *Nvidia Cuda* GPU parallelisation, one needs to have a *Cuda* compatible *Nvidia* GPU [14]. For *Cuda* mode to function, one also needs to install the *Nvidia Cuda* toolkit [15]. If *Cuda* is not available on the system, the simulator will automatically parallelise over multicore CPUs instead.

Dependencies

numba (0.50.1 or greater)
numpy (1.19.3)
matplotlib (for example code, 3.2)
neuralsense (for benchmark code)

List of contributors

1. Alex Tritt
School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Primary author of the released packages.
2. Joshua Morris
School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Present address: Faculty of Physics, University of Vienna, 1010 Vienna, Austria.
Author of first version of code.
3. Joel Hockstetter
School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Present address: School of Physics, University of Sydney, NSW 2006, Australia.
Optimization and extension to spin one of first version of code.
4. Russell P. Anderson
School of Molecular Sciences, La Trobe University, PO box 199, Bendigo, Victoria 3552, Australia.
Original conception of first version of code.
5. James Saunderson
Department of Electrical and Computer Systems Engineering, Monash University, Victoria 3800, Australia.
Advice on numerical analysis.
6. Lincoln D. Turner
School of Physics & Astronomy, Monash University, Victoria 3800, Australia.
Original conception of released version of algorithm.

Software location:
Archive

Name: Monash Bridges
Persistent identifier: 10.26180/13285460
Licence: Apache 2.0
Publisher: Alex Tritt
Version published: 1.0.0
Date published: dd/mm/yy

Code repository

Name: GitHub
Persistent identifier: https://github.com/alexander-tritt-monash/spinsim
Licence: BSD 3 Clause
Date published: 18/11/20

Language
English.

(3) Reuse potential

Use potential and limitations

`spinsim` will be useful for any research group needing quick, accurate, and / or large numbers of simulations involving spin half or spin one systems. This is immediately relevant to developing new quantum sensing protocols with spin half and spin one systems. This package is being used in the context of Bose Einstein Condensate (BEC) magnetic sensing protocol design by our lab.
This project is to be able to measure neural signals using BECs. The electrical pulses made by neurons are currently measured using electrical probes, which is intrusive and damages the cells. We instead propose to sense the small magnetic fields that these electrical currents produce. Rubidium BECs can potentially be made sensitive enough to these tiny magnetic fields that they can be measured by them. `spinsim` was written to simulate possible measurement protocols for this, showing the behaviour of the array of spin one atoms interacting with the magnetic fields of the neurons, control signals, and noise. The package is also now being used to simulate other BEC magnetometry experiments by the lab group.
Another example of spin based magnetic field sensing is the use of Nitrogen Vacancy Centres (NVCs). These are spin one structures found in diamond doped with Nitrogen atoms. This leaves a vacancy in a position adjacent to the Nitrogen atom, which pairs of electrons occupy to obtain the spin one properties. Similar to BECs, NVCs can be placed and addressed in 2D arrays in order to take many samples in one measurement. A paper was only recently released covering simulation experiments of magnetic neural pulse sensing using NVCs [16], which is something that `spinsim` could be useful for.
`spinsim` is designed to simulate small dimensional quantum systems, including large arrays of non-interacting spin systems. This means that it would not be able to integrate large arrays of entangled states or interacting particles. As a result, despite being fast at simulating qubits, it is inappropriate for the package to be used

for quantum computing. In addition, `spinsim` is currently designed to integrate the time evolution of pure states only. This means that it may not be adequate for use in some Nuclear Magnetic Resonance (NMR) applications where relaxation [17] is important (or other kinds of simulations involving decoherence).
With these restrictions in mind, `spinsim` could be used for some simplified simulations in various areas of NMR. There are many atomic nuclei with spins of half (eg protons, Carbon 13) and, and fewer that have spins of one (eg Lithium 6, Nitrogen 14) [18], which, if relaxation and interactions between systems are not important for the application, `spinsim` could be used to simulate for spectroscopy experiments, for example. The inclusion of a quadrupole operator means that, with the same level of simplifications, `spinsim` should be able to simulate Nuclear Quadrupole Resonance (NQR) spectroscopy for spin one nuclei [19], such as Nitrogen 14, provided a suitable coordinate system is chosen. This technique measures energy level differences between levels split by electric field gradients, rather than static magnetic bias fields. Another possible use case could be for Magnetic Resonance Imaging (MRI) simulation and pulse sequence design. MRI uses measures the response of spins of an array of spin half protons to a spatially varying pulse sequence [20], which essentially just corresponds to many separate `spinsim` simulations of spins at different positions in space. While this package offers some advantages over state of the art simulators in the field [21], with its use of quantum mechanics over classical mechanics, and its absence of rotating wave approximations, its parametrised pulse sequence definitions and geometric integrator, again, the lack of interacting particles and decoherence features are may limit its use in this area.

Support

Documentation for `spinsim` is available on *Read the Docs*. This documentation contains a thorough tutorial on how to use the package, and installation instructions. For direct support with the `spinsim` package, one can open an issue in the *github* repository. One can also use this contact to suggest extensions to the package. `spinsim` is planned to be maintained by the Monash University spinor BEC lab into the future.

Acknowledgements
Thank you to the Monash University School of Physics and Astronomy spinor BEC lab group, particularly Hamish Taylor and Travis Hartley, who have started using `spinsim` for their own projects and have given useful feedback of their user experience with the package.

Funding statement
If the software resulted from funded research please give the funder and grant number.

Competing interests
The authors declare that they have no competing interests.

References

[1] Lam SK, Pitrou A, Seibert S. Numba: a LLVM-based Python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15. Austin, Texas: ACM Press; 2015. p. 1–6. Available from: `http://dl.acm.org/citation.cfm?doid=2833157.2833162`.

[2] Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004.; 2004. p. 75–86.

[3] Nickolls J, Buck I, Garland M, Skadron K. Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? Queue. 2008 Mar;6(2):40–53. Available from: `https://doi.org/10.1145/1365490.1365500`.

[4] Degen CL, Reinhard F, Cappellaro P. Quantum sensing. Reviews of Modern Physics. 2017 Jul;89(3):035002. Available from: `https://link.aps.org/doi/10.1103/RevModPhys.89.035002`.

[5] Hamley CD, Gerving CS, Hoang TM, Bookjans EM, Chapman MS. Spin-nematic squeezed vacuum in a quantum gas. Nature Physics. 2012 Apr;8(4):305–308. Number: 4 Publisher: Nature Publishing Group. Available from: `https://www.nature.com/articles/nphys2245`.

[6] Di Y, Wang Y, Wei H. Dipole–quadrupole decomposition of two coupled spin 1 systems. Journal of Physics A: Mathematical and Theoretical. 2010 Jan;43(6):065303. Publisher: IOP Publishing. Available from: `https://doi.org/10.1088%2F1751-8113%2F43%2F6%2F065303`.

[7] J J Sakurai (Jun John). Modern quantum mechanics. Rev. ed. ed. Reading, Mass.: Addison-Wesley PubCo; 1994.

[8] Auer N, Einkemmer L, Kandolf P, Ostermann A. Magnus integrators on multicore CPUs and GPUs. Computer Physics Communications. 2018 Jul;228:115–122. Available from: `http://www.sciencedirect.com/science/article/pii/S0010465518300584`.

[9] Moler C, Van Loan C. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. SIAM Review. 2003;45(1):3–49. Publisher: Society for Industrial and Applied Mathematics. Available from: `https://www.jstor.org/stable/25054364`.

[10] alexander-tritt monash. alexander-tritt-monash/neural-sense; 2020. Original-date: 2020-10-06T00:58:21Z. Available from: `https://github.com/alexander-tritt-monash/neural-sense`.

[11] Wolfram Research I. Mathematica. Champaign, Illinois: Wolfram Research, Inc.; 2020. Available from: `https://www.wolfram.com/mathematica`.

[12] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature Methods. 2020 Mar;17(3):261–272. Number: 3 Publisher: Nature Publishing Group. Available from: `https://www.nature.com/articles/s41592-019-0686-2`.

[13] Johansson JR, Nation PD, Nori F. QuTiP 2: A Python framework for the dynamics of open quantum systems. Computer Physics Communications. 2013 Apr;184(4):1234–1240. Available from: `http://www.sciencedirect.com/science/article/pii/S0010465512003955`.

[14] CUDA GPUs; 2012. Available from: `https://developer.nvidia.com/cuda-gpus`.

[15] CUDA Toolkit; 2013. Available from: `https://developer.nvidia.com/cuda-toolkit`.

[16] Parashar M, Saha K, Bandyopadhyay S. Axon hillock currents enable single-neuron-resolved 3D reconstruction using diamond nitrogen-vacancy magnetometry. Communications physics. 2020;3(1):174.

[17] Veshtort M, Griffin RG. SPINEVOLUTION: A powerful tool for the simulation of solid and liquid state NMR experiments. Journal of Magnetic Resonance. 2006 Feb;178(2):248–282. Available from: `http://www.sciencedirect.com/science/article/pii/S1090780705002442`.

[18] Fuller GH. Nuclear Spins and Moments. Journal of Physical and Chemical Reference Data. 1976 Oct;5(4):835–1092. Publisher: American Institute of Physics. Available from: `https://aip.scitation.org/doi/abs/10.1063/1.555544`.

[19] Bain AD, Khasawneh M. From NQR to NMR: The complete range of quadrupole interactions. Concepts in Magnetic Resonance Part A. 2004;22A(2):69–78. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cmr.a.20013. Available from: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cmr.a.20013`.

[20] McKinnon G. The Physics of Ultrafast MRI. In: Debatin JF, McKinnon GC, editors. Ultrafast MRI: Techniques and Applications. Berlin, Heidelberg: Springer; 1998. p. 1–51. Available from: `https://doi.org/10.1007/978-3-642-80384-0_1`.

[21] Kose R, Setoi A, Kose K. A Fast GPU-optimized 3D MRI Simulator for Arbitrary k-space Sampling. Magnetic Resonance in Medical Sciences. 2019;18(3):208–218.