Software paper for submission to the Journal of Open Research Software

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

---

1 (1) Overview

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

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

4 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.

5 Abstract
The *spinsim python* package 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 for *Nvidia Cuda* [3] compatible systems using GPU parallelisation. Along with other optimisations, this allows for speed improvements of up to four orders of magnitude while keeping staying just as accurate. *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.

6 Keywords
Time dependent Schrödinger equation; Quantum; Physics; Spin one; Spin half; Integrator; Exponentiator; GPU; Parallel computing; Solver; python; numba;

7 Introduction

7.1 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, or even multiple days.
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.
Given the recent boom in machine learning, many research computers are now equipped with advanced graphics processing units (GPUs). Their many cores, which can range from hundreds to tens of thousands in number, allows them to run some highly parallel algorithms much faster than a central processing unit (CPU). Examples include calculating colours for many pixels on a screen in the titular graphics processing, or the weights in a large neural network. By finding ways to parallelise the problem of solving quantum spin systems, we can use the many cores of a GPU to our advantage in this context as well.

8 Implementation and architecture

8.1 Quantum mechanics background

spinsim solves the time-dependent Schrödinger equation

$$i\hbar\frac{\mathrm{d}\psi(t)}{\mathrm{d}t} = H(t)\psi(t), \tag{1}$$

where the quantum state $\psi(t) \in \mathbb{C}^N$ is assumed normalised, and the Hamiltonian $H(t) \in \mathbb{C}^{N\times N}$ is Hermitian. Here N is the number of levels in the quantum system. Often we are considering systems with spin J of half or one. Here spin-half is the $N = 2$ case, and spin-one is $N = 3$. We set $\hbar = 1$, so that the Hamiltonian has physical dimension of inverse frequency.
Rather than parametrizing the problem in terms of the matrix elements of $H(t)$, we consider time varying real coefficients in a linear combination of fixed operators,

$$H(t) = \sum_{j=1}^{N^2-1} \omega_j(t) O_j. \tag{2}$$

It is well known that a charged spin-half system with magnetic moment $\overrightarrow{\mu}$, and gyromagnetic ratio γ, in a magnetic field $\overrightarrow{B}(t)$, has Hamiltonian

$$\begin{aligned} H(t) &= -\overrightarrow{B}(t)\cdot\overrightarrow{\mu} && (3)\\ &= -\gamma\overrightarrow{B}(t)\cdot\overrightarrow{J} && (4)\\ &= -\gamma\left(B_x(t)J_x + B_y(t)J_y + B_z(t)J_z\right) && (5)\\ &= \left(-\gamma B_x(t)J_x\right) + \left(-\gamma B_y(t)J_y\right) + \left(-\gamma B_z(t)J_z\right) && (6)\\ &= \omega_x(t)J_x + \omega_y(t)J_y + \omega_z(t)J_z. && (7) \end{aligned}$$

Here J_x, J_y and J_z are spin operators, equal to the Pauli matrices scaled down by $\frac{1}{2}$. We exclude the identity so that the Hamiltonian is traceless, which we can do because it corresponds to choosing an energy zero point, which is physically meaningless.

In the spin-one case there is no standard basis of operators A_j. Choices include the Gell Mann matrices, and multiple dipole-quadrupole bases [5] [6]. In general, we can choose any basis from the 8-dimensional Lie algebra $\mathfrak{su}(3)$, which is the vector space of traceless Hermitian operators that can generate transformations (from the corresponding Lie group $SU(3)$) in the spin-one system. With this in mind, we choose to represent the Hamiltonian a linear combination of matrices from a 4-dimensional subspace of $\mathfrak{su}(3)$, consisting of the spin matrices J_x, J_y and J_z, and a single quadrupole operator $Q = \frac{1}{3}\text{diag}(1, -2, 1)$,

$$H(t) = \omega_x(t)J_x + \omega_y(t)J_y + \omega_z(t)J_z + \omega_q(t)Q. \tag{8}$$

Note that Q is proportional to Q_{zz} [5] and Q_0 [6] from alternative quadrupole bases. These are the only operators necessary to simulate many spin-one quantum systems, including quadratic Zeeman splitting described by the Breit-Rabi formula [7] important to experiments in our lab. The *spinsim* simulator can also be configured to solve a general spin-one system by setting the Hamiltonian to an arbitrary point in $\mathfrak{su}(3)$, using the full quadrupole basis, which extends the possible Hamiltonian to

$$\begin{aligned} H(t) = {} & \omega_x(t)J_x + \omega_y(t)J_y + \omega_z(t)J_z + \omega_q(t)Q \\ & + \omega_{u1}(t)U_1 + \omega_{u2}(t)U_2 + \omega_{v1}(t)V_1 + \omega_{v2}(t)V_2. \end{aligned} \tag{9}$$

Here the added operators are those defined in [6]. Note that this is included for completeness, but has not been thoroughly tested, as our lab has no physical context for modelling the U_1, U_2, V_1 and V_2 operators.

As well as integrating the Schrödinger equation, *spinsim* also has the functionality to calculate the expected spin projection $\left\langle \overrightarrow{J} \right\rangle(\text{t})$ of a system from its state. In an experimental setting, one cannot measure the value of the state itself, and must instead measure observables such as spin projection. If this is done for an ensemble of systems, the average spin projection over all systems will converge to the expected value as calculated here.

8.2 Unitary time evolution and the Magnus expansion

It is possible to write $\psi(t)$ in terms of a unitary transformation $\mathcal{U}(t, t_0)$ of the state $\psi(t_0)$, for any time t_0, that is, $\psi(t) = \mathcal{U}(t, t_0)\psi(t_0)$. It then follows from Equation (1) that $\mathcal{U}(t, t_0)$ also follows a Schrödinger equation,

$$i\hbar\frac{\mathrm{d}\mathcal{U}(t, t_0)}{\mathrm{d}t} = H(t)\mathcal{U}(t, t_0). \tag{10}$$

Thus, to solve Equation (1) for $\psi(t)$ given a particular $\psi(t_0)$, one only needs to solve Equation (10) for $\mathcal{U}(t, t_0)$. Since $\mathcal{U}(t, t_0)$ is unitary, it can be written as a matrix exponential of a skew-Hermitian (also termed anti-Hermitian) matrix. If the Hamiltonian is constant, then this exponential is

$$\mathcal{U}(t, t_0) = \exp(-iH \cdot (t - t_0)). \tag{11}$$

For a time varying Hamiltonian, the general solution for $\mathcal{U}(t, t_0)$ is much more complex, because it encapsulates the full solution to the time-dependent Schrödinger

equation. The Dyson series [8] gives an explicit expression for $\mathcal{U}(t, t_0)$ in terms of multiple time integrals over nested time commutators of $H(t')$. While the Dyson series has been used numerically [8], it has only been so recently, in particular because once truncated, it is in general no longer unitary.
The Magnus series

$$\mathcal{U}(t, t_0) = \exp\left(\Omega(t, t_0)\right) = \exp\left(\sum_{m=1}^{\infty} \Omega_m(t, t_0)\right) \tag{12}$$

explicitly preserves unitarity when truncated [9]. The first terms in the Magnus series are [10]

$$\Omega_1(t, t_0) = \int_0^t \mathrm{d}t_1 A(t_1) \tag{13}$$

$$\Omega_2(t, t_0) = \frac{1}{2}\int_{t_0}^{t} \mathrm{d}t_1 \int_{t_0}^{t_1} \mathrm{d}t_2 [A(t_1), A(t_2)] \tag{14}$$

$$\Omega_3(t, t_0) = \frac{1}{6}\int_{t_0}^{t} \mathrm{d}t_1 \int_{t_0}^{t_1} \mathrm{d}t_2 \int_{t_0}^{t_2} \mathrm{d}t_3 \left([A(t_1), [A(t_2), A(t_3)]] + [A(t_3), [A(t_2), A(t_1)]]\right), \tag{15}$$

where$[X, Y] = XY - YX$ is the commutator, and, in the case of quantum mechanics, $A(t') = -iH(t')$. Note that, when truncated to first order, the magnus expansion $\mathcal{U}(t, t_0) = \exp\left(\int_0^t \mathrm{d}t_1 \left(-iH(t_1)\right)\right)$ reduces to Equation (11), with H approximated to be constant with its average value.
A Magnus series over a time step T will be guaranteed to converge only if $\int_{t_0}^{t_0+T} \mathrm{d}t_1 \|H(t_1)\| < \pi$ [10]. Furthermore, each subsequent term in the expansion increases in complexity rapidly. For these reasons, the Magnus expansion is used as a time stepping method rather than a single step to solve the complete system. Time stepping can be made from the fact that time evolution operators can be split into a product via

$$\mathcal{U}(t, t_0) = \mathcal{U}(t, t_{n-1})\mathcal{U}(t_{n-1}, t_{n-2}) \cdots \mathcal{U}(t_2, t_1)\mathcal{U}(t_1, t_0). \tag{16}$$

Each of the time-evolution operators can be approximated by a Magnus expansion. Many unitary time stepping techniques have been developed based on the Magnus expansion [11]. Some of these techniques use Gauss-Legendre quadrature sampling and Baker–Campbell–Hausdorff formula to respectively avoid the integration and commutators of the Hamiltonian, producing simple expressions that can be used for unitary time stepping [12].
In particular, we use the commutator free, fourth order method (CF4) from Reference [11]. Suppose we wish to evaluate a time step of $\mathcal{U}(t + \mathrm{d}t, t)$ using the CF4 method. The Hamiltonians are sampled at times

$$t_1 = t + \frac{1}{2}\left(1 - \frac{1}{\sqrt{3}}\right)\mathrm{d}t \quad \text{and} \tag{17}$$

$$t_2 = t + \frac{1}{2}\left(1 + \frac{1}{\sqrt{3}}\right)\mathrm{d}t, \tag{18}$$

based on the second order Gauss-Legendre quadrature

$$\overline{H}_1 = \frac{3+2\sqrt{3}}{12}H(t_1) + \frac{3-2\sqrt{3}}{12}H(t_2) \quad \text{and} \tag{19}$$

$$\overline{H}_2 = \frac{3-2\sqrt{3}}{12}H(t_1) + \frac{3+2\sqrt{3}}{12}H(t_2). \tag{20}$$

Then

$$\mathcal{U}(t+\mathrm{d}t,t) \approx \exp(-i\overline{H}_2\mathrm{d}t)\exp(-i\overline{H}_1\mathrm{d}t) \tag{21}$$

is used to approximate the time-evolution operator.

8.3 Exponentiator

Evaluating matrix exponentials is a core part of the integration algorithm. Rather than exponentiating the Hamiltonian directly as in Equation (21), *spinsim* works with the field functions $\omega_i(t)$.
For spin-half, the exponentiator is in an analytic form in ω_x, ω_y and ω_z.For spin-one, an exponentiator based on the Lie-Trotter product formula [13]

$$\exp(X+Y) = \lim_{n\to\infty}\left(\exp\left(\frac{X}{n}\right)\exp\left(\frac{Y}{n}\right)\right)^n, \tag{22}$$

is used. An advantage of the Lie-Trotter approach is that $\exp(-iA_k/n)$ has known analytic forms for the Lie algebra basis elements A_k. Hence the unitary time-evolution operator is approximated by $U = \exp(-iHt) \approx T^n$ where

$$T = \exp\left(\frac{-i\omega_x J_x}{n}\right)\exp\left(\frac{-i\omega_y J_y}{n}\right)\exp\left(\frac{-i\omega_z J_z}{n}\right)\exp\left(\frac{-i\omega_q Q}{n}\right). \tag{23}$$

In fact, commutation relations between the Lie basis operators and the leapfrog splitting method [14] allow us to write

$$T = \exp\left(-i\frac{1}{2}D_{z,q}\right)\exp(-i\Phi J_\phi)\exp\left(-i\frac{1}{2}D_{z,q}\right). \tag{24}$$

Here $z = \omega_z/n$, $q = \omega_q/n$, $\Phi = \sqrt{\omega_x^2+\omega_y^2}/n$, $\phi = \mathrm{atan2}(\omega_y,\omega_x)$, $D_{z,q} = zJ_z + qQ$ and $J_\phi = \cos(\phi)J_x + \sin(\phi)J_y$. The element-wise analytic form of this is

$$T = \begin{pmatrix} \left(\cos\left(\frac{\Phi}{2}\right)e^{-iz/2}e^{-iq/6}\right)^2 & \frac{-i}{\sqrt{2}}\sin(\Phi)e^{iq/6}e^{-iz/2}e^{-i\phi} & -\left(\sin\left(\frac{\Phi}{2}\right)e^{iq/6}e^{-i\phi}\right)^2 \\ \frac{-i}{\sqrt{2}}\sin(\Phi)e^{iq/6}e^{-iz/2}e^{i\phi} & \cos(\Phi)e^{i4q} & \frac{-i}{\sqrt{2}}\sin(\Phi)e^{iq/6}e^{iz/2}e^{-i\phi} \\ -\left(\sin\left(\frac{\Phi}{2}\right)e^{-iq/6}e^{i\phi}\right)^2 & \frac{-i}{\sqrt{2}}\sin(\Phi)e^{iq/6}e^{iz/2}e^{i\phi} & \left(\cos\left(\frac{\Phi}{2}\right)e^{iz/2}e^{-iq/6}\right)^2 \end{pmatrix}. \tag{25}$$

Matrix exponentiation is completed by raising T to an integer n. To approximate the limit in Equation (22), n must be a large number. We choose n to be of the form $n = 2^\tau$, as powers of two are particularly effient large numbers to raise matrices to.This can be done just by iterative squaring, which requires a small number of

operations. While there are some loose lower bounds for the number of squares τ [15], using these to choose a value can cause floating point errors from over-squaring. Instead, we tested the formula on 10^5 matrices for different values of τ, and compared them to results given by `scipy.linalg.expm()`, from the popular python library *SciPy*. We found that the error minimised after a number of squares of $\tau = 24$. Thus, by default, $n = 2^{24}$.
In practice the calculations here are done by finding the differences of the matrices from the identity to avoid floating point cancellation errors from subtracting a small number from 1.
This spin-one exponentiator evolves Hamiltonians spanned by J_x, J_y, J_z and Q which is sufficient for three level systems in arbitrary bias fields, but with single-photon coupling. An exponentiator capable of evolving an arbitrary spin-one Hamiltonian, for example with different coupling between the lower and upper pairs of states, or with two-photon coupling, is included in the package. When assessing accuracy for spin-one problems, we have used the single-photon exponentiator.
Note that, the methods for both spin-half and spin-one use analytic forms of matrix exponentials so that T is unitary by construction; $\mathcal{U} = T^n$ is then also unitary. Simulations in *spinsim* thus maintain unitarity and so conserve probability even over very large numbers of time steps.

8.4 Discretisation and parallelisation

Frequently we wish to sample the state at times t_k spaced more coarsely than the integration time step. Consider discrete times $t_k = t_0 + \mathrm{D}t \cdot k$ where $\mathrm{D}t$ is the time step of the time-series, in contrast to $\mathrm{d}t$, the time step of the integration. Denoting $\psi_k = \psi(t_k)$ and $\mathcal{U}_k = \mathcal{U}(t_k, t_{k-1})$, the time-series of states ψ_k and time-evolution operators $\mathcal{U}_k$ satisfy

$$\psi_k = \mathcal{U}_k \psi_{k-1}. \tag{26}$$

This presents an opportunity for parallelism. While ψ_k depends on ψ_{k-1}, the operators $\mathcal{U}_k$ depend only on the field functions, and not on previous states $\psi_{k'}$ or previous operators $\mathcal{U}_{k'}$. Hence, the time-evolution operators $\mathcal{U}_k$ can be calculated in parallel.
spinsim splits the full simulation into small time intervals $[t_{k-1}, t_k]$, and calculates time-evolution operators $\mathcal{U}_k$ for these intervals in massive parallel on a GPU. When all the $\mathcal{U}_k$ are calculated, the CPU then multiplies them together (a comparatively less demanding job than calculating them) using Equation (26) to determine the output samples ψ_k of the evolving state.
Beyond discretisation for parallelisation, we discretise our time-evolution operator into individual integration time steps. Each of the $\mathcal{U}_k$ is further split into products $u^k_{L-1} \cdots u^k_0$ of L time-evolution operators, now separated by the integration time step $\mathrm{d}t = \mathrm{D}t/L$.

8.5 Dynamically rotating frame

Hamiltonians which have dominant and slowly-varying terms induce rotations around a primary axis, which is usually chosen to be the quantisation axis z. By trans-

forming into a frame rotating around this axis, a new Hamiltonian H^r with reduced norm is formed, making the time evolution operators more accurate. In numerical analysis, this is known as preconditioning [*Is this right James?*]; in quantum mechanics, as transforming to the interaction picture. If the rotating frame option is selected, the time-evolution operators $\mathcal{U}_k$ are first calculated within a rotating frame of reference as $\mathcal{U}_k^r$, which, in some situations, reduces the size of the field functions used in the calculation, increasing accuracy. The rotation speed of the rotating frame is calculated locally for each parallel time step $\mathcal{U}_k$, and only for rotations around the z axis. The rotating from field functions $\omega_x^r, \omega_y^r, \omega_z^r$, and ω_q^r are related to the field function ω from the user input via

$$\omega_x^r(t) + i\omega_y^r(t) = e^{-i\omega_r t}(\omega_x(t) + i\omega_y(t)), \tag{27}$$

$$\omega_z^r(t) = \omega_z(t) - \omega_r \quad \text{and} \tag{28}$$

$$\omega_q^r(t) = \omega_q(t), \quad \text{for spin-one.} \tag{29}$$

Where $\omega_r = \omega_z(t_k + \frac{1}{2}\mathrm{D}t)$ is sampled the midpoint value of the fields 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 $\omega_z(t)$, while leaving the other field components at an equivalent magnitude. The rotation is then applied to obtain the lab frame time-evolution operator $\mathcal{U}_k$ via

$$\mathcal{U}_k = \exp(-i\omega_r J_z \mathrm{D}t)\mathcal{U}_k^r. \tag{30}$$

Specifically, this relationship is $\mathcal{U}_k = \mathrm{diag}\left(\exp\left(-i\frac{1}{2}\omega_r \mathrm{D}t\right), \exp\left(i\frac{1}{2}\omega_r \mathrm{D}t\right)\right)$ for spin-half, and $\mathcal{U}_k = \mathrm{diag}(\exp(-i\omega_r \mathrm{D}t), 0, \exp(i\omega_r \mathrm{D}t))$ for spin-one.
It is a common technique in solving quantum mechanical problems to enter rotating frames, and their more abstract counterparts of interaction pictures [16]. This is typically done in to enable the Rotating Wave Approximation (RWA), which is an assumption that the oscillatory components of $\omega_x^r, \omega_y^r, \omega_z^r$, and ω_q^r on an average make minor contributions to time evolution, 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.

8.6 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 $\mathcal{U}_k$. This returns when each of the $\mathcal{U}_k$ have been evaluated.
The thread starts by calculating t_k and, if the rotating frame is being used, ω_r. The latter is done by sampling a (`numba.cuda.jit()`ed version of a) user provided *python* function ω describing how to sample the Hamiltonian. The code then loops over each integration time step dt 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 $\omega(t)$, as well as calculate $e^{-i\omega_r t}$, 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 $\overline{H}_1$ and $\overline{H}_2$ values, and finally taking the matrix exponentiation in a fourth device function. u_l^k is premultiplied to $\mathcal{U}_k^r$ (which is initialised to 1), and the loop continues.
When the loop has finished, if the rotating frame is being used, $\mathcal{U}_k^r$ is transformed to $\mathcal{U}_k$ as in Equation (26), and this is returned. Once all threads have executed, the state ψ_k is calculated in a (CPU) `numba.jit()`ed function from the $\mathcal{U}_k$ and an initial condition ψ_{init}.

8.7 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 field function ω is provided by the user as a plain python function (that must be `numba.cuda.jit()` compatible). This allows users to define ω 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 different GPUs. Third, 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. Note that the expected spin projection is calculated as a lazy parameter if needed, rather than returned by the simulator object.
spinsim is designed so that a single simulator compilation can be used to execute many simulations, sweeping through a particular parameter, while not needing to be recompiled. This is done through the `sweep_parameter` argument to the user provided field function ω. The user uses `sweep_parameter` to determine a variable parameter for the Hamiltonian. After compiling the `spinsim.Simulator` object, the user can set the value for `sweep_parameter` for a particular simulation as the first argument when calling the function `spinsim.Simulator.evaluate()`. For a use case example, one might want to simulate many spin systems at different locations in a magnetic field gradient in an MRI experiment. To do this they could choose to set $\omega_z =$ `sweep_parameter`, which would vary the magnetic field bias in the ω direction for each experiment. This feature is demonstrated with examples in the documentation.

9 Quality control

9.1 Evaluation of accuracy

All accuracy benchmarks were run in *Cuda* mode, on the desktop computer with the *Ryzen 7 5800X* and *GeForce RTX 3080*, which from the tests in Figure 4 are the fastest CPU and GPU from the devices tested. Note that these benchmarks do not take into account the amount of time required to JIT compile simulation code for the first time (order of seconds), as we want to look at this in the limit of running many simulations while sweeping through a parameter that differs in each one.

Benchmarks were performed using `neural-sense.sim.benchmark` (where `neural-sense` [17] is the quantum sensing package that *spinsim* was written for). This simulation involves continuously driving transitions in the system for 100ms, while exposing it to a short 1ms pulsed signal that the system should be able to sense. We first wanted to test the accuracy of the different integration techniques for various integration time steps. Here we wanted to test the advantages, if any of using a Magnus based integration method. Accuracy was calculated by taking the quantum state simulation evaluations of a typical quantum sensing experiment and finding the Root Mean Squared (RMS) error from a baseline simulation run by `scipy.integrate.ivp_solve()` as part of the *SciPy* python package, via

$$\epsilon = \frac{1}{K}\sqrt{\sum_{k=0}^{K-1}\sum_{m_j=-j}^{j}|\psi_{k,(m_j)} - \psi_{k,(m_j)}^{\text{baseline}}|^2}, \tag{31}$$

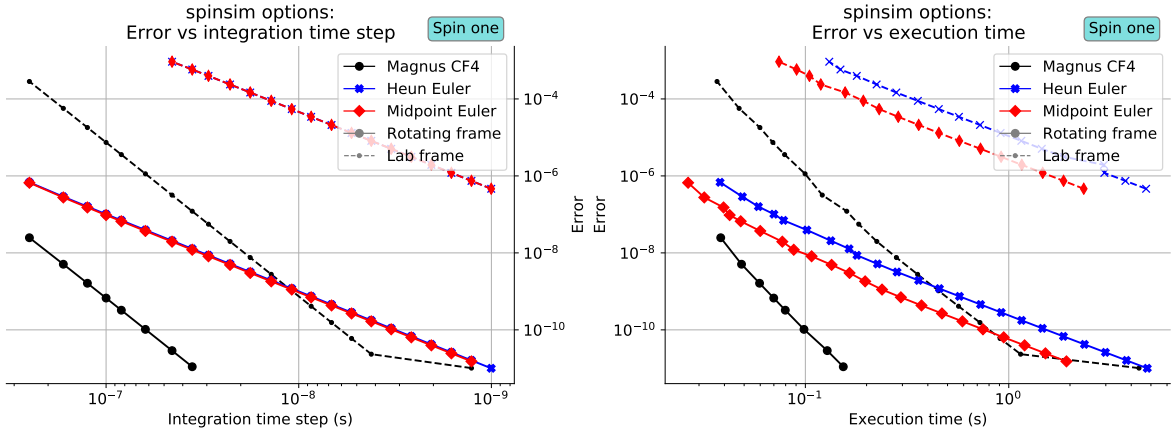
where $j \in \{\frac{1}{2}, 1\}$ is the spin quantum number of the system.

This baseline was computed in 2.4 hours (in comparison to order of 10ths of a second these *spinsim* tests were executed in), and was also used for comparisons to other software packages. These are shown in Figures 1a, and 2a. 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 1b, and 2b. 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. Additionally, errors bellow 10^{-11} were excluded, as this was the order of magnitude of the errors in the reference simulation.

The integration techniques tested were the Magnus based CF4, as well as two Euler based sampling methods. A midpoint Euler method was chosen as the simplest (and fastest for a given integration time step) possible sampling method, whereas a Heun Euler sampling method was used as a comparison to previous versions of this code. We benchmarked these methods both while using and not using the rotating frame option. This was done separately for spin-one and spin-half systems, to ensure they both yield accurate results.

From Figures 1 and 2, we find that overall, the results that *spinsim* gives are accurate to those of *SciPy*. Figures 1a, and 2a 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 1b, and 2b, 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.



(a) (b)

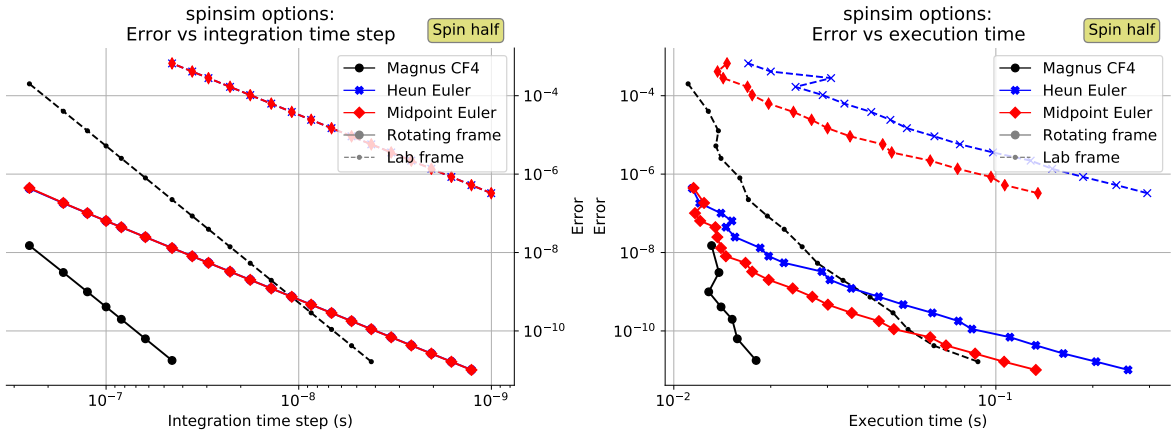
Figure 1: Speed and accuracy of the spin-one options of *spinsim*. A simulation of a typical neural sensing experiment was run for every integration time step, for each of the possible integration techniques (Magnus based commutator free 4, and two Euler methods). In the simulation, transitions are continuously driven in the spin system for a duration of 100ms, and an additional small signal is injected for 1ms. Each technique was tested while both using and not using a transformation into a rotating frame. Both execution time and error were recorded for each of the simulations. Error is RMS error compared to a long running *SciPy* baseline.

9.2 Comparison to alternatives

We ran the same error (using Equation (31)) and execution time benchmarks on some alternative packages to compare *spinsim*'s performance to theirs. The packages compared were are listed in Table 1.

In each case, the step sizes of the alternative integrators were limited to a maximum value to obtain simulation results of different accuracies, and the maximum number of steps were modified in some cases to allow more steps to take place. 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. Again, we used the longest running *SciPy* simulation as a ground truth for comparison, as the accuracy of *Mathematica* plateaus at small time steps. Functions used for sampling were compiled in both *spinsim* and *QuTip*, and the time taken



(a) (b)

Figure 2: Speed and accuracy of the spin-half options of *spinsim*. A simulation of a typical neural sensing experiment was run for every integration time step, for each of the possible integration techniques (Magnus based commutator free 4, and two Euler methods). In the simulation, transitions are continuously driven in the spin system for a duration of 100ms, and an additional small signal is injected for 1ms. Each technique was tested while both using and not using a transformation into a rotating frame. Both execution time and error were recorded for each of the simulations. Error is RMS error compared to a long running *SciPy* baseline.

to complete a simulation was measured in both cases for a second simulation using the already compiled functions.

Each benchmark was run one simulation at a time. However, it might be possible to increase the average speed of many benchmarks from *Mathematica* and *SciPy* 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 run them all at once. Multithreading was also not attempted using *SciPy*, due to the fact that running the full set of benchmarks of only a single simulation per integration time step consumes a day of computational time. But to be fair, both *Mathematica* and *SciPy* results are plotted with an artificial reduction in execution time by a factor of 8, which is an upper bound for the speed increase that could be obtained by running them parallel on an 8 core processor, which appears as dotted lines. Results from *spinsim* and *QuTip* automatically run multithreaded, so this handicap is not plotted for these packages.

From Figure 3, for any given error tolerance, *spinsim* is over 3 orders of magnitude faster than *Mathematica* and *QuTip*, and 4 orders of magnitude more accurate than *SciPy*. In practice, this means that a 25 minute *SciPy* simulation is reduced to 50ms, and a three week long *SciPy* batch simulation of 1000 separate systems (a realistic requirement for testing quantum sensing protocols) would take less than one minute in *spinsim*.

Table 1: The software packages used for and verification of *spinsim*.

Software package	Function	Details
spinsim	`Simulator()`	The best performing *spinsim* configuration, using the CF4 integrator and the rotating frame mode. This was run both on CPU and GPU.
QuTip [18]	`sesolve()`	The Schrödinger equation solver from the popular *python* quantum mechanics library, *QuTip*. This was chosen as a comparison to a specially designed solver used within the physics community for this application. Like *spinsim*, *QuTip* allows users to sample from compiled functions, and uses parallelisation.
Mathematica [19]	`NDSolve()`	A generic ODE solver from the *Mathematica* software. This was chosen as it is popular with our lab group for simulating magnetometry experiments.
SciPy [20]	`integrate.ivp_solve()`	A generic ODE solver from the popular *python* scientific computing library. This was chosen as a comparison to a generic solver from within the python ecosystem.

9.3 Parallelisation performance

Once the algorithm behind *spinsim* was developed, we wanted to check its execution speed while running on various devices. The main reason for this test was to quantify the speed increase of parallelisation by comparing execution speeds on highly parallel devices (being GPUs), compared highly procedural devices (being CPUs). Speed 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 4. The integration code was compiled by `numba` for multicore CPUs, CPUs running single threaded, and *Nvidia Cuda* compatible GPUs, and run on different models of each of them. These test devices are given in Table 2.

The results in Figure 4 show the benefit to using parallelisation when solving a spin system problem. Moving from the 6 core *Core i7-8750H* CPU to the 12 core *Ryzen 9 5900X* CPU doubles the execution speed, as does moving from the 384 core *Quadro K620* GPU to the 768 core *Quadro T1000* GPU. So, in these cases, performance scales in proportion to thread count. Moving from a single core processor to a high end GPU increases performance by well over an order of magnitude on three of the four computers tested on. Even the low end *Quadro K620* was an improvement over

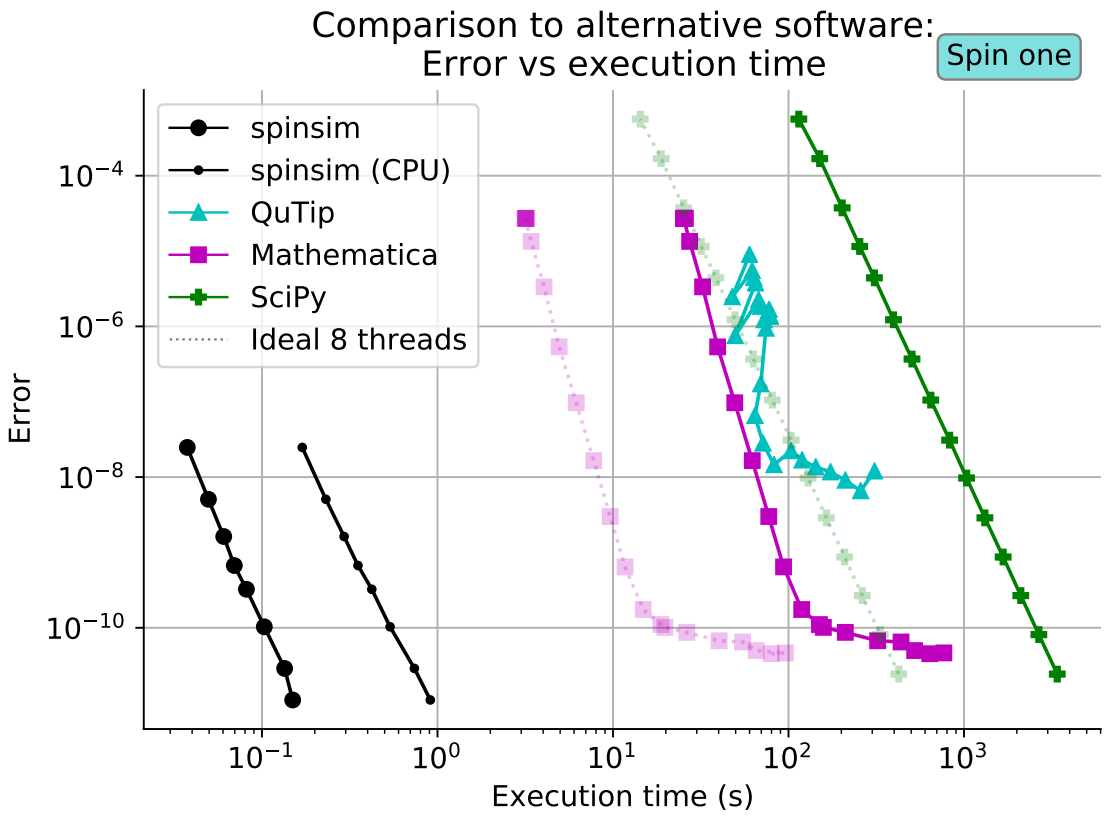


Figure 3: Speed vs accuracy of two alternative integration packages. A simulation of a typical neural sensing experiment was run for every integration time step, for each of alternative packages (`qutip.sesolve()` from *QuTip*, `NDSolve()` from *Mathematica*, and `scipy.integrate.ivp_solve()` from *SciPy*). In the simulation, transitions are continuously driven in the spin system for a duration of 100ms, and an additional small signal is injected for 1ms. Both execution time and error were recorded for each of the simulations. Error is RMS error compared to a long running *SciPy* baseline. The *Mathematica* and *SciPy* results are also shown with a speed up by a factor of 8 to represent the upper bound of hypothetical parallelisation across an 8 core CPU.

the *Core i7-6700* used by the same computer. Execution speed vs number of cuda cores starts to plateau as the number of cores increases. This happens because the time it takes to transfer memory from RAM to VRAM (dedicated graphics memory), which is independent on the number of cores of the GPU, becomes a comparable to the execution time of the simulator logic. However, there is still a large improvement from using the high end *GeForce RTX 3070* to the *GeForce RTX 3080*, with the latter simulating the experiment almost twice as fast as it would take to run the simulated experiment in the real world.
Surprisingly, we found that the *Ryzen 7 5800X* 8 core CPU was able to execute the benchmark faster than the *Ryzen 9 5900X* 12 core CPU. This can be explained by the fact that the *Ryzen 7 5800X* was liquid cooled, rather than being air cooled, meaning it was likely able to boost to a higher core clock, and resist thermal throt-

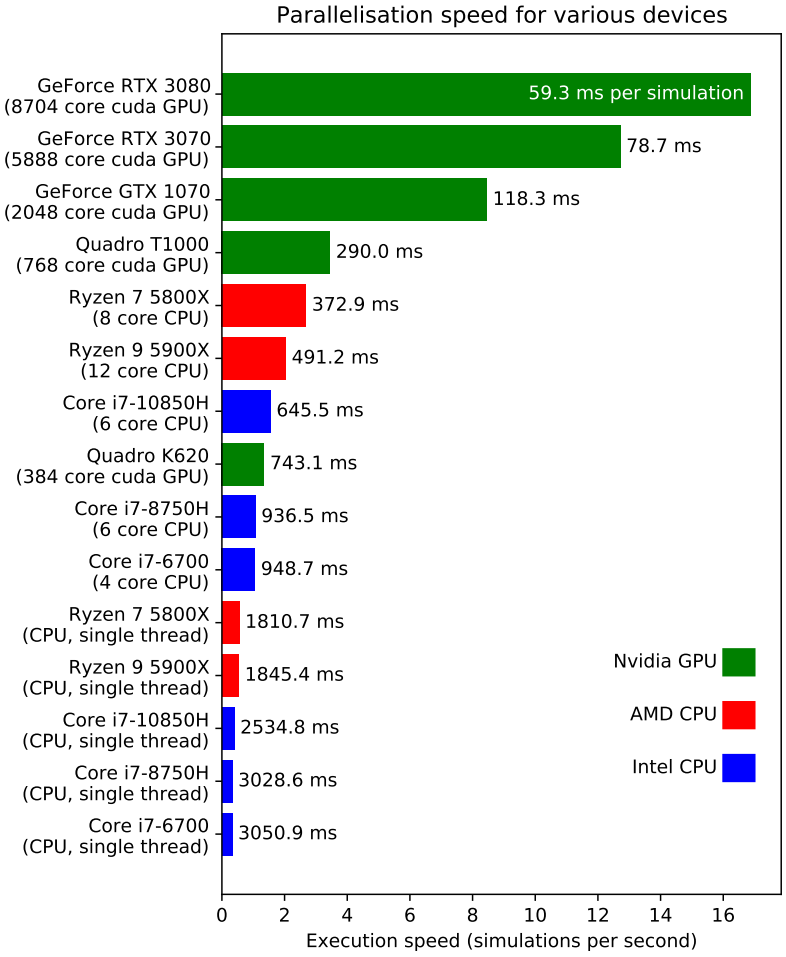


Figure 4: Evaluation speed of a simulation of a typical spin-one sensing experiment on both CPUs and GPUs. Integration time step is set to 100ns. Transitions are continuously driven in the spin system for a duration of 100ms, and an additional small signal is injected for 1ms. Evaluation time is determined by an average of 100 similar simulations for each device, where each individual simulation varies in dressing amplitude (transition frequency).

Table 2: Devices used in the parallelisation speed test. These devices are part of individual computers, which are separated here by horizontal lines.

Device	**Type**	**RAM (GiB)**	**Cores**	**Cooling**
Core i7-6700	Intel CPU	16	4	Air
Quadro K620	Nvidia GPU	2	384	Air
Core i7-8750H	Intel CPU	16	6	Air
GeForce GTX 1070	Nvidia GPU	8	2048	Air
Core i7-10850H	Intel CPU	32	6	Air
Quadro T1000	Nvidia GPU	4	768	Air
Ryzen 9 5900X	AMD CPU	32	12	Air
GeForce RTX 3070	Nvidia GPU	8	5888	Air
Ryzen 7 5800X	AMD CPU	32	8	Liquid
GeForce RTX 3080	Nvidia GPU	10	8704	Air

tling.
Figure 4 can be used by potential *spinsim* users for finding the relative performance for devices of varying abilities of parallelisation. We would recommend Another factor not shown in the plot is the fact that, in practice, running highly code parallel code on a CPU on a personal computer will severely limit the responsiveness of other applications, as it can utilise the entire CPU (as it should). In contrast, this does not happen when running a GPU based program, as it requires very little CPU utilisation to function. This can be convenient when running simulations on a personal laptop or desktop, as other work on the computer does not have to halt while simulations are being run.

9.4 Testing

During the accuracy tests, it was confirmed that all possible modes of *spinsim* agree with a standard *SciPy* simulation up to an arbitrarily 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 the *GeForce GTX 1070*, 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`.

10 (2) Availability

11 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.

12 Programming language

Python (3.7 or greater)

13 Additional system requirements

To use the (default) *Nvidia Cuda* GPU parallelisation, one needs to have a *Cuda* compatible *Nvidia* GPU [21]. For *Cuda* mode to function, one also needs to install the *Nvidia Cuda* toolkit [22]. If *Cuda* is not available on the system, the simulator will automatically parallelise over multicore CPUs instead.

14 Dependencies

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

15 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.

16 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

17 Language

English.

18 (3) Reuse potential

18.1 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 [23], 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 [24] 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) [25], 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 [26], 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 [27], 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 [28], 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.

18.2 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.

19 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.

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

21 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] Mockler RC. Atomic Beam Frequency Standards. In: Marton L, editor. Advances in Electronics and Electron Physics. vol. 15. Academic Press; 1961. p. 1–71. Available from: https://www.sciencedirect.com/science/article/pii/S0065253908609312.

[8] Kalev A, Hen I. An integral-free representation of the Dyson series using divided differences. arXiv:201009888 [cond-mat, physics:hep-th, physics:math-ph, physics:nucl-th, physics:quant-ph]. 2020 Oct. ArXiv: 2010.09888. Available from: http://arxiv.org/abs/2010.09888.

[9] Magnus W. On the exponential solution of differential equations for a linear operator. Communications on Pure and Applied Mathematics. 1954;7(4):649–673. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160070404. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.3160070404.

[10] Blanes S, Casas F, Oteo JA, Ros J. The Magnus expansion and some of its applications. Physics Reports. 2009 Jan;470(5):151–238. Available from: http://www.sciencedirect.com/science/article/pii/S0370157308004092.

[11] 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.

[12] Blanes S, Moan PC. Fourth- and sixth-order commutator-free Magnus integrators for linear and non-linear dynamical systems. Applied Numerical Mathematics. 2006 Dec;56(12):1519–1537. Available from: https://www.sciencedirect.com/science/article/pii/S0168927405002163.

[13] 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.

[14] Barthel T, Zhang Y. Optimized Lie–Trotter–Suzuki decompositions for two and three non-commuting terms. Annals of Physics. 2020 Jul;418:168165. Available from: https://www.sciencedirect.com/science/article/pii/S0003491620300981.

[15] Suzuki M. Generalized Trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems. Communications in Mathematical Physics. 1976 Jun;51(2):183–190. Available from: http://link.springer.com/10.1007/BF01609348.

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

of its initial publication in this journal.

By submitting this paper you agree to the terms of this Copyright Notice, which will apply to this submission if and when it is published by this journal.