Classical motion in synthetic monopole fields

UPPSALA
UNIVERSITET

Bachelor's thesis 15c
Department of Physics and Astronomy
May 8, 2022

Author: Ola Carlsson
Supervisor: Erik Sjöqvist
Subject reviewer: Patrik Thunström

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 Introduction

1.1 On the occurrence of monopoles

Our theory of electromagnetism carries certain asymmetries between the magnetic and electric fields. Such a distinction well known and often discussed is the absence of magnetic monopoles, i.e. that any analogue of electric charge is absent for the magnetic field. It is an extension often suggested by theorists, initially by P.M. Dirac in 1931[1], to include magnetic charges and also potentially magnetic currents in Maxwell's equations, but there is as of yet no accepted empirical data to support this cause. Magnetic monopoles do however appear occasionally in a less fundamental sense, as emergent phenomena in many-body systems for example in spin ice [2] or Bose-Einstein condensates [3].

Another area in which magnetic monopoles has appeared is the study of so called *geometric* phases of quantum systems, first described by M. Berry in 1984 [4]. Roughly speaking this is the phase acquired by the wave function of a system which is not dynamic in origin. The dynamic phase of a system state is induced by the energy of that state, while the geometric phase is surprisingly enough *in*dependent of the energy values, and rather arise as a function of the path taken by the system through the relevant "parameter-space" parametrising the Hamiltonian. For certain systems this geometric phase evolution resembles the evolution of a charged system in a magnetic vector potential field that lives in parameter space. This field, here named the *synthetic* magnetic field, happens to exhibit non-zero divergence at certain points, the degeneracy points of the state energies, which thus correspond to monopoles of the synthetic field.

Geometrical phase is an interesting field of study in of itself, but it is here with the appearance of magnetic-type monopoles that the present body of work finds its premise. While the study of fundamental monopoles remains impossible, we can through construction of a suitable system study the effects of magnetic monopoles through their action on the system state in parameter space. Parameter space can be put into correspondence with real space, and the movement of charged matter through monopole fields becomes measurable, even though fundamental monopoles remain fictitious.

1.2 A select system

Berry's original 1984 article considers a simple spin-system with a single magnetic dipole moment in an external magnetic field $\vec{\mathbf{B}}$. The relevant parameter space is the space of all possible external fields, and the geometric phase contribution takes the form of a synthetic magnetic field purely generated by a monopole sitting at the origin, i.e. at $\vec{\mathbf{B}} = \vec{\mathbf{0}}$. This simple field is a most trivial example of a monopolar field, so to find more complex behaviour this starting point of a system can be extended to include multiple spin components, i.e. multiple magnetic dipoles, and interactions between those dipoles. The effects of introducing such interactions is roughly that of splitting the origin-centred monopole into smaller constituent parts whose positions in parameter space depend on the exact nature of the spin-spin interactions[5].

This splitting is desired, and so the system studied will be composed of two massive spin-$\frac{1}{2}$ components that interact through the so-called Ising interaction described in section 3.2. This is to some extent the simplest spin-spin interaction and is dependent only on spin along a chosen axis, here chosen to be the axis connecting the two masses. Movement through parameter space can easily be mapped to movement of the center of mass through real space given an external inhomogeneous field, and the movement of the spin components relative to one another is as a simplest case the rotation through polar and azimuthal angles with fixed inter-component distance. Such a setup is reminiscent of a dumbbell translating and rotating through space, with the added complication that each "weight" of the dumbbell acts as a dipole (has spin) interacting with both the other weight and an external magnetic field.

It is the time evolution of this system, henceforth referred to as the dumbbell, with which this project is concerned. Approximate equations of motion will be derived and then be put to test in a numerical simulation. Underlying the process is the hope of discerning effects of the synthetic magnetic monopoles on the movement of the dumbbell.

1.3 Interpretations of the system

In addition to carrying dynamics of interest, the described dumbbell is also appropriate for its potential to model realisable physical systems. Such a realisation of the model herein described opens up the possibility of experimentally measuring the action of the synthetic fields, and by extension the action of synthetic monopoles.

Two paths of realisation spring to mind: Firstly a diatomic molecule with appropriate effective spin of the two constituent atoms could be tested. It would be of importance that the magnetic moment of each atom be large, so that the molecule couples to the external field strongly enough, and it would further be desired that spin-spin interactions between the molecules be strong so as to achieve the more exotic synthetic field texture mentioned in section 1.2. Preferably the gas phase of such a chemical should be easily obtainable, for it would be desired to measure the dynamics of single such molecules without inter-molecular interaction. For the dumbbell model to apply reasonably well the interatomic binding would also have to be of such a nature that the interatomic distance would not vary greatly. The plausibility of this approach, and the selection of suitable candidate molecules, is an interesting question in its own right and warrants further research. Something as simple as hydrogen gas is not necessarily unsuited.

Secondly, one might imagine a substantially smaller system consisting of a single atom with non-negligible nuclear and electronic spin. The same considerations concerning the strength of interactions apply as above, but this method appears to, at least for the layman, carry larger obstructions. Nuclear magnetic moments, while measurable due to resonance effects such as in nuclear magnetic resonance (NMR) measurements, are orders of magnitude smaller than their electronic counterparts[6]. This, together with the disparate masses of electrons and nucleus would necessitate a heavily asymmetric dumbbell model. It is also a well known fact that such classical approximations as the definite position of the electronic part of the system implied by a dumbbell model break down at these length scales. We must consider an atom a quantum thing, for if we do not we will find incorrect results.

2 Background

2.1 The adiabatic approximation

Imagine sitting in a train with a spinning top in somewhat stable movement on the table in front of you. Consider then what would happen were the train to (de)accelerate. If the conductor breaks forcefully everything in the cabin experiences a fictitious force in the opposite direction of the train acceleration, and the spinning top would fall quickly onto the table. If instead the train slows down gradually the fictitious force will be much smaller. It can even be small enough in comparison with the properties of the top, its angular momentum and distribution of inertia, that the top manages to keep spinning on the table. Resilience to perturbation is the very phenomena that keeps a spinning top stable in the first place, even if the table were placed in a static house and not a moving train. The top keeps its state, the stable position and direction of rotation on the table, even though its environment, the train, undergoes change.

This thought experiment if translated to the quantum realm, where states are kets or wavefunctions and the environment is encapsulated in the system Hamiltonian, captures the essence of the *adiabatic theorem* finalized by Born and Fock[7]. Concisely stated, the original version reads in translation to English:

> A physical system remains in its instantaneous eigenstate if a given perturbation is acting on it slowly enough and if there is a gap between the eigenvalue and the rest of the Hamiltonians spectrum

In addition to the classical analogue given above, where the speed of the change of environment was required to be "slow", we note an additional requirement on the state: the energy of the state must not lie close to the energy of any other state available. This facet will be of great importance in the discussion of geometric phase, and is in a sense the origin of the synthetic magnetic monopoles. The assumption that the conditions for this theorem are all satisfied is often referred to as the *adiabatic approximation*, which will be done also in this text.

As an illustrative example consider a particle of spin s in some external magnetic field $\vec{\mathbf{B}}$. The total energy, and thus Hamiltonian, for such a system is

$$\mathcal{H} = -\gamma \vec{\mathbf{B}} \cdot \vec{\mathbf{S}}.$$

Here $\vec{\mathbf{S}}$ is the spin operator and γ is a constant determining the magnetic dipole moment per unit of angular momenta, i.e. it looks typically as $\gamma = \frac{g\mu}{\hbar}$. Here g is a dimensionless so-called *g-factor* and μ is an appropriate magneton. The eigenstates to this Hamiltonian are the same as the spin eigenstates

$|\vec{\mathbf{n}}, m\rangle$ in the direction $\vec{\mathbf{n}}$ of $\vec{\mathbf{B}}$ with the quantum number m describing their directional spin eigenvalues. The eigenvalues E_m to the Hamiltonian, the energies of the states, then become

$$E_m = -\gamma B\hbar m.$$

Here B is the magnetic field strength. The adiabatic theorem then states for this example that any "slow" change of the Hamiltonian, that is any slow change of the magnetic field $\vec{\mathbf{B}}$, will not change the state of the system if it were originally put into an eigenstate $|\vec{\mathbf{n}}, m\rangle$. The only things that change for the state are possibly the direction $\vec{\mathbf{n}}$ and the energy level E_m. The criteria for the theorem to hold will here translate into, apart from the speed of change, that B and m both be nonzero. If either of those values were zero, typically the magnetic field strength, the energy levels would be degenerate and the theorem would not be applicable. Values of m can only be zero of the particle is bosonic, i.e. has integer spin quantum number, and if the system was put into such a degenerate state from the onset.

2.2 The geometric phase

It was the evolution of systems satisfying the adiabatic theorem that concerned Berry as he demonstrated the nature of the geometrical phase in the eighties [4], although non-adiabatic extensions have been made since[8]. As mentioned in section 1.1 the adiabatic geometric phase is a contribution to the phase of a system undergoing adiabatic change which is *independent* of the energies during the adiabatic change[9]. Instead it is a contribution dependent only on the path traversed through parameter space, which in the above example is the three-dimensional space of possible external magnetic fields. The speed by which the path, often considered to be a loop $\mathcal{C}$ returning to its original position, is traversed matters not for the geometrical phase. Note however that this speed cannot become arbitrarily large, as it would then eventually break the adiabatic approximation. It is also relevant to the discussion to remind oneself that the accumulation of phase in quantum mechanics compromise all of dynamics, i.e. all forms of time evolution can be phrased as changes in the phase of states.

In less abstract terms, it can be shown that an arbitrary quantum energy eigenstate $|n\rangle$ (labelled after its energy E_n) affected by the Hamiltonian $\mathcal{H}$ will under adiabatic evolution along some loop $\mathcal{C}$ accumulate a geometric phase η_n. This means that if the loop is traversed in time T the state will after one revolution end up as $e^{-\frac{i}{\hbar}\int_0^T \langle n|\mathcal{H}|n\rangle dt} e^{\eta_n} |n\rangle$. Note the inclusion of the dynamic phase with the first factor, owing to the "standard" time evolution due to energy. The value of η_n can be calculated from the following integral[4]:

$$\eta_n(\mathcal{C}) = i \oint_{\mathcal{C}} \left\langle n \middle| \vec{\boldsymbol{\nabla}}_{\vec{\mathbf{R}}} n \right\rangle \cdot d\vec{\mathbf{R}}. \tag{1}$$

Here $\vec{\mathbf{R}}$ denotes the parameters at some point in parameter space, so the line integral is appropriately carried out over that quantity. Note also that the Hamiltonian and therefore also all states, bras and kets, depend on $\vec{\mathbf{R}}$. By Stoke's theorem this integral can be transformed into a surface integral over the enclosed area in parameter space. If parameter space is three dimensional as in the example in section 2.1 above the cross product can additionally be utilized, and the geometric phase can be recast as:

$$\eta_n(\mathcal{C}) = -\iint_{\mathcal{C}} d\vec{\mathbf{S}} \cdot \vec{\mathbf{G}}_n(\vec{\mathbf{R}}), \tag{2}$$

where

$$\vec{\mathbf{G}}_n(\vec{\mathbf{R}}) = \mathrm{Im} \sum_{l \neq n} \frac{\langle n| \vec{\boldsymbol{\nabla}}_{\vec{\mathbf{R}}} \mathcal{H} |l\rangle \times \langle l| \vec{\boldsymbol{\nabla}}_{\vec{\mathbf{R}}} \mathcal{H} |n\rangle}{(E_l - E_n)^2}. \tag{3}$$

It is here that the monopole field makes its entrance. While the integrand of equation 1 is only defined up to a constant, owing to an arbitrary phase factor of the eigenstate basis, the field $\vec{\mathbf{G}}_n$ is but the curl of this integrand and thus independent of the arbitrary constant. The phase evolution due to the geometric phase has then taken the form of the time evolution through a magnetic field $\vec{\mathbf{G}}_n$ derived from the vector potential

$$\vec{\mathbf{D}}_n = i \langle n| \vec{\boldsymbol{\nabla}}_{\vec{\mathbf{R}}} |n\rangle . \tag{4}$$

We call this field the synthetic magnetic field, or the synthetic gauge field, and consider it the field the action of which results in the geometric phase. The monopole structure so often mentioned previously

is also present in precisely this field, by which it is meant that the singularities of this field which occur at points of energy degeneration may infer a non-zero divergence. Since the field is not defined at these points, divergence should here be interpreted in the rather loose sense of a closed flux integral about some point divided by the enclosed volume. A nonzero divergence of a magnetic field is not allowed for standard Maxwellian magnetic fields and would imply some form of magnetic charge, monopoles. Even though our synthetic field does not follow Maxwell's equations it shares the magnetic field property of being the curl of a vector potential, and as will be clear in section 4.2 also magnetic properties in how it establishes dynamics. For these reasons the monopoles present at the degeneracies allow us to study, in a sense, the action of Maxwellian magnetic monopoles. It may be worthwhile to emphasize that the singular nature of the field at the energy degeneracies, that the field is there undefined, is crucial to the nonzero divergence. This mimics the model of electrical point charges, but also makes the dependence on the adiabatic approximation abundantly clear.

2.3 Regarding the monopoles

The system outlined in section 2.1 forms a sort of minimal working example of a synthetic field as well. The energies of different states are there degenerate only for an external magnetic field of zero magnitude, that is at the origin of parameter space, so the synthetic field has a monopole precisely at the origin. The action from the synthetic field, that is the accumulation of geometrical phase, could be achieved either from slowly varying the external field magnitude and direction or likewise from the movement of the particle through an inhomogeneous external field yielding the same effect. It is further the case for this simple example that the synthetic field not only contains monopoles, but that it is *purely* monopolar in origin. This is to say that a "charge" placed at the origin emanating a field that falls off as the inverse square of the distance yields precisely the synthetic field in question, the field is only due to "charges"[4]. It is important to note that this must not always be the case, the synthetic magnetic field may, depending on the Hamiltonian, take a form which is not possible to describe only through inverse-square falloff from placed charges. It could even be the case that no such charges are present at all, what the synthetic field simply does is allow them.

One case for which the synthetic field is *not* purely monopolar is two spin constituents interacting with both an external magnetic field and each other as shown by Eriksson and Sjöqvist [5], much like the system outlined in section 1.2. Eriksson and Sjöqvist further note that the nonzero curl of a synthetic field that is not purely monopolar can be used to extend the allegory between synthetic and Maxwellian magnetic fields through a synthetic "current" defined through the curl of the field. The main effect of spin-spin interaction of composite spin systems is however the "splitting" or movement of magnetic charge away from the origin of parameter space yielding more exotic fields, also shown by Eriksson and Sjöqvist. This movement of magnetic monopoles is continuous with respect to the spin-spin interaction parameters, and will be fully determined by these parameters. Since the system outlined in section 1.2 contains precisely an Ising interaction along some select axis the synthetic field cannot be fully spherically symmetric, but must have a rotational symmetry determined by said axis.

For the sake of this body of work we note but that the synthetic field of the dumbbell model will be nontrivial in nature, carries distinct synthetic monopoles and further may exhibit nonzero curl, so that it is not purely monopolar.

3 System description

3.1 Coordinates and quantities

Now follows a complete model of the scenario outlined in section 1.2, with the purpose of simulating the system numerically to gain insights into the dynamics of synthetic magnetic fields. Consider then a dumbbell-like system consisting of two equal masses at a distance l from one another, and let m be the total mass. The system can be freely translated and rotated throughout space, so let x, y, z be the position of the centre off mass, and θ_r, φ_r be the polar and azimuthal angle respectively of the axis connecting the two masses. Notate these coordinates compactly as the vector

$$\vec{\mathbf{r}} = \begin{pmatrix} x \\ y \\ z \\ \vartheta_r \\ \varphi_r \end{pmatrix}.$$

Fix the angles such that a polar angle of $\varphi_r = m_0$ implies a dumbbell parallell to the z-axis and so that an azimuthal angle of $\vartheta_r = 0$ implies that the dumbbell axis lies in the xz-plane.

Now, consider also each of the masses of the dumbbell to carry spin, intrinsic angular momentum, of size $\frac{1}{2}$ each. The state of the spin components must be described quantum mechanically, so let $|s, m'\rangle$ denote the state of the system with *total* spin magnitude squared $s(s+1)\hbar^2$ and *total* spin measured along the z-axis $\hbar m'$. Note that the spin quantum number s will be 1 for the composite system, and so values of m' will range from -1 to 1. An external field $\vec{\mathbf{B}}$ is present, which we can describe by its magnitude B and its angular direction ϑ_B, φ_B in analogue with the angles defined above.

3.2 The Hamiltonian

The time evolution of such a system is governed in both classical and quantum mechanics by its Hamiltonian. Since spin is the epitome of a phenomena demanding a quantum mechanical interpretation we have no choice but to model the whole system quantum mechanically. The Hamiltonian which will be assumed for the system here is:

$$\mathcal{H} = \sum_{i=1}^{5} \frac{\vec{\mathbf{p}}_\mathbf{i}^{\,2}}{2m_i} + \frac{4J}{\hbar} S_\mu^{(1)} S_\mu^{(2)} - \gamma \vec{\mathbf{B}} \cdot \vec{\mathbf{S}}. \tag{5}$$

The first sum is over the five degrees of translational and rotational coordinates in $\vec{\mathbf{r}}$. The momentum and angular momentum operators are taken to be $p_i = i\hbar\partial_i$ with ∂_i as the derivative with respect to the corresponding coordinate. Note that it is not a priori clear that the effective masses m_i for all degrees of freedom are the same, but we can until later note that at least the first three are equal to m.

For the potential energy the spin-spin interaction is taken to be of Ising form, which is the first term after the sum, while the interaction between spin and magnetic field is considered in the final term. An Ising interaction contains a preferred axis of interaction, which for symmetry reasons of the system has been chosen to be the direction of the dumbbell axis μ, the axis connecting the two masses. The parameters J and γ are the strengths of both of these interactions, while the operators $\vec{\mathbf{S}}$ and $S_\mu^{(n)}$ are respectively the one related to the total spin of the system and the spin in the μ-direction of one of the system components. Note that the parameter γ much like the example of section 2.1 typically looks like $\gamma = \frac{g\mu_f}{\hbar}$ where g is a g-factor and μ_f is some appropriate magneton.

Similar systems as the one considered here has been has been studied before, in particular a bipartite spin-$\frac{1}{2}$ system with coordinate-fixed Ising axis by Sjöqvist and Yi [10]. If the rotational degrees of the present system are ignored and the coordinate ϑ_r is set to 0 whenever present the system of Sjöqvist and Yi will be matched in full, save for that the external field is there varied directly instead of through centre of mass motion.

3.3 Effective mass

To clearly see the values of the effective masses paired with the rotational momenta a quick derivation of the kinetic part of the Hamiltonian is in order. The kinetic energy related to rotation is of the form

$$K_{rot} = \frac{m}{2}\left(\frac{l}{2}\right)^2 \left(\dot{\vartheta}_r^2 + \dot{\varphi}_r^2\right),$$

which is the same as the relevant terms of the Lagrangian. The quantum mechanical momenta correspond to the momenta received from differentiating the classical Lagrangian, and as of such we have in the classical picture that

$$p_4 = \frac{\partial K}{\partial \dot{\vartheta}_r} = \frac{ml^2}{4}\dot{\vartheta}_r$$
$$p_5 = \frac{\partial K}{\partial \dot{\varphi}_r} = \frac{ml^2}{4}\dot{\varphi}_r$$

.

Performing the Legendre transform from the Lagrangian to the Hamiltonian yields:

$$\mathcal{H}_{rot} = p_4\dot{\vartheta}_r + p_5\dot{\varphi}_r - K = \frac{p_4^2 + p_5^2}{2}\frac{4}{ml^2}.$$

It is then clear that the effective masses to be used in equation 5 are:

$$m_i = \begin{cases} m & i = 1,2,3 \\ \frac{ml^2}{4} & i = 4,5 \end{cases}.$$

3.4 Rotation matrices

The potential energy operators will be of great use in some matrix form, so let the spin state of the entire system be described as a coordinate vector in the basis $(|0,0\rangle, |1,-1\rangle, |1,0\rangle, |1,1\rangle)$ with the coordinate z-axis as the spin measurement direction. In the special case where the axis of the dumbbell (henceforth "Ising axis") and the magnetic field $\vec{\mathbf{B}}$ is parallell to the z-axis, it is clear that the operators take the form:

$$\vec{\mathbf{B}} \cdot \vec{\mathbf{S}} = B\hbar \begin{pmatrix} 0 & & & \\ & -1 & & \\ & & 0 & \\ & & & 1 \end{pmatrix} \tag{6}$$

$$S_\mu^{(1)} S_\mu^{(2)} = \frac{\hbar^2}{4} \begin{pmatrix} -1 & & & \\ & 1 & & \\ & & -1 & \\ & & & 1 \end{pmatrix}. \tag{7}$$

The second matrix follows from the well known representation of a two-component spin-$\frac{1}{2}$ system as singlet and triplet states: Let $|m_1, m_2\rangle_1$ be the state with spin-z number m_1 for the first component and m_2 for the second component. Then

$$\begin{aligned} |0,0\rangle &= \frac{1}{\sqrt{2}}\left[\left|\frac{1}{2},-\frac{1}{2}\right\rangle_1 - \left|-\frac{1}{2},\frac{1}{2}\right\rangle_1\right] \\ |1,-1\rangle &= \left|-\frac{1}{2},-\frac{1}{2}\right\rangle_1 \\ |1,0\rangle &= \frac{1}{\sqrt{2}}\left[\left|\frac{1}{2},-\frac{1}{2}\right\rangle_1 + \left|-\frac{1}{2},\frac{1}{2}\right\rangle_1\right] \\ |1,1\rangle &= \left|\frac{1}{2},\frac{1}{2}\right\rangle_1. \end{aligned}$$

Both matrices above assume that the basis is aligned with $\vec{\mathbf{B}}$ and the Ising axis respectively. Therefore we must find some rotation operator that can describe our state given in the z-axis basis in a basis aligned with $\vec{\mathbf{B}}$ or the Ising axis.

Consider therefore first a rotation of the *state* vectors, which can then easily be inverted to receive the forward transformation also necessary for the transformation of operator matrices. The inversion process is but a complex conjugation since the operator in question is unitary. It can be shown that the rotation about three Euler angles α, β, δ of a state is given by the matrix with elements as[11]:

$$\mathcal{U}_{m'm''} = \langle s, m'| e^{\frac{-iS_\mu \alpha}{\hbar}} e^{\frac{-iS_y \beta}{\hbar}} e^{\frac{-iS_\mu \delta}{\hbar}} |s, m''\rangle.$$

Here s, m', m'' are spin quantum numbers of the system, which in the more general case can be replaced by angular momentum quantum numbers. The rotations α, β and δ are done about the $z-$, $y-$ and then $z-$ body axes of the system in turn. Since the spin states considered here are symmetric about their body z-axes the final rotation δ is superfluous and thus will be discarded. Identifying the angles $\alpha = \varphi$ and $\beta = \vartheta$ for rotation to some spherical coordinates it can further be shown that the exponential operators amount to:

$$\mathcal{U} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{e^{-i\varphi}}{2}(1+\cos(\vartheta)) & \frac{e^{-i\varphi}}{\sqrt{2}}\sin(\vartheta) & \frac{e^{-i\varphi}}{2}(1-\cos(\vartheta)) \\ 0 & -\frac{1}{\sqrt{2}}\sin(\vartheta) & \cos(\vartheta) & \frac{1}{\sqrt{2}}\sin(\vartheta) \\ 0 & \frac{e^{i\varphi}}{2}(1-\cos(\vartheta)) & -\frac{e^{i\varphi}}{\sqrt{2}}\sin(\vartheta) & \frac{e^{i\varphi}}{2}(1+\cos(\vartheta)) \end{pmatrix}.$$

An operator matrix A transforms under rotation as $A_{rot} = \mathcal{U}A\mathcal{U}^\dagger$, so the operator of equation 6 which is expressed in terms of a basis rotated by angles ϑ_B and φ_B can be written in the z-axis basis as:

$$\vec{\mathbf{B}} \cdot \vec{\mathbf{S}} = B\hbar \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -\cos(\vartheta_B) & \frac{e^{-i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) & 0 \\ 0 & \frac{e^{i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) & 0 & \frac{e^{-i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) \\ 0 & 0 & \frac{e^{i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) & \cos(\vartheta_B) \end{pmatrix}. \tag{8}$$

Analogously the matrix of equation 7 is expressed in a basis rotated through angles ϑ_r and φ_r, so in the z-axis basis it can be written:

$$S_\mu^{(1)} S_\mu^{(2)} = \frac{\hbar^2}{4} \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & \cos^2(\vartheta_r) & -\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & e^{-2i\varphi_r}\sin^2(\vartheta_r) \\ 0 & -\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & -\cos(2\vartheta_r) & \frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) \\ 0 & e^{2i\varphi_r}\sin^2(\vartheta_r) & \frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & \cos^2(\vartheta_r) \end{pmatrix}. \tag{9}$$

In equation 8 and 9 it is readily visible that the spin singlet state $|0,0\rangle$ is unaffected by the external magnetic field, as could be concluded even without the explicit Hamiltonian. As a result the total Hamiltonian for the singlet state is but the sum of two terms dependent on different sets of variables. This is to say that separation of variables can be used to solve the eigenstate problem, so the qualities presently at interest are lost. For this reason henceforth only the non-singlet (so called triplet) states are considered, and matrices will often be reduced to the relevant three-dimensional subspace for simplicity's sake.

At last all parts of the potential energy are expressed in a single basis, such that the potential part of the Hamiltonian takes the form:

$$\mathcal{H}_f = \gamma B\hbar \begin{pmatrix} \xi\cos^2(\vartheta_r) + \cos(\vartheta_B) & -\frac{e^{-i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) - \xi\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & \xi e^{-2i\varphi_r}\sin^2(\vartheta_r) \\ -\frac{e^{i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) - \xi\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & -\xi\cos(2\vartheta_r) & -\frac{e^{-i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) + \xi\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) \\ \xi e^{2i\varphi_r}\sin^2(\vartheta_r) & -\frac{e^{i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) + \xi\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & \xi\cos^2(\vartheta_r) - \cos(\vartheta_B) \end{pmatrix}. \tag{10}$$

Here $\xi = \frac{J}{\gamma B}$ is a proportionality factor between the spin-spin and spin-field interactions which will simplify calculations. This together with the kinetic part of the Hamiltonian $\mathcal{H}_s$ will determine the time evolution of the system:

$$\mathcal{H}_s = \sum_{n=1}^{5} \frac{p_i^2}{2m_i}. \tag{11}$$

4 The Born-Oppenheimer approximation

4.1 Derivation

Solving for the eigenstates of such a Hamiltonian as described above is a mighty task. Note in particular that the contribution from potential energy to the Hamiltonian, equation 10, is heavily dependent on the position and rotation of the dumbbell through all parameters ϑ_r, ϑ_B, φ_r, φ_B and B. This couples all degrees of freedom for the system, which complicates the problem greatly.

An approximation is therefore in order. If the position and rotation, henceforth the "slow" parameters, is more or less static in comparison with the spin degrees of freedom, henceforth "fast parameters", the so-called Born-Oppenheimer approximation is applicable. A version of the adiabatic approximation, it assumes that a "fast" subsystem, described by the fast parameters, can be described by eigenstates to a Hamiltonian parametrised by the slow parameters. A fast system in such an eigenstate can be considered to remain in the same eigenstate as the associated "fast" Hamiltonian slowly changes, changing its eigenvalue as the slow parameters evolve. So far this is what is known as the adiabatic approximation[7]. In the terminology used for geometric phases the slow parameters then form the parameter space of the fast system.

The Born-Oppenheimer approximation involves the extension of this to also consider how the "slow" system evolves, in practice finding an effective Hamiltonian to the slow system as well. Originally, and

still mainly, an approximation used in molecular physics proposed in 1927 [12] it applies also to the present situation. The full system is then considered to be described by the product of a wave function to the slow system Ψ_s and some eigenstate to the fast Hamiltonian $|n\rangle$, i.e.:

$$|\Psi_{full}\rangle = \Psi_s |n\rangle .$$

The aforementioned fast and slow Hamiltonians are for the system in consideration the previously found $\mathcal{H}_f$ and $\mathcal{H}_s$ respectively. The full solution to the fast system is assumed to be found, i.e.:

$$\mathcal{H}_f |n\rangle = E_n |n\rangle .$$

The Schrödinger equation then implies, since $\frac{\partial}{\partial t}|n\rangle = 0$:

$$\begin{aligned} i\hbar\frac{\partial}{\partial t}(\Psi_s |n\rangle) &= (\mathcal{H}_f + \mathcal{H}_s)\Psi_s |n\rangle \\ i\hbar\frac{\partial \Psi_s}{\partial t} &= \langle n| (\mathcal{H}_f + \mathcal{H}_s) |n\rangle \Psi_s = (\langle n| \mathcal{H}_s |n\rangle + E_n)\Psi_s. \end{aligned}$$

This can be interpreted as an effective Hamiltonian $\mathcal{H}_{eff} = \langle n| \mathcal{H}_s |n\rangle + E_n$ governing the slow wave function. The inner product term can be further manipulated as follows:

$$\langle n| \mathcal{H}_s |n\rangle \Psi_s = \sum_{i=1}^{5} \left[\langle n| \frac{p_i^2}{2m_i} |n\rangle \Psi_s + \langle n| \frac{p_i}{m_i} |n\rangle p\Psi_s + p_i^2 \Psi \right].$$

Here it is to be understood that the momentum operators in $\mathcal{H}_s$ act on *both* the spin ket and and the slow wave fucntion. An operator to the left of a ket and wave function product will however be understood to act on the ket only, if no clarifying parentheses are written out explicitly. Since all p_i are hermitian operators and can thus be acted on bras to the left without conjugation the most troublesome term can also be written as:

$$\begin{aligned} \langle n| p_i^2 |n\rangle &= \langle p_i n | p_i n\rangle \\ &= \langle p_i n|n\rangle \langle n|p_i n\rangle + \langle p_i n| (\mathbb{1} - |n\rangle \langle n|) |p_i n\rangle \\ &= \langle n|p_i n\rangle^2 + \langle p_i n| (\mathbb{1} - |n\rangle \langle n|) |p_i n\rangle . \end{aligned}$$

An identity relation was used in the second step, where $\mathbb{1}$ is the identity operator. Inserting the derivative form of the momentum operator as seen in section 3.2 and rearranging terms with some convenient notation we arrive to the Hamiltonian providing the interesting properties sought after.

$$\mathcal{H}_{eff} = \sum_{i=5}^{5} \frac{(p_i - A_i)^2}{2m_i} + \Phi + E_n \tag{12}$$

$$A_i = i\hbar \langle n|\partial_i n\rangle \tag{13}$$

$$\Phi = \sum_{i=1}^{5} \frac{\hbar^2}{2m_i} \langle \partial_i n| (\mathbb{1} - |n\rangle \langle n|) |\partial_i n\rangle . \tag{14}$$

4.2 Interpretation

Equation 12 has been aptly written on a form which suggests the physics to be studied. Note that the sum over i looks precisely like the action of a magnetic field with vector potential $\vec{\mathbf{A}} = \sum_{i=1}^{5} i\hbar \left\langle n \middle| \vec{\boldsymbol{\nabla}} n \right\rangle$ on a particle of charge 1 and momentum $\vec{\mathbf{p}} = \sum_{i=1}^{5} p_i$. Note also that this magnetic field is the same as the synthetic magnetic field outlined in section 2.2, and as of such carries precisely the same properties. In particular the field will carry a monopolar dependence, as desired. A difference present to Maxwellian magnetic fields is that both field and momentum are here five dimensional, and furthermore that the masses of the two final degrees of freedom are rather moments of inertia. The dynamics of this term is the primal interest to this discussion, but we note also a scalar field, analogously called the synthetic electric field, or the synthetic scalar field. Roughly speaking however this field is a factor $\hbar$ smaller than the synthetic magnetic field and will in most cases be negligible.

It can however be shown that the scalar field acts as a repulsive inverse square force near degeneracies in the fast Hamiltonian[13]. The inverse square dependence to the distance of a degeneracy point means that the scalar field will have appreciable effects if the slow parameters are close enough to the degeneracy, and furthermore the repulsive nature actually leads to a strengthening of the Born-Oppenheimer approximation as the adiabatic approximation loses validity at points of degeneracy.

4.3 Dynamics

Having found an effective Hamiltonian to the slow system the application of this Hamiltonian to the dynamics of the system remains to be performed. One could proceed with the quantum mechanical methods applied so far, solving for eigenstates to $\mathcal{H}_{eff}$. This however requires finding the dependence of the eigenstates to $\mathcal{H}_f$ on the slow parameters, which may not be achievable analytically. Instead the slow system can be considered to be effectively lying in the classical domain, and the Hamiltonian derived by quantum mechanical means will be utilized in the role of the Hamiltonian for classical mechanics.

Hamilton's canonical equations indicate the time evolution of $\vec{\mathbf{r}}$:

$$\frac{\partial \vec{\mathbf{r}}}{\partial t} = \frac{\partial \mathcal{H}_{eff}}{\partial \vec{\mathbf{p}}} = \frac{\vec{\mathbf{p}} - \vec{\mathbf{A}}}{m}$$
$$\frac{\partial \vec{\mathbf{p}}}{\partial t} = -\frac{\partial \mathcal{H}_{eff}}{\partial \vec{\mathbf{r}}} = \left(\frac{\partial \vec{\mathbf{A}}}{\partial \vec{\mathbf{r}}}\right)^T \frac{\vec{\mathbf{p}} - \vec{\mathbf{A}}}{m} - \frac{\partial \Phi}{\partial \vec{\mathbf{r}}} - \frac{\partial E_n}{\partial \vec{\mathbf{r}}} = \left(\frac{\partial \vec{\mathbf{A}}}{\partial \vec{\mathbf{r}}}\right)^T \frac{\partial \vec{\mathbf{r}}}{\partial t} - \frac{\partial \Phi}{\partial \vec{\mathbf{r}}} - \frac{\partial E_n}{\partial \vec{\mathbf{r}}}.$$

Note in particular that the first of these equations imply that the canonical momentum $\vec{\mathbf{p}}$ is *not* $m\frac{\partial \vec{\mathbf{r}}}{\partial t}$. The effective force acting on the system can be found, utilizing that the synthetic vector potential does not depend explicitly on time, i.e. that $\frac{\partial \vec{\mathbf{A}}}{\partial t} = 0$:

$$m\frac{\partial^2 \vec{\mathbf{r}}}{\partial t^2} = \frac{\partial \vec{\mathbf{p}}}{\partial t} - \frac{\partial \vec{\mathbf{A}}}{\partial t} = \left(\frac{\partial \vec{\mathbf{A}}}{\partial \vec{\mathbf{r}}}\right)^T \frac{\partial \vec{\mathbf{r}}}{\partial t} - \left(\frac{\partial \vec{\mathbf{r}}}{\partial t} \cdot \vec{\boldsymbol{\nabla}}\right)\vec{\mathbf{A}} - \frac{\partial \Phi}{\partial \vec{\mathbf{r}}} - \frac{\partial E_n}{\partial \vec{\mathbf{r}}}. \tag{15}$$

The Jacobian matrix can be treated elementwise, as well as the second term:

$$\frac{1}{i\hbar}\left(\frac{\partial \vec{\mathbf{A}}}{\partial \vec{\mathbf{r}}}\right)_{ji} = \partial_i \langle n|\partial_j n\rangle = \langle \partial_i n|\partial_j n\rangle + \langle n|\partial_i \partial_j n\rangle$$
$$\frac{1}{i\hbar}\left(\left(\frac{\partial \vec{\mathbf{r}}}{\partial t} \cdot \vec{\boldsymbol{\nabla}}\right)\vec{\mathbf{A}}\right)_i = \sum_{j=1}^{5} \frac{\partial r_j}{\partial t}\partial_j \langle n|\partial_i n\rangle = \sum_{j=1}^{5} \frac{\partial r_j}{\partial t}(\langle \partial_j n|\partial_i n\rangle + \langle n|\partial_j \partial_i n\rangle).$$

Insertion into equation 15 then yields a higher dimensional analogue to [the usual case] for the forces due to the synthetic magnetic field:

$$\begin{aligned}\frac{1}{i\hbar}\left[\left(\frac{\partial \vec{\mathbf{A}}}{\partial \vec{\mathbf{r}}}\right)^T \frac{\partial \vec{\mathbf{r}}}{\partial t} - \left(\frac{\partial \vec{\mathbf{r}}}{\partial t} \cdot \vec{\boldsymbol{\nabla}}\right)\vec{\mathbf{A}}\right]_i &= \frac{1}{i\hbar}F_i^A = \sum_{j=1}^{5} \frac{\partial r_j}{\partial t}[\langle \partial_i n|\partial_j n\rangle - \langle \partial_j n|\partial_i n\rangle] \\ &= \sum_{j=1}^{5}\sum_{l} \frac{\partial r_j}{\partial t}[\langle \partial_i n|l\rangle \langle l|\partial_j n\rangle - \langle \partial_j n|l\rangle \langle l|\partial_i n\rangle] \\ &= \sum_{j=1}^{5}\sum_{l\neq n} \frac{\partial r_j}{\partial t}[\langle \partial_i n|l\rangle \langle l|\partial_j n\rangle - \langle \partial_j n|l\rangle \langle l|\partial_i n\rangle] \\ &= \sum_{j\neq i}\sum_{l\neq n} \frac{\partial r_j}{\partial t}[\langle \partial_i n|l\rangle \langle l|\partial_j n\rangle - \langle \partial_j n|l\rangle \langle l|\partial_i n\rangle].\end{aligned}$$

Here $|l\rangle$ simply denotes an eigenstate to $\mathcal{H}_f$ of some index l, and the sum over l is over all available states. The exclusion of $n = l$-terms follows as $\langle \partial_i n|n\rangle$ is pure imaginary, which can be seen from differentiating $\langle n|n\rangle$. This rearrangement is highly desirable, for it now so happens that this allows us to take derivatives of the Hamiltonian instead of the rather tricky differentiation of the eigenkets.

Differentiating the Schrödinger equation and acting on it with some other eigenbra yields:

$$\begin{aligned}\mathcal{H}_f \left|n\right\rangle &= E_n \left|n\right\rangle \\ \partial\mathcal{H}_f \left|n\right\rangle + H_f \left|\partial n\right\rangle &= E_n \left|\partial n\right\rangle \\ \left\langle l\right| \partial\mathcal{H}_f \left|n\right\rangle &= \left\langle l|\partial n\right\rangle (E_n - E_l).\end{aligned}$$

Rearranging, a very useful relation emerges:

$$\left\langle l|\partial n\right\rangle = \frac{\left\langle l\right| \partial\mathcal{H}_f \left|n\right\rangle}{E_n - E_l}. \tag{16}$$

This we can insert into above:

$$\frac{1}{i\hbar}F_i^A = \sum_{j\neq i}\sum_{l\neq n} \frac{\frac{\partial r_j}{\partial t}}{(E_n - E_l)^2}[\left\langle n\right| \partial_i\mathcal{H}_f \left|l\right\rangle \left\langle l\right| \partial_j\mathcal{H}_f \left|n\right\rangle - \left\langle n\right| \partial_j\mathcal{H}_f \left|l\right\rangle \left\langle l\right| \partial_i\mathcal{H}_f \left|n\right\rangle] \tag{17}$$

$$= 2i\sum_{j\neq i}\sum_{l\neq n} \frac{\frac{\partial r_j}{\partial t}}{(E_n - E_l)^2} \operatorname{Im}[\left\langle n\right| \partial_i\mathcal{H}_f \left|l\right\rangle \left\langle l\right| \partial_j\mathcal{H}_f \left|n\right\rangle]. \tag{18}$$

Equation 16 can be used for something similar when evaluating the synthetic electric potential:

$$\Phi = \sum_{i=1}^{5}\sum_{l\neq n} \frac{\hbar^2}{2m_i}(\left\langle \partial_i n|l\right\rangle \left\langle l|\partial_i n\right\rangle) = \sum_{i=1}^{5}\sum_{l\neq n} \frac{\hbar^2}{2m_i} \frac{\left\langle n\right| \partial_i\mathcal{H}_f \left|l\right\rangle \left\langle l\right| \partial_i\mathcal{H}_f \left|n\right\rangle}{(E_n - E_l)^2} \tag{19}$$

$$= \sum_{i=1}^{5}\sum_{l\neq n} \frac{\hbar^2}{2m_i} \frac{|\left\langle n\right| \partial_i\mathcal{H}_f \left|l\right\rangle|^2}{(E_n - E_l)^2}. \tag{20}$$

For the last equalities to hold in both contributions we require that the derivative of the Hamiltonian is hermitian, but thankfully derivatives of hermitian operators are hermitian so this holds true. Note however that no simple form to the derivative of the electric potential Φ has been found, which might not be easily described analytically.

The problem has thus been reduced to evaluating, per equations 17 and 19,

$$m\frac{\partial^2\vec{\mathbf{r}}}{\partial t^2} = \vec{\mathbf{F}}^A - \frac{\partial\Phi}{\partial\vec{\mathbf{r}}} - \frac{\partial E_n}{\partial\vec{\mathbf{r}}}. \tag{21}$$

4.4 Differentiation of the Hamiltonian

In order to easily evaluate equations 17 and 19 derivatives of $\mathcal{H}_f$ from equation 10 are to be found. Writing any of the coordinates x, y, z as r the derivatives can be written:

$$\partial_r\mathcal{H}_f = \gamma B\hbar \begin{pmatrix} \frac{\dot{B}}{B}\cos(\vartheta_B) - \dot{\vartheta}_B\sin(\vartheta_B) & \Omega & 0 \\ \Omega^* & 0 & \Omega \\ 0 & \Omega^* & -\frac{\dot{B}}{B}\cos(\vartheta_B) + \dot{\vartheta}_B\sin(\vartheta_B) \end{pmatrix} \tag{22}$$

$$\partial_{\vartheta_r}\mathcal{H}_f = \gamma B\xi\hbar \begin{pmatrix} -\sin(2\vartheta_r) & -\sqrt{2}e^{-i\varphi_r}\cos(2\vartheta_r) & e^{-2i\varphi_r}\sin(2\vartheta_r) \\ -\sqrt{2}e^{i\varphi_r}\cos(2\vartheta_r) & 2\sin(2\vartheta_r) & \sqrt{2}e^{-i\varphi_r}\cos(2\vartheta_r) \\ e^{2i\varphi_r}\sin(2\vartheta_r) & \sqrt{2}e^{i\varphi_r}\cos(2\vartheta_r) & -\sin(2\vartheta_r) \end{pmatrix} \tag{23}$$

$$\partial_{\varphi_r}\mathcal{H}_f = \gamma B\xi\hbar \begin{pmatrix} 0 & i\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & -2ie^{-2i\varphi_r}\sin^2(\vartheta_r) \\ -i\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & 0 & -i\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) \\ 2ie^{2i\varphi_r}\sin^2(\vartheta_r) & i\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & 0 \end{pmatrix}. \tag{24}$$

Here $(-\frac{\dot{B}}{B}\sin(\vartheta_B) + i\dot{\varphi}_B\sin(\vartheta_B) - \dot{\vartheta}_B\cos(\vartheta_B))\frac{e^{-i\varphi_B}}{\sqrt{2}}$ is introduced as a means of compressing the rather lengthy expressions for the derivative with respect to r, and asterisks signify the complex conjugate.

4.5 Solution of the fast subsystem

The usage of the Born-Oppenheimer approximation requires a solution to the fast subsystem, i.e. that the eigenvalues and eigenvectors to $\mathcal{H}_f$ are found. Unfortunately this is not possible analytically for the

present system, but we note that it is the same as solving the following transcendental characteristic equation, which follows from equation 10, for the eigenvalues $\lambda_n = \frac{E_n}{\gamma B \hbar}$:

$$0 = \begin{vmatrix} \xi\cos^2(\vartheta_r) + \cos(\vartheta_B) - \lambda_n & -\frac{e^{-i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) - \xi\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & \xi e^{-2i\varphi_r}\sin^2(\vartheta_r) \\ -\frac{e^{i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) - \xi\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & -\xi\cos(2\vartheta_r) - \lambda_n & -\frac{e^{-i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) + \xi\frac{e^{-i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) \\ \xi e^{2i\varphi_r}\sin^2(\vartheta_r) & -\frac{e^{i\varphi_B}}{\sqrt{2}}\sin(\vartheta_B) + \xi\frac{e^{i\varphi_r}}{\sqrt{2}}\sin(2\vartheta_r) & \xi\cos^2(\vartheta_r) - \cos(\vartheta_B) - \lambda_n \end{vmatrix}. \tag{25}$$

This is rather messy, and unfortunately does not bring much clarity to the behaviour of the eigenvalues.

Solving for the eigenvectors as a function of the eigenvalues does not yield any strikingly useful relation neither, so the calculation of eigenvalues and eigenvectors may be left to numerics from the onset. Equation 25 could in principle be differentiated implicitly to receive the derivatives of the energies also needed, but since simulation of the system requires many other quantities to also be calculated numerically the derivatives of the energies will be done likewise for practicality's sake.

References

[1] Paul Adrien Maurice Dirac. "Quantised singularities in the electromagnetic field," in: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 133.821 (1931), pp. 60–72. DOI: 10.1098/rspa.1931.0130. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1931.0130. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1931.0130.

[2] C. Castelnovo, R. Moessner, and S. L. Sondhi. "Magnetic monopoles in spin ice". In: *Nature* 451.7174 (2008), pp. 42–45. ISSN: 1476-4687. DOI: 10.1038/nature06433. URL: https://doi.org/10.1038/nature06433.

[3] M. W. Ray et al. "Observation of isolated monopoles in a quantum field". In: *Science* 348.6234 (2015), pp. 544–547. DOI: 10.1126/science.1258289. eprint: https://www.science.org/doi/pdf/10.1126/science.1258289. URL: https://www.science.org/doi/abs/10.1126/science.1258289.

[4] Michael Victor Berry. "Quantal phase factors accompanying adiabatic changes". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 392.1802 (1984), pp. 45–57. DOI: 10.1098/rspa.1984.0023. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1984.0023. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1984.0023.

[5] Andreas Eriksson and Erik Sjöqvist. "Monopole field textures in interacting spin systems". In: *Physical Review A* 101.5 (2020). DOI: 10.1103/physreva.101.050101. URL: https://doi.org/10.1103%2Fphysreva.101.050101.

[6] K.S. Krane, D. Halliday, and John Wiley & Sons. "Introductory Nuclear Physics". In: Wiley, 1988, p. 606. ISBN: 9780471805533. URL: https://books.google.se/books?id=ConwAAAAMAAJ.

[7] M. Born and V. Fock. "Beweis des Adiabatensatzes". In: *Zeitschrift für Physik* 51.3 (1928), pp. 165–180. ISSN: 0044-3328. DOI: 10.1007/BF01343193. URL: https://doi.org/10.1007/BF01343193.

[8] Y. Aharonov and J. Anandan. "Phase change during a cyclic quantum evolution". In: *Phys. Rev. Lett.* 58 (16 1987), pp. 1593–1596. DOI: 10.1103/PhysRevLett.58.1593. URL: https://link.aps.org/doi/10.1103/PhysRevLett.58.1593.

[9] Erik Sjöqvist. "Geometric phases in quantum information". In: *International Journal of Quantum Chemistry* 115.19 (2015), pp. 1311–1326. DOI: https://doi.org/10.1002/qua.24941. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.24941. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.24941.

[10] X. X. Yi and Erik Sjöqvist. "Effect of intersubsystem coupling on the geometric phase in a bipartite system". In: *Phys. Rev. A* 70 (4 2004), p. 042104. DOI: 10.1103/PhysRevA.70.042104. URL: https://link.aps.org/doi/10.1103/PhysRevA.70.042104.

[11] J. J. Sakurai and Jim Napolitano. *Modern Quantum Mechanics*. 3rd ed. Cambridge University Press, 2020. DOI: 10.1017/9781108587280.

[12] M. Born and R. Oppenheimer. "Zur Quantentheorie der Molekeln". In: *Annalen der Physik* 389.20 (1927), pp. 457–484. DOI: https://doi.org/10.1002/andp.19273892002. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.19273892002. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.19273892002.

[13] M V Berry and R Lim. "The Born-Oppenheimer electric gauge force is repulsive near degeneracies". In: *Journal of Physics A: Mathematical and General* 23.13 (1990), pp. L655–L657. DOI: 10.1088/0305-4470/23/13/004. URL: https://doi.org/10.1088/0305-4470/23/13/004.

A Code

Below follow the scripts used for numerical simulation of the described dumbbell system, all written in Python. The code is divided into three files: one containing the tools for constructing the external magnetic fields used, one containing the tools used to integrate the dynamics within some such field and one main script for setting parameters and calling the necessary functions from the two other modules. To execute properly all three .py - files must be placed in the same folder additionally containing a subfolder named ”saves”, and that subfolder must in turn contain three subfolders ”fields”, ”odesols” and ”graphs”.

A.1 magfield.py

```
import pandas as pd
import xarray as xr
import numpy as np
import matplotlib.pyplot as plt
from scipy.constants import mu_0 as mu
from scipy.constants import pi as pi

def simplewire(nr, lablength, I, overwrite=False):
    ##Returns a placeholder field corresponding to a current through a wire
        along the
    ##x-axis. Saves the field and won't generate a preexisting field unless
    ##'overwrite=True' is called. Takes nr as the number of points along
        each axis.

    #Initiate variables:
    stepr = lablength/(nr-1) #Length of each lattice site
    generate = False #Whether the field needs to be generated
    wire = np.array([0,0]) #Position in x and y of current

    try: #Try to load pregenerated field
        field = np.load(f'saves/simplewire{I}field{nr},{lablength}.npy')
    except FileNotFoundError:
        generate = True

    #Returned saved field unless not found or overwrite turned on
    if not (generate or overwrite):
        #Return pregenerated field
        print("Loading saved magnetic field")
        return field
    else:
        #Generate field
        print("Generating magnetic field")

        xx, yy, zz = np.mgrid[0:nr, 0:nr, 0:nr]
        distance = [(xx - xx)*stepr, (yy - wire[0])*stepr, (zz - wire[1])*
            stepr]
        #Use Biot-Savart formula to calculate the magnetic field
        Bx, By, Bz = np.zeros([nr, nr, nr]), np.zeros([nr, nr, nr]), np.
            zeros([nr, nr, nr])
        Bx, By, Bz = mu/(4*pi)*2*I*np.cross([1, 0, 0], distance,
                axisb=0, axisc=0)/griddot(distance, distance)
        Br, Bth, Bph = cart_to_sph(np.array([Bx,By,Bz])) #Express as
            spherical coordinates
        #field = xr.DataArray([Bx, By, Bz, Br, Bth, Bph], dims=("direc", "x
            ", "y", "z"), coords={"direc":
        field = np.array((Bx, By, Bz, Br, Bth, Bph))
```

```
        np.save(f'saves/simplewire{I}field{nr},{lablength}.npy', field)
        #field.to_netcdf("saves/simplewirefield.nc")
        return field

def oppositecoils(nr, lablength, I, overwrite=False):
    ##Returns a field corresponding to two currents of opposing directions
        through square coils
    ##placed orthogonally to the z-axis centred 1/4th from the edges. Saves
         the field and won't generate a preexisting field unless
    ##'overwrite=True' is called. Takes nr as the number of points along
        each axis.

    #Initiate variables:
    stepr = lablength/(nr-1) #Length of each lattice site
    generate = False #Whether the field needs to be generated

    #Positions of all flowing currents below:
    #qindex = int(np.floor(nr/4))
    qindex = int(nr/3)
    wire1 = np.array([-qindex,-qindex]) #Current in positive x close to z=0
         and y=0 (pos in y,z)
    wire2 = np.array([-qindex,nr-1+qindex]) #Current in positive y close to
         z=0 and far from x=0 (pos in z,x)
    wire3 = np.array([nr-1+qindex,-qindex]) #Current in negative x close to
         z=0 and far from y=0 (pos in y,z)
    wire4 = np.array([-qindex,-qindex]) #Current in negative y close to z=0
         and close to x=0 (pos in z,x)
    wire5 = np.array([-qindex,nr-1+qindex]) #Current in negative x far from
         z=0 and close to y=0 (pos in y,z)
    wire6 = np.array([nr-1+qindex,nr-1+qindex]) #Current in negative y far
        from z=0 and x=0 (pos in z,x)
    wire7 = np.array([nr-1+qindex,nr-1+qindex]) #Current in positive x far
        from z=0 and y=0 (pos in y,z)
    wire8 = np.array([nr-1+qindex,-qindex]) #Current in positive y far from
         z=0 and close to x=0 (pos in z,x)

    try: #Try to load pregenerated field
        field = np.load(f'saves/fields/oppositecoils{I}field{nr},{lablength
            }.npy')
    except FileNotFoundError:
        generate = True

    #Returned saved field unless not found or overwrite turned on
    if not (generate or overwrite):
        #Return pregenerated field
        print("Loading saved magnetic field")
        return field
    else:
        #Generate field
        print("Generating magnetic field")

        xx, yy, zz = np.mgrid[0:nr, 0:nr, 0:nr]
        #Integrate the field per Biot-Savart along all currents
        Bx, By, Bz = np.zeros([nr, nr, nr]), np.zeros([nr, nr, nr]), np.
            zeros([nr, nr, nr])

        #Currents along x:
```

```
for px in range(-qindex, nr+qindex): #Maybe adjust with a +1
    dx = [stepr, 0, 0]
    #wire1
    distance = [(xx-px)*stepr, (yy-wire1[0])*stepr, (zz-wire1[1])*
        stepr]
    dBx1, dBy1, dBz1 = (mu*I/(4*pi) * np.cross(dx, distance, axisb
        =0,
                        axisc=0)/(griddot(distance, distance)
                            **(3/2)))
    Bx += dBx1
    By += dBy1
    Bz += dBz1

    #wire3
    distance = [(xx-px)*stepr, (yy-wire3[0])*stepr, (zz-wire3[1])*
        stepr]
    dBx3, dBy3, dBz3 = -(mu*I/(4*pi) * np.cross(dx, distance, axisb
        =0,
                        axisc=0)/(griddot(distance, distance)
                            **(3/2)))
    Bx += dBx3
    By += dBy3
    Bz += dBz3

    #wire5
    distance = [(xx-px)*stepr, (yy-wire5[0])*stepr, (zz-wire5[1])*
        stepr]
    dBx5, dBy5, dBz5 = -(mu*I/(4*pi) * np.cross(dx, distance, axisb
        =0,
                        axisc=0)/(griddot(distance, distance)
                            **(3/2)))
    Bx += dBx5
    By += dBy5
    Bz += dBz5

    #wire7
    distance = [(xx-px)*stepr, (yy-wire7[0])*stepr, (zz-wire7[1])*
        stepr]
    dBx7, dBy7, dBz7 = (mu*I/(4*pi) * np.cross(dx, distance, axisb
        =0,
                        axisc=0)/(griddot(distance, distance)
                            **(3/2)))
    Bx += dBx7
    By += dBy7
    Bz += dBz7

#Currents along y:
for py in range(-qindex, nr+qindex): #Maybe adjust with a +1
    dy = [0, stepr, 0]
    #wire2
    distance = [(xx-wire2[1])*stepr, (yy-py)*stepr, (zz-wire2[0])*
        stepr]
    dBx2, dBy2, dBz2 = (mu*I/(4*pi) * np.cross(dy, distance, axisb
        =0,
                        axisc=0)/(griddot(distance, distance)
                            **(3/2)))
    Bx += dBx2
    By += dBy2
```

```
            Bz += dBz2

            #wire4
            distance = [(xx-wire4[1])*stepr, (yy-py)*stepr, (zz-wire4[0])*
                stepr]
            dBx4, dBy4, dBz4 = -(mu*I/(4*pi) * np.cross(dy, distance, axisb
                =0,
                                 axisc=0)/(griddot(distance, distance)
                                     **(3/2)))
            Bx += dBx4
            By += dBy4
            Bz += dBz4

            #wire6
            distance = [(xx-wire6[1])*stepr, (yy-py)*stepr, (zz-wire6[0])*
                stepr]
            dBx6, dBy6, dBz6 = -(mu*I/(4*pi) * np.cross(dy, distance, axisb
                =0,
                                 axisc=0)/(griddot(distance, distance)
                                     **(3/2)))
            Bx += dBx6
            By += dBy6
            Bz += dBz6

            #wire8
            distance = [(xx-wire8[1])*stepr, (yy-py)*stepr, (zz-wire8[0])*
                stepr]
            dBx8, dBy8, dBz8 = (mu*I/(4*pi) * np.cross(dy, distance, axisb
                =0,
                                 axisc=0)/(griddot(distance, distance)
                                     **(3/2)))
            Bx += dBx8
            By += dBy8
            Bz += dBz8

        Br, Bth, Bph = cart_to_sph(np.array([Bx,By,Bz])) #Express as
            spherical coordinates
        field = np.array((Bx, By, Bz, Br, Bth, Bph))

        np.save(f'saves/fields/oppositecoils{I}field{nr},{lablength}.npy',
            field)
        return field

def griddot(a, b):
    ##Returns the dot product for each point in the supplied grids a, b.
        Contracts the
    ##first dimension.
    result = np.zeros(a[0].shape) + 1e-20 #Super ugly bodge to fix NaN
    for i in range(len(a)):
        result += a[i]*b[i]

    result = np.where(result == 0, 1e-20, result)
    return result

def cart_to_sph(cart):
    ##Returns spherical coordinates of the form(r, polar, azimuthal) for
        the given cartesian
```

```
    ##coordinates of the form (x,y,z) takes an array with coords as the
        first dimension.
    sph = np.zeros(cart.shape) #Initialize array
    xsqysq = cart[0]**2 + cart[1]**2 #Value of x^2 + y^2
    sph[0] = np.sqrt(xsqysq + cart[2]**2) #Radius r
    sph[1] = np.arctan2(np.sqrt(xsqysq), cart[2]) #Polar angle theta
    sph[2] = np.arctan2(cart[1],cart[0]) #Azimuthal angle phi
    return sph


def sliceplot(field):
    ##Plots the given field in the y-z-plane at x=0

    #Load the slice of B-values at x=0, indexed after y, z.
    By = field.sel(direc="By")[0,:,:]
    Bz = field.sel(direc="Bz")[0,:,:]
    #By = By.drop(dim="x")
    #Bz = Bz.drop(dim="x")
    y = np.arange(field.sizes["y"])
    z = np.arange(field.sizes["z"])

    fig = plt.figure()
    ax = fig.add_subplot(111)
    #Plot neat field lines
    color = np.array(2 * np.log(np.hypot(By, Bz) + 1e-10))
    #Note that streamplot unfortunately indexes the velocities as [vertical
        , horizontal],
    #such that this yields z on the horizontal and y on the vertical.
    ax.streamplot(z, y, Bz, By, linewidth=1,
            cmap=plt.cm.inferno, color=color, density=2, arrowstyle='->',
                arrowsize=1.5)

    #Touch up appearance and show plot
    plt.xlabel('Position along z-axis')
    plt.ylabel('Position along y-axis')
    plt.show()
```

A.2 synfieldtools.py

```
import pandas as pd
import numpy as np
import scipy
from matplotlib import pyplot as plt
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits import mplot3d
from scipy.constants import hbar
from scipy.integrate import solve_ivp
import warnings

##This is a collection of scripts related to the finding of eigenstates and
    thence the
##derivation of the synthetic fields.

#Common variables defined below:
ntheta = 250 #Number of points of theta
steptheta = np.pi/(ntheta-1) #Step size of theta
nphi = 500 #Number of points of phi
stepphi = 2*np.pi/(nphi) #Step size of phi
lablength = 1e-3 #Side length of environment cube in meters
```

```
tmax = 1 #Maximum simulated time in seconds

##In general x,y,z-coordinates are gives as index numbers, angles in
    radians

J = 1e9 #Spin-spin coupling strength
Gamma = 1e9 #Spin-field coupling strength

mass0 = 3.58e-25 #The total mass of the dumbbell in kg, as a placeholder
    this is the mass of two
                #silver atoms
len0 = 5e-10 #The distance between dumbbell edges in m
mass = np.repeat(mass0, 5) #Full mass vector
mass[3] = mass0*len0**2/4
mass[4] = mass[3]

tickcount=0 #For testing
dscalaravg=0
Faavg=0
denergyavg=0


##Returns the differentiated Hamiltonian w.r.t. the specified par. diffpar=
    r, th_r, ph_r at
##the point given in point=(x, y, z, th_r, ph_r ) in the external field
    field.
##If diffpar=r a list of matrices for derivatives w.r.t. x, y, z are
    returned
diffhamsave = {}
def diffhamiltonian(diffpar, point, field):

    #Check if the differentiated Hamiltonian has already been calculated
        here
    global diffhamsave
    if (diffpar, point) in diffhamsave:
        #print('diffhamsave used')
        return diffhamsave[(diffpar, point)]

    #Find nr and stepr
    nr = field.shape[1] #Number of points along each axis of the lattice
    stepr = lablength/(nr-1) #Distance between lattice points
    #Select derivative to return
    if(diffpar == "r"):

        #Initialize return matrices
        diff_hamx = np.zeros([3, 3], dtype='complex_')
        diff_hamy = np.zeros([3, 3], dtype='complex_')
        diff_hamz = np.zeros([3, 3], dtype='complex_')

            #First find coordinate values of all neighbouring sites
        pointx = int(point[0]) #Get point index (this is needed for dtype
            purposes)
        pointy = int(point[1])
        pointz = int(point[2])

        neighgrid = np.mgrid[-1:2,-1:2,-1:2] ##Meshgrid to generate
            neighbours
        neighgrid = np.array([pointx, pointy, pointz])[:,None,None,None] +
```

```
    neighgrid ##Uses broadcasting to duplicate
    #x,y,z into each point on the grid. The result has first
        dimension determining which
    #coordinate is given and the remaining specifying position
        related to the point

    #Find neighbouring r_B values
Bgrid = np.zeros([3,3,3])
Bgrid[1,1,1] = field[3, neighgrid[0,1,1,1], neighgrid[1,1,1,1],
    neighgrid[2,1,1,1]]
Bgrid[2,1,1] = field[3, neighgrid[0,2,1,1], neighgrid[1,2,1,1],
    neighgrid[2,2,1,1]]
Bgrid[0,1,1] = field[3, neighgrid[0,0,1,1], neighgrid[1,0,1,1],
    neighgrid[2,0,1,1]]
Bgrid[1,2,1] = field[3, neighgrid[0,1,2,1], neighgrid[1,1,2,1],
    neighgrid[2,1,2,1]]
Bgrid[1,0,1] = field[3, neighgrid[0,1,0,1], neighgrid[1,1,0,1],
    neighgrid[2,1,0,1]]
Bgrid[1,1,2] = field[3, neighgrid[0,1,1,2], neighgrid[1,1,1,2],
    neighgrid[2,1,1,2]]
Bgrid[1,1,0] = field[3, neighgrid[0,1,1,0], neighgrid[1,1,1,0],
    neighgrid[2,1,1,0]]
B = Bgrid[1,1,1] #Magnetic field strength at point

    #Find neighbouring theta_B values
thetagrid = np.zeros([3,3,3])
thetagrid[1,1,1] = field[4, neighgrid[0,1,1,1], neighgrid[1,1,1,1],
     neighgrid[2,1,1,1]]
thetagrid[2,1,1] = field[4, neighgrid[0,2,1,1], neighgrid[1,2,1,1],
     neighgrid[2,2,1,1]]
thetagrid[0,1,1] = field[4, neighgrid[0,0,1,1], neighgrid[1,0,1,1],
     neighgrid[2,0,1,1]]
thetagrid[1,2,1] = field[4, neighgrid[0,1,2,1], neighgrid[1,1,2,1],
     neighgrid[2,1,2,1]]
thetagrid[1,0,1] = field[4, neighgrid[0,1,0,1], neighgrid[1,1,0,1],
     neighgrid[2,1,0,1]]
thetagrid[1,1,2] = field[4, neighgrid[0,1,1,2], neighgrid[1,1,1,2],
     neighgrid[2,1,1,2]]
thetagrid[1,1,0] = field[4, neighgrid[0,1,1,0], neighgrid[1,1,1,0],
     neighgrid[2,1,1,0]]
theta = thetagrid[1,1,1] #Value of theta_B at point

    #Find neighbouring phi_B values
phigrid = np.zeros([3,3,3])
phigrid[1,1,1] = field[5, neighgrid[0,1,1,1], neighgrid[1,1,1,1],
    neighgrid[2,1,1,1]]
phigrid[2,1,1] = field[5, neighgrid[0,2,1,1], neighgrid[1,2,1,1],
    neighgrid[2,2,1,1]]
phigrid[0,1,1] = field[5, neighgrid[0,0,1,1], neighgrid[1,0,1,1],
    neighgrid[2,0,1,1]]
phigrid[1,2,1] = field[5, neighgrid[0,1,2,1], neighgrid[1,1,2,1],
    neighgrid[2,1,2,1]]
phigrid[1,0,1] = field[5, neighgrid[0,1,0,1], neighgrid[1,1,0,1],
    neighgrid[2,1,0,1]]
phigrid[1,1,2] = field[5, neighgrid[0,1,1,2], neighgrid[1,1,1,2],
    neighgrid[2,1,1,2]]
phigrid[1,1,0] = field[5, neighgrid[0,1,1,0], neighgrid[1,1,1,0],
    neighgrid[2,1,1,0]]
```

```
        phi = phigrid[1,1,1] #Value of phi_B at point

        #Approximate the derivatives of B
        dBdr = np.zeros(3)
        dBdr[0] = 0.5*(Bgrid[2,1,1]-Bgrid[0,1,1])/stepr
        dBdr[1] = 0.5*(Bgrid[1,2,1]-Bgrid[1,0,1])/stepr
        dBdr[2] = 0.5*(Bgrid[1,1,2]-Bgrid[1,1,0])/stepr

        #Approximate the derivatives of theta_B
        dthdr = np.zeros(3)
        dthdr[0] = 0.5*(thetagrid[2,1,1]-thetagrid[0,1,1])/stepr
        dthdr[1] = 0.5*(thetagrid[1,2,1]-thetagrid[1,0,1])/stepr
        dthdr[2] = 0.5*(thetagrid[1,1,2]-thetagrid[1,1,0])/stepr

        #Approximate the derivatives of phi_B
        dphdr = np.zeros(3)
        dphdr[0] = 0.5*(phigrid[2,1,1]-phigrid[0,1,1])/stepr
        dphdr[1] = 0.5*(phigrid[1,2,1]-phigrid[1,0,1])/stepr
        dphdr[2] = 0.5*(phigrid[1,1,2]-phigrid[1,1,0])/stepr

        #Assign matrix elements
        sin = np.sin(theta) #Calculate the sine
        cos = np.cos(theta) #Calculate the cosine
        omega = (- dBdr*sin + 1j*B*dphdr*sin - B*dthdr*cos)*np.exp(-1j*phi)
            /np.sqrt(2) #Calculates offdiagonals
        diff_hamx[0,0], diff_hamy[0,0], diff_hamz[0,0] = (dBdr*cos - B*
            dthdr*sin)
        diff_hamx[0,1], diff_hamy[0,1], diff_hamz[0,1] = omega
        diff_hamx[1,0], diff_hamy[1,0], diff_hamz[1,0] = np.conjugate(omega
            )
        diff_hamx[1,2], diff_hamy[1,2], diff_hamz[1,2] = omega
        diff_hamx[2,1], diff_hamy[2,1], diff_hamz[2,1] = np.conjugate(omega
            )
        diff_hamx[2,2], diff_hamy[2,2], diff_hamz[2,2] = (-dBdr*cos + B*
            dthdr*sin)

        #Return the matrices with correct prefactors
        diff_hamx = Gamma*hbar*diff_hamx
        diff_hamy = Gamma*hbar*diff_hamy
        diff_hamz = Gamma*hbar*diff_hamz

        diffhamsave[(diffpar, point)] = diff_hamx, diff_hamy, diff_hamz #
            Save the result

        return diff_hamx, diff_hamy, diff_hamz

    elif(diffpar=="th_r"):

        diff_ham  = np.zeros([3,3], dtype='complex_') #Initialize return
            matrix

        #Assign matrix elements
        sin = np.sin(2*point[3]) #Calculate the sine of twice theta_r
        cos = np.cos(2*point[3]) #Calculate the cosine of twice theta_r
        exp = np.exp(1j*point[4]) #Calculate the complex exponential of
            phi_r
        sq = np.sqrt(2) #Calculate the square root of two used
        omega = sq*exp*cos # Calculate an often occuring element
```

```
        diff_ham[0,0] = -sin
        diff_ham[0,1] = -1*np.conjugate(omega)
        diff_ham[0,2] = sin/exp**2
        diff_ham[1,0] = -1*omega
        diff_ham[1,1] = 2*sin
        diff_ham[1,2] = np.conjugate(omega)
        diff_ham[2,0] = exp**2*sin
        diff_ham[2,1] = omega
        diff_ham[2,2] = -sin

        #Return the matrix with correct prefactors
        diff_ham = J*hbar*diff_ham

        diffhamsave[(diffpar, point)] = diff_ham #Save the result

        return diff_ham

    elif(diffpar=="ph_r"):

        diff_ham  = np.zeros([3,3], dtype='complex_') #Initialize return
            matrix

        #Assign matrix elements
        sin = np.sin(2*point[3]) #Calculate the sine of twice theta_r
        sin2 = np.sin(point[3])**2 #Calculate the square of sine of theta_r
        exp = np.exp(1j*point[4]) #Calculate the complex exponential of
            phi_r
        sq = np.sqrt(2) #Calculate the square root of two used
        omega = 1j*exp*sin/sq # Calculate an often occuring element

        diff_ham[0,1] = -np.conjugate(omega)
        diff_ham[0,2] = -2j*sin2/exp**2
        diff_ham[1,0] = -omega
        diff_ham[1,2] = np.conjugate(omega)
        diff_ham[2,0] = 2j*sin2*exp**2
        diff_ham[2,1] = omega

        #Return the matrix with correct prefactors
        diff_ham = J*hbar*diff_ham

        diffhamsave[(diffpar, point)] = diff_ham #Save the result

        return diff_ham

    else:
        raise ValueError("Invalid string for differentiation coordinate
            passed to diffhamiltonian")


##Solves the eigenvalue problem of the fast Hamiltonian at point point for
    field field.
##The point is taken to be of shape (x, y, z, theta_r, phi_r).
##Returns the energies and eigenvectors in pairs with ascending energies.
    The vectors are
##normalized column vectors in the singlet-triplet basis.
def eigensolver(point, field):
```

```
    warnings.filterwarnings("error")

    #First calculate the fast Hamiltonian at point
    ham = np.zeros([3,3], dtype='complex_') #Initialize empty matrix
    pointx = int(point[0]) #Get point index (this is needed for dtype
        purposes)
    pointy = int(point[1])
    pointz = int(point[2])

    #Find the values of B, theta_B and phi_B at point
    B = field[3, pointx, pointy, pointz]
    thetaB = field[4, pointx, pointy, pointz]
    phiB = field[5, pointx, pointy, pointz]
    xi = J/(Gamma*B)
    if xi == np.inf or np.isnan(xi):
        print(f'External field is zero at point {point}! Please correct the
            field or the streams')

    #Extract theta_r and phi_r at point
    thetar = point[3]
    phir = point[4]

    #Assign matrix elements
    cosr = np.cos(thetar) #Calculate the cosine of theta_r
    sin2r = np.sin(2*thetar) #Calculate the sine of twice theta_r
    cos2r = np.cos(2*thetar) #Calculate the cosine of twice theta_r
    sinrsq = np.sin(thetar)**2 #Calculate the square of sine of theta_r
    cosrsq = np.cos(thetar)**2 #Calculate the square of cosine of theta_r
    cosB = np.cos(thetaB) #Calculate the cosine of theta_B
    sinB = np.sin(thetaB) #Calculate the sine of theta_B
    expr = np.exp(1j*phir) #Calculate the complex exponential of phi_r
    expB = np.exp(1j*phiB) #Calculate the complex exponential of phi_B
    sq = np.sqrt(2) #Calculate the square root of two often used

    ham[0,0] = xi*cosrsq + cosB
    ham[0,1] = -sinB / (expB*sq) - xi*sin2r/(expr*sq)
    ham[0,2] = xi*sinrsq/expr**2
    ham[1,0] = -expB*sinB/sq - xi*expr*sin2r/sq
    ham[1,1] = -xi*cos2r
    ham[1,2] = -sinB/(expB*sq) + xi*sin2r/(expr*sq)
    ham[2,0] = xi*expr**2*sinrsq
    ham[2,1] = -expB*sinB/sq + xi*expr*sin2r/sq
    ham[2,2] = xi*cosrsq-cosB

    #Fix matrix prefactors
    ham = Gamma*B*hbar*ham

    #Calculate eigenvalues and eigenvectors
    eigenvalues, eigenvectors = scipy.linalg.eigh(ham)

    #Return the result
    return eigenvalues, eigenvectors

##Calculates the synthetic scalar field at point point for field field for
    state number n.
##The point is taken to be of shape (x, y, z, theta_r, phi_r).
##Returns a tuple of the scalar field value followed by the fast energy.
scalarsave = {}
```

```
def scalarcalc(point, field, n):

    point = tuple(point)

    #First check if the scalar field has been calculated here before
    global scalarsave
    if (point, n) in scalarsave:
        #print('scalarsave used')
        return scalarsave[(point, n)]

    #Fix point format
    point = tuple(point)

    #First retrieve the energies and eigenstates
    energies, eigvec = eigensolver(point, field)

    #Differentiate the Hamiltonian w.r.t. each coordinate
    dHam = [0,0,0,0,0]
    dHam[0], dHam[1], dHam[2] = diffhamiltonian('r', point, field)
    dHam[3] = diffhamiltonian('th_r', point, field)
    dHam[4] = diffhamiltonian('ph_r', point, field)

    Phi = 0 #Initialize synthetic scalar

    for i in range(5):
        for l in range(3):
            if not l == n: #Remove diagonals
                braket = np.vdot(eigvec[n], np.dot(dHam[i], eigvec[l])) #
                    Braket for formula
                Phi += (hbar**2 /(2*mass[i]) * #Add up contributions to the
                     synthetic scalar
                braket*np.conjugate(braket) / (energies[n] - energies[l])
                    **2).real #Note the discard of the imaginary part,
                    numerical errors otherwise arise
    #print(f'Phi = {Phi}')
    returnlist = (Phi, energies[n])

    scalarsave[(point, n)] = returnlist

    return returnlist

##Calculates the acceleration due to the synthetic magnetic field and
    summarizes all
##acceleration contributions. This is done for the position pos, the
    velocity vel as a tuple
##with the field field for state number n. The position and velocity is
    taken to be of
##shape (x, y, z, theta_r, phi_r). Note that position is here given in m,
    and will be
##fitted to the discrete lattice.
##Returns the velocity (for integration purposes) followed by the
    acceleration of the
##system in m/s^2.
##Note that the t argument is a dummy.
def acc(t, posvel, field, n, norot):
    #Find nr and stepr
    nr = field.shape[1] #Number of points along each axis of the lattice
    stepr = lablength/(nr-1) #Distance between lattice points
```

```
#Extract pos and vel:
pos = posvel[0:5]
vel = posvel[5:10]


#Fit position to a point
point = [0,0,0,0,0]
for i in range(3):
    point[i] = int(round(pos[i]/stepr))
    if point[i] >= nr-2 or point[i] < 2:
        return np.concatenate((vel, np.zeros(5))) ##Sets the
            acceleration to zero if a
                                                  ##point outside the
                                                      grid is sampled
point[3] = pos[3]
point[4] = pos[4]

point = tuple(point)

##Make sure the B-field is nonzero
pointx = int(point[0]) #Get point index (this is needed for dtype
    purposes)
pointy = int(point[1])
pointz = int(point[2])

#Find the values of B, theta_B and phi_B at point
B = field[3, pointx, pointy, pointz]
if B == 0.0:
    print(f'Warning, external field of zero encountered at {pos}')
    return np.zeros(10) #Freeze stream

#Find energies and eigenstates at point
energies, eigvec = eigensolver(point, field)

#Differentiate the Hamiltonian w.r.t. each coordinate
dHam = [0,0,0,0,0]
dHam[0], dHam[1], dHam[2] = diffhamiltonian('r', point, field)
dHam[3] = diffhamiltonian('th_r', point, field)
dHam[4] = diffhamiltonian('ph_r', point, field)

#Calculate the acceleration due to the syn. magnetic field
Fa = np.zeros(5)
for i in range(5):
    for j in range(5):
        if not j == i: #Remove diagonals
            for l in range(3):
                if not l == n: #Remove diagonals
                    if energies[n] == energies[l]:
                        print(f'Degenerate fast eigenvalues {energies[n
                            ]} and {energies[l]}!')
                    Fa[i] += (-2*hbar * vel[j] / (energies[n]-energies[
                        l])**2 *
                            np.imag(np.vdot(eigvec[n], np.dot(dHam[i],
                                eigvec[l])) *
                            np.vdot(eigvec[l], np.dot(dHam[j], eigvec[n
                                ])))))

#print(f'Fa = {Fa}') #For testing
```

```
global Faavg
Faavg = (Faavg + np.linalg.norm(Fa))/2

    #To get the derivatives of the scalar fields find coordinate values
        of all neighbouring sites

##meshgrid to generate neighbours
neighgrid = np.mgrid[-1:2,-1:2,-1:2, -1:2, -1:2].astype('float')
neighgrid[3,:,:,:,:,:] *= steptheta #Fix theta and phi step sizes
neighgrid[4,:,:,:,:,:] *= stepphi
neighgrid = np.array(point)[:,None,None,None, None, None] + neighgrid
#uses broadcasting to duplicate
    #x,y,z into each point on the grid. the result has first dimension
        determining which
    #coordinate is given and the remaining specifying position related
        to the point

    #Find neighbouring scalar field values and eigenstate energies
scalar = np.zeros([3, 3, 3, 3, 3])
energy = np.zeros([3, 3, 3, 3, 3])
scalar[2,1,1,1,1], energy[2,1,1,1,1] = scalarcalc(neighgrid
    [:,2,1,1,1,1], field, n)
scalar[0,1,1,1,1], energy[0,1,1,1,1] = scalarcalc(neighgrid
    [:,0,1,1,1,1], field, n)
scalar[1,2,1,1,1], energy[1,2,1,1,1] = scalarcalc(neighgrid
    [:,1,2,1,1,1], field, n)
scalar[1,0,1,1,1], energy[1,0,1,1,1] = scalarcalc(neighgrid
    [:,1,0,1,1,1], field, n)
scalar[1,1,2,1,1], energy[1,1,2,1,1] = scalarcalc(neighgrid
    [:,1,1,2,1,1], field, n)
scalar[1,1,0,1,1], energy[1,1,0,1,1] = scalarcalc(neighgrid
    [:,1,1,0,1,1], field, n)
scalar[1,1,1,2,1], energy[1,1,1,2,1] = scalarcalc(neighgrid
    [:,1,1,1,2,1], field, n)
scalar[1,1,1,0,1], energy[1,1,1,0,1] = scalarcalc(neighgrid
    [:,1,1,1,0,1], field, n)
scalar[1,1,1,1,2], energy[1,1,1,1,2] = scalarcalc(neighgrid
    [:,1,1,1,1,2], field, n)
scalar[1,1,1,1,0], energy[1,1,1,1,0] = scalarcalc(neighgrid
    [:,1,1,1,1,0], field, n)

##Test the accuracy
#appr = (scalar[2,1,1,1,1] - scalar[0,1,1,1,1])/scalar[2,1,1,1,1]
#print(f'acctest = {appr}')

#Differentiate the scalar fields
dscalar = np.zeros(5)
dscalar[0] = (scalar[2,1,1,1,1] - scalar[0,1,1,1,1])/(2*stepr)
dscalar[1] = (scalar[1,2,1,1,1] - scalar[1,0,1,1,1])/(2*stepr)
dscalar[2] = (scalar[1,1,2,1,1] - scalar[1,1,0,1,1])/(2*stepr)
dscalar[3] = (scalar[1,1,1,2,1] - scalar[1,1,1,0,1])/(2*steptheta)
dscalar[4] = (scalar[1,1,1,1,2] - scalar[1,1,1,1,0])/(2*stepphi)

#print(f'dscalar = {dscalar}')
global dscalaravg
dscalaravg = (dscalaravg + np.linalg.norm(dscalar))/2

#Differentiate the energies
```

```
    denergy = np.zeros(5)
    denergy[0] = (energy[2,1,1,1,1] - energy[0,1,1,1,1])/(2*stepr)
    denergy[1] = (energy[1,2,1,1,1] - energy[1,0,1,1,1])/(2*stepr)
    denergy[2] = (energy[1,1,2,1,1] - energy[1,1,0,1,1])/(2*stepr)
    denergy[3] = (energy[1,1,1,2,1] - energy[1,1,1,0,1])/(2*steptheta)
    denergy[4] = (energy[1,1,1,1,2] - energy[1,1,1,1,0])/(2*stepphi)

    acc = Fa - dscalar - denergy #Summarize forces

    #print(f'denergy = {denergy}')
    global denergyavg
    denergyavg = (denergyavg + np.linalg.norm(denergy))/2

    for i in range(5):
        acc[i] = acc[i]/mass[i] #Divide by mass to get acceleration

    if norot: #Freeze rotational axes if norot is turned on
        vel[3:5] = 0
        acc[3:5] = 0

    if acc[3] > 1e14:
        print(f'The rotation is out of control, acc = {acc}')
    #print(f'Vel, acc = {vel}, {acc}')
    #global tickcount
    #tickcount += 1
    #print(f'Tick! {tickcount}') #For testing purposes
    #print(f'Velocity/acc. is: {vel}, {acc}')
    return np.concatenate((vel, acc))

##Finds the acceleration due to non-synthetic effects, i.e. the fast
    energies only.
def accnosyn(t, posvel, field, n, norot):
    #Find nr and stepr
    nr = field.shape[1] #Number of points along each axis of the lattice
    stepr = lablength/(nr-1) #Distance between lattice points
    #Extract position and velocity:
    pos = posvel[0:5]
    vel = posvel[5:10]


    #Fit position to a point
    point = [0,0,0,0,0]
    for i in range(3):
        point[i] = int(round(pos[i]/stepr))
        if point[i] >= nr-2 or point[i] < 2:
            return np.concatenate((vel, np.zeros(5))) ##Sets the
                acceleration to zero if a
                                                    ##a points outside the
                                                        grid is sampled
    point[3] = pos[3]
    point[4] = pos[4]

        #To get the derivatives of the scalar fields find coordinate values
            of all neighbouring sites

    ##meshgrid to generate neighbours
    neighgrid = np.mgrid[-1:2,-1:2,-1:2, -1:2, -1:2].astype('float')
    neighgrid[3,:,:,:,:] *= steptheta #Fix theta and phi step sizes
```

```
    neighgrid[4,:,:,:,:,:] *= stepphi
    neighgrid = np.array(point)[:,None,None,None, None, None] + neighgrid
    #uses broadcasting to duplicate
        #x,y,z into each point on the grid. the result has first dimension
            determining which
        #coordinate is given and the remaining specifying position related
            to the point

    #Find energies at neighbouring points
    energy = np.zeros([3, 3, 3, 3, 3])
    energy[2,1,1,1,1] = eigensolver(neighgrid[:,2,1,1,1,1], field)[0][n]
    energy[0,1,1,1,1] = eigensolver(neighgrid[:,0,1,1,1,1], field)[0][n]
    energy[1,2,1,1,1] = eigensolver(neighgrid[:,1,2,1,1,1], field)[0][n]
    energy[1,0,1,1,1] = eigensolver(neighgrid[:,1,0,1,1,1], field)[0][n]
    energy[1,1,2,1,1] = eigensolver(neighgrid[:,1,1,2,1,1], field)[0][n]
    energy[1,1,0,1,1] = eigensolver(neighgrid[:,1,1,0,1,1], field)[0][n]
    energy[1,1,1,2,1] = eigensolver(neighgrid[:,1,1,1,2,1], field)[0][n]
    energy[1,1,1,0,1] = eigensolver(neighgrid[:,1,1,1,0,1], field)[0][n]
    energy[1,1,1,1,2] = eigensolver(neighgrid[:,1,1,1,1,2], field)[0][n]
    energy[1,1,1,1,0] = eigensolver(neighgrid[:,1,1,1,1,0], field)[0][n]

    #Differentiate the energy
    denergy = np.zeros(5)
    denergy[0] = (energy[2,1,1,1,1] - energy[0,1,1,1,1])/(2*stepr)
    denergy[1] = (energy[1,2,1,1,1] - energy[1,0,1,1,1])/(2*stepr)
    denergy[2] = (energy[1,1,2,1,1] - energy[1,1,0,1,1])/(2*stepr)
    denergy[3] = (energy[1,1,1,2,1] - energy[1,1,1,0,1])/(2*steptheta)
    denergy[4] = (energy[1,1,1,1,2] - energy[1,1,1,1,0])/(2*stepphi)

    global denergyavg
    denergyavg = (denergyavg + np.linalg.norm(denergy))/2

    acc = - denergy
    for i in range(5):
        acc[i] = acc[i]/mass[i] #Divide by mass to get acceleration

    if norot: #Freeze rotational axes if norot is turned on
        vel[3:5] = 0
        acc[3:5] = 0

    #print("Tick!") #For testing purposes
    return np.concatenate((vel, acc))

##Solves the ODE and returns the solution as per sscipy.integrate.solve_ivp
    .
##The external magnetic field is given as field. The
##dumbbell is placed initially at position pos and with velocity vel of the
    shape (x, y,
##z, theta_r, phi_r). Note that cartesian position here is in m.
##The spin subsystem is assumed to remain in the fast
##eigenstate labeled n. Runs until the time reaches tmax.
##The given initial conditions must be tuples.
def solvedyn(pos, vel, field, n, norot=False, nosyn=False):
    posvel = pos + vel

    #Reset average force counters
    global Faavg
    global dscalaravg
```

```
    global denergyavg
    Faavg = 0
    dscalaravg = 0
    denergyavg = 0

    #Find times to require ODE evaluation
    t_eval = np.linspace(0, tmax, 100000)

    #Set error tolerances
    nr = field.shape[1]
    tolr = lablength/(2*(nr - 1))
    #tolr = lablength/4
    toltheta = steptheta/2
    #toltheta = 1
    tolphi = stepphi/2
    #tolphi = 1
    atol = [tolr, tolr, tolr, toltheta, tolphi, tolr/10, tolr/10, tolr/10,
        toltheta/10,
            tolphi/10]
    if nosyn:
        edgedistance.terminal = True
        sol = solve_ivp(accnosyn, (0,tmax), posvel, events=edgedistance,
            args=(field, n,
            norot), atol=atol, t_eval=t_eval)
    else:
        edgedistance.terminal = True
        sol = solve_ivp(acc, (0,tmax), posvel, events=edgedistance, args=(
            field, n, norot), atol=atol, t_eval=t_eval)

    sol.Faavg = Faavg
    sol.denergyavg = denergyavg
    sol.dscalaravg = dscalaravg

    return sol

##Event to terminate integration, returns distance to the closest edge
    minus a small
##correction to avoid hitting the edge
def edgedistance(t, posvel, field, n, norot):
    pos= posvel[0:3]
    mindist = np.amin(pos)
    maxdist = np.amax(pos)
    distancetoedge = min(mindist, lablength - maxdist)

    #Find nr and stepr
    nr = field.shape[1] #Number of points along each axis of the lattice
    stepr = lablength/(nr-1) #Distance between lattice points

    return distancetoedge - 2*stepr


##Plotting function, takes a list of solutions sol from solve_ivp, a field
    field and displays an
##interactive 3D swarm plot. Uses matplotlib.
def lineplot(sol, field, I, initvel, swarmnum, n, norot, nosyn):

    #For testing print average acceleration components
    for stream in sol:
```

```
        print(f'Faavg = {stream.Faavg}')
        print(f'dscalaravg = {stream.dscalaravg}')
        print(f'denergyavg = {stream.denergyavg}')

    fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
    #fig = plt.figure()
    #ax = fig.axes(projection='3d')
    ax.set_xlim((0, lablength))
    ax.set_ylim((0, lablength))
    ax.set_zlim((0, lablength))
    ax.set_xlabel('x', fontsize=10, color='blue')
    ax.set_ylabel('y', fontsize=10, color='blue')
    ax.set_zlabel('z', fontsize=10, color='blue')

    #Find nr and stepr
    nr = field.shape[1] #Number of points along each axis of the lattice
    stepr = lablength/(nr-1) #Distance between lattice points
    #Create grids for the quiver:
    xx, yy, zz = stepr*np.mgrid[0:nr, 0:nr, 0:nr]
    xx = xx[0::int(nr/5), 0::int(nr/5), 0::int(nr/5)]
    yy = yy[0::int(nr/5), 0::int(nr/5), 0::int(nr/5)]
    zz = zz[0::int(nr/5), 0::int(nr/5), 0::int(nr/5)]
    Bx = field[0,:,:,:][0::int(nr/5), 0::int(nr/5), 0::int(nr/5)]
    By = field[1,:,:,:][0::int(nr/5), 0::int(nr/5), 0::int(nr/5)]
    Bz = field[2,:,:,:][0::int(nr/5), 0::int(nr/5), 0::int(nr/5)]

    for stream in sol:
    #Extract pos
        pos = stream.y[0:3,:]
        ax.plot3D(pos[0,:], pos[1,:], pos[2,:], color='red') #Plot the
            integrated path
    #Plot magnetic field for testing purposes
    ##ax.quiver(xx, yy, zz, Bx, By, Bz, length=0.0001, normalize=True)
    #plot = mlab.plot3d(pos[0,:], extent=[0,0,0,lablength,lablength,
        lablength])


    plt.savefig(f'saves/graphs/I{I}nr{nr}lablength{lablength}tmax{tmax}J{J}
        Gamma{Gamma}mass{mass0}len{len0}n{n}vel{initvel}swarmnum{swarmnum}
        norot{norot}nosyn{nosyn}.png',
                bbox_inches='tight')
    plt.show()
```

A.3 synfieldsolver.py

```
import numpy as np
import magfield as mg
import synfieldtools as sn
import pickle
from numpy import pi as pi

if __name__ == '__main__':
    #Set field and modes
    availablefieldtypes = ['simplewire', 'oppositecoils']
    fieldtype = 'oppositecoils'
    I = 100 #Current parameter for the field
    norot = False #Whether to ignore rotational degrees of freedom
    nosyn = True #Whether to ignore synthetic fields
    overwriteresult = False #Whether to overwrite previous ODE results
```

```
#Define parameters

nr = 101 #Number of points in field lattice
lablength = 1e-3 #Cube side of lab in m
tmax = 0.1 #Trajectory time in s
J = 1e2 #Spin-spin coupling strength
Gamma = 1e9 #Spin-field coupling strength

mass0 = 3.58e-25 #The total mass of the dumbbell in kg, as a
    placeholder this is the mass of
                #two silver atoms
len0 = 5e-5 #The distance between dumbbell edges in m
mass = np.repeat(mass0, 5) #Full mass vector
mass[3] = mass0*len0**2/4
mass[4] = mass[3]

#Set parameters
sn.nr = nr
sn.lablength = lablength
sn.tmax = tmax
sn.J = J
sn.Gamma = Gamma
sn.mass0 = mass0
sn.len0 = len0
sn.mass = mass

#Set initial position, velocity and the fast eigenstate to consider
step = lablength/nr
initposarray = np.array((10*step, 10*step, 10*step, 0, 0))
swarmnum = 4 #Square root of number of streams in swarm
swarmgrid = np.mgrid[0:lablength-20*step:swarmnum*1j,0:lablength-20*
    step:swarmnum*1j] #Grid to swarm the initial positions
swarmgrid = np.insert(swarmgrid, 2, np.zeros(swarmgrid.shape[1:3]),
    axis=0)
swarmgrid = np.insert(swarmgrid, 2, np.zeros(swarmgrid.shape[1:3]),
    axis=0)
swarmgrid = np.insert(swarmgrid, 0, np.zeros(swarmgrid.shape[1:3]),
    axis=0)
initposarray = initposarray[:,None,None] + swarmgrid
initvel = (1e-2, 0, 0, 0, 0)
eigenstate = 2

try: #Try to load pregenerated result
    with open(f'saves/odesols/resultF{fieldtype}I{I}nr{nr}lablength{
        lablength}tmax{tmax}J{J}Gamma{Gamma}mass{mass0}len{len0}n{
        eigenstate}vel{initvel}swarmnum{swarmnum}norot{norot}nosyn{nosyn
        }.bin', 'rb') as file:
         sol = pickle.load(file)
except FileNotFoundError:
    overwriteresult = True

#Generate a field
if fieldtype == 'simplewire':
    field = mg.simplewire(nr, lablength, I)
if fieldtype == 'oppositecoils':
    field = mg.oppositecoils(nr, lablength, I, overwrite=False)
```

```
    if overwriteresult: #Only perform calculations if result not avaiable
        on save:

        #Integrate paths
        print('Integrating dynamics . . . . Please wait')
        sol = []
        for i in range(initposarray.shape[1]):
            for j in range(initposarray.shape[2]):
                print(tuple(initposarray[:,i,j]))
                try:
                    sol.append(sn.solvedyn(tuple(initposarray[:,i,j]),
                        initvel, field, eigenstate, norot, nosyn))
                except Exception:
                    print(f'A stream has failed to generate, most probably
                        due to crossing the centre')

        #Save the result
        print(f'Saving result to resultF{fieldtype}I{I}nr{nr}lablength{
            lablength}tmax{tmax}J{J}Gamma{Gamma}mass{mass0}len{len0}n{
            eigenstate}vel{initvel}norot{norot}nosyn{nosyn}.bin')
        with open(f'saves/odesols/resultF{fieldtype}I{I}nr{nr}lablength{
            lablength}tmax{tmax}J{J}Gamma{Gamma}mass{mass0}len{len0}n{
            eigenstate}vel{initvel}swarmnum{swarmnum}norot{norot}nosyn{nosyn
            }.bin', 'wb') as file:
            pickle.dump(sol, file)

    else:
        print('Loading previously generated result')
    #Extract positions and orientations
    pos = sol[0].y[0:3,:]
    vel = sol[0].y[5:8,:]
    ori = sol[0].y[3:5,:]
    #Print the result
    print('Times sampled:')
    print(sol[0].t)
#     print('Path integrated:')
#     print(pos)
    print('Rotation integrated:')
    print(ori)
#     print('Velocity integrated:')
#     print(vel)



    sn.lineplot(sol, field, I, initvel, swarmnum, eigenstate, norot, nosyn)
```