Time-Dependent Schrödinger Equation with Magnetic Field

Etienne Corminboeuf

27th June 2020

Contents

1 Introduction

In this report we consider a spinless particle in $\mathbb{R}^d$ with mass $m \in \mathbb{R}_{\geq 0}$ and charge $e \in \mathbb{R}$ in a homogeneous magnetic field $B(t)$. We follow the notation introduced in the paper by Gradinaru and Rietmann and quickly recap the important elements. For a full derivation please consult [1].

1.1 Mathematic model

In quantum mechanics, the time evolution of a particle subject to a magnetic field is governed by the Pauli equation

$$i\hbar\partial_t \Psi(x,t) = H_P(t)\Psi(x,t) \tag{1}$$

$$H_P(t) := \frac{1}{2m}\sum_{k=1}^{d}(p_k - eA_k(x,t))^2 + e\phi(x,t) + \tilde{V}(x,t) \tag{2}$$

where $\tilde{V}(x,t)$ is some external potential. The magnetic field 2-form dA associated with $B(t)$ is independent of x because of the homogeneity of $B(t)$ and we can thus rewrite the magnetic vector potential to

$$A(x,t) := \frac{1}{2}B_{jk}(t)x^j \mathrm{d}x^k, \tag{3}$$

where $B(t) = (B_{jk}(t))_{j,k=1}^{d}$ is a real, skew-symmetric matrix. Using the operators

$$L_{jk} := x_j p_k - x_k p_j \tag{4}$$

$$H_B(t) := -\sum_{j,k=1}^{d} B_{jk}(t) L_{jk} \tag{5}$$

the Pauli-Hamiltonian takes the form

$$H_P(t) = \frac{1}{2m}\left(\hbar^2(-\Delta) - e\sum_{1\leqslant j<k\leqslant d} B_{jk}(t)L_{jk} + \frac{e^2}{4}\|B(t)x\|_{\mathbb{R}^d}^2\right) + e\phi(x,t) + \tilde{V}(x,t). \tag{6}$$

1.2 Numerical model

We introduce $\epsilon^2 := \hbar$ and redefine t, x and B to find the simplified form

$$H_P(t) = -\Delta + H_B(t) + V(x,t) \tag{7}$$

where $V(x,t) := \frac{1}{2m}\frac{e^2}{4}\|B(t)x\|_{\mathbb{R}^d}^2 + e\phi(x,t) + \tilde{V}(x,t)$ can be considered to be a total effective potential. The Schrödinger equation

$$i\epsilon^2\partial_t\Psi(x,t) = H_P(t)\Psi(x,t) \tag{H}$$

is then split up into three separate parts that can be solved numerically.

$$i\epsilon^2\partial_t\Psi = -\Delta\Psi \tag{K}$$

$$i\epsilon^2\partial_t\Psi = H_B(t)\Psi \tag{M}$$

$$i\epsilon^2\partial_t\Psi = V(x,t)\Psi \tag{P}$$

(K) can be solved discretely in Fourier-space and (P) by pointwise multiplication with $e^{-i/\epsilon^2\int_{t_0}^{t} dtV(x,t)}$. (M) is reduced to the linear differential equation

$$\frac{d}{dt}y(t) = B(t)y(t) \tag{B}$$

[2] proves the existence of a flow map $U(t,t_0)$ which is a solution to (B). The unitary representation

$$\rho : SO(d) \longrightarrow U(L^2(\mathbb{R}^d)) \tag{8}$$

$$R \longmapsto (\rho(R)\Psi)(x) = \Psi(R^{-1}x) \tag{9}$$

maps the solution $U(t,t_0)$ of (B) to a solution of (M). The proof of this statement can be found in [1]. The exact flow map $U(t,t_0)$ is approximated through a Magnus expansion proposed by Blanes and Moan [3]. Direct calculation shows $[-\Delta,\, H_B(t)] = 0$. Thus the flow maps solving (K) and (M) yield a solution to the differential equation

$$i\epsilon^2\partial_t\Psi = (-\Delta + H_B(t))\Psi \tag{K+M}$$

in the following way: Denote the solutions to (K) and (M) by $\Phi_{-\Delta}$ and Φ_{H_B} respectively. This means that the flow map to (K) satisfies

$$i\frac{d}{dt}\Phi_{-\Delta}(t,t_0) = -\Delta\Phi_{-\Delta}(t,t_0), \quad \Phi_{-\Delta}(t_0,t_0) = id, \tag{10}$$

and analogously for the flow map to (M). The flow map $\Phi_{-\Delta+H_B}$ which is a solution to (K+M) then follows as a simple multiplication

$$\Phi_{-\Delta+H_B}(t,t_0) = \Phi_{H_B}(t,t_0)\Phi_{-\Delta}(t,t_0) = \Phi_{-\Delta}(t,t_0)\Phi_{H_B}(t,t_0). \tag{11}$$

Finally, combining the solutions to (K+M) and (P) using a splitting scheme leads to a solution of H.

1.3 Splitting

Consider a splitting scheme with the coefficients (a_i, b_i) for $1 \leq i \leq n$ and the time grids

$$t_i = t_0 + (t - t_0)\sum_{j=1}^{i} b_j \quad \text{and} \quad s_i = t_0 + (t - t_0)\sum_{j=1}^{i} a_j \tag{12}$$

to an initial time t_0. [1] derives an explicit expression of the solution to H as follows:

$$\Phi_H\left(t_0 + Nh, t_0\right) \approx \left(\prod_{j=0}^{N-1}\prod_{i=0}^{n-1} \Phi_{-\Delta}\left(t_{i+1}, t_i\right) \Phi_{\rho(U(t_0+Nh,t_i+jh))V}\left(s_{i+1}+jh, s_i+jh\right)\right) \Phi_{H_B}\left(t_0+Nh, t_0\right) \tag{13}$$

where h is the splitting time step, ρ the representation defined in eq. 8 and $U(t,t_0)$ the exact flow map to (B).

2 Implementation

The method summarised above was implemented into the WaveBlocksND project, developed by Bourquin and Gradinaru [4]. For that a new *FourierMagneticPropagator*-class was created, based on the existing *FourierPropagator*-class. The contents of said method are contained in the *'post_propagate' − function. The full class code can be found in Appendix A. In this section, we present the*

//include convergence plots here
//also include wavefct plot and discuss similarities to paper

3 Summary and Conclusion

Appendices

A Code

```
r"""The WaveBlocks Project

This file contains the Fourier Magnetic Propagator class. The wavefunction
:math:`\Psi` is propagated in time with a splitting of the
exponential :math:`\exp(-\frac{i}{\varepsilon^2} \tau H)`.
```

```
6
7  @author: R. Bourquin
8  @copyright: Copyright (C) 2012, 2016 R. Bourquin
9  @license: Modified BSD License
10 """
11
12 from numpy import array, complexfloating, dot, exp, eye, zeros, shape
13 from numpy.fft import fftn, ifftn
14 from scipy.linalg import expm
15
16 from WaveBlocksND.BlockFactory import BlockFactory
17 from WaveBlocksND.Propagator import Propagator
18 from WaveBlocksND.KineticOperator import KineticOperator
19 from WaveBlocksND.MagneticField import MagneticField
20 from WaveBlocksND.SplittingParameters import SplittingParameters
21
22 __all__ = ["FourierMagneticPropagator"]
23
24
25 class FourierMagneticPropagator(Propagator, SplittingParameters):
26     r"""This class can numerically propagate given initial values :math:`\Psi(x_0, t_0)`
       on
27     a potential hyper surface :math:`V(x)`, in presence of a magnetic field. The
       propagation is done with a splitting
28     of the time propagation operator :math:`\exp(-\frac{i}{\varepsilon^2} \tau H)`.
29     Available splitting schemes are implemented in :py:class:`SplittingParameters`.
30     """
31
32     def __init__(self, parameters, potential, initial_values):
33         r"""Initialize a new :py:class:`FourierMagneticPropagator` instance.
       Precalculate the
34         the kinetic operator :math:`T_e` and the potential operator :math:`V_e`
35         used for time propagation.
36
37         :param parameters: The set of simulation parameters. It must contain at least
38                            the semi-classical parameter :math:`\varepsilon` and the
39                            time step size :math:`\tau`.
40         :param potential: The potential :math:`V(x)` governing the time evolution.
41         :type potential: A :py:class:`MatrixPotential` instance.
42         :param initial_values: The initial values :math:`\Psi(\Gamma, t_0)` given
43                                in the canonical basis.
44         :type initial_values: A :py:class:`WaveFunction` instance.
45
46         :raise: :py:class:`ValueError` If the number of components of :math:`\Psi` does
       not match the
47                            number of energy surfaces :math:`\lambda_i(x)` of the
       potential.
48
49         :raise: :py:class:`ValueError` If the number of components of :math:`\Psi` does
       not match the dimension of the magnetic field :math:`\vec{B}(x)`.
50
51         :raise: :py:class:`ValueError` If the dimensions of the splitting scheme
       parameters :math:`a` and :math:`b` are not equal.
52         """
53         # The embedded 'MatrixPotential' instance representing the potential 'V'.
54         self._potential = potential
55
56         # The initial values of the components '\psi_i' sampled at the given grid.
```

```
        self._psi = initial_values

        if self._potential.get_number_components() != self._psi.get_number_components():
            raise ValueError("Potential dimension and number of components do not match.")

        # The time step size.
        self._dt = parameters["dt"]

        # Final time.
        self._T = parameters["T"]

        # The model parameter '\varepsilon'.
        self._eps = parameters["eps"]

        # Spacial dimension d
        self._dimension = parameters["dimension"]

        # The position space grid nodes '\Gamma'.
        self._grid = initial_values.get_grid()

        # The kinetic operator 'T' defined in momentum space.
        self._KO = KineticOperator(self._grid, self._eps)

        # Exponential '\exp(-i/2*eps^2*dt*T)' used in the Strang splitting.
        # not used
        self._KO.calculate_exponential(-0.5j * self._dt * self._eps**2)
        self._TE = self._KO.evaluate_exponential_at()

        # Exponential '\exp(-i/eps^2*dt*V)' used in the Strang splitting.
        # not used
        self._potential.calculate_exponential(-0.5j * self._dt / self._eps**2)
        VE = self._potential.evaluate_exponential_at(self._grid)
        self._VE = tuple([ve.reshape(self._grid.get_number_nodes()) for ve in VE])

        # The magnetic field
        self._B = MagneticField(parameters["B"])
        # check if magnetic field and potential are of same dimension
        if self._B.get_dimension() != self._dimension:
            raise ValueError("Spacial dimension of potential and magnetic field must be the same")

        #precalculate the splitting needed
        self._a, self._b = self.build(parameters["splitting_method"])
        if shape(self._a) != shape(self._b):
            raise ValueError("Splitting scheme shapes must be the same")

        # Get inital data as function
        packet_descr = parameters["initvals"][0]
        self._initalpacket = BlockFactory().create_wavepacket(packet_descr)


    # TODO: Consider removing this, duplicate
    def get_number_components(self):
        r"""Get the number :math:`N` of components of :math:`\Psi`.

        :return: The number :math:`N`.
        """
```

```
        return self._potential.get_number_components()


    def get_wavefunction(self):
        r"""Get the wavefunction that stores the current data :math:`\Psi(\Gamma)`.

        :return: The :py:class:`WaveFunction` instance.
        """
        return self._psi


    def get_operators(self):
        r"""Get the kinetic and potential operators :math:`T(\Omega)` and :math:`V(\Gamma)`.

        :return: A tuple :math:`(T, V)` containing two ``ndarrays``.
        """
        # TODO: What kind of object exactly do we want to return?
        self._KO.calculate_operator()
        T = self._KO.evaluate_at()
        V = self._potential.evaluate_at(self._grid)
        V = tuple([v.reshape(self._grid.get_number_nodes()) for v in V])
        return (T, V)


    @staticmethod
    def _Magnus_CF4(tspan, B, N, *args):
        r"""Returns the  Fourth Order Magnus integrator :math:`\Omega(A)` according to [#]_.

        :param tspan: Full timespan of expansion.

        :param B: Magnetic field matrix :math:`B(t) = (B_{j,k}(t))_{1 \leq j, k \leq d}`.

        :param N: Number of timesteps for the expansion.

        :param *args: Additional arguments for the magnetic field :math:`B(t, *args)`

        .. [#] S. Blanes and P.C. Moan. "Fourth- and sixth-order commutator-free Magnus integrators for linear and non-linear dynamical systems". Applied Numerical Mathematics, 56(12):1519 - 1537, 2006.
        """
        # Magnus constants
        c1 = 0.5*(1.0 - 0.5773502691896258)
        c2 = 0.5*(1.0 + 0.5773502691896258)
        a1 = 0.5*(0.5 - 0.5773502691896258)
        a2 = 0.5*(0.5 + 0.5773502691896258)

        R = 1.*eye( len( B(1.*tspan[0], *args) ) )
        h = (tspan[1]-tspan[0]) / (1.*N)
        for k in range(N):
            t0 = k*h + tspan[0]
            t1 = t0 + c1*h
            t2 = t0 + c2*h
            B1 = B(t1, *args)
            B2 = B(t2, *args)
            R = dot(expm(a1*h*B1+a2*h*B2), dot(expm(a2*h*B1+a1*h*B2), R))
```

```
166
167         return R
168
169
170     def post_propagate(self, tspan):
171         r"""Given an initial wavepacket :math:`\Psi_0` at time :math:`t=0`, calculate
    the propagated wavepacket :math:`\Psi` at time :math:`tspan \[ 0 \]`. We perform :
    math:`n = \lceil tspan\[ 0 \] /dt \rceil` steps of size :math:`dt`.
172
173         :param tspan: :py class:`ndarray` consisting of end time at position 0, other
    positions are irrelevant.
174         """
175
176         # (ignoriere tspan[0])
177         nsteps = int(tspan[0] / self._dt + 0.5)
178         print("Perform " + str(nsteps) + " steps from t = 0.0 to t = " + str(tspan[0]))
179
180
181         # Magnetfeld Matrix B(t)
182         B = lambda t: self._B(t)
183
184         #how many components does Psi have
185         N = self._psi.get_number_components()
186
187         #start time t_0 = 0?
188         t0 = 0
189         t_a = t0
190         t_b = t0
191
192         #calculate R = U(t0 + N*h, t0)
193         #Use N = n_steps to account for large time difference
194         t_interval = array([t0, tspan[0]])
195         R = FourierMagneticPropagator._Magnus_CF4(t_interval, B, nsteps)
196
197         # rotate the grid by the transpose of R
198         self._grid.rotate(R.T)
199
200         # Compute rotated initial data
201         X = self._grid.get_nodes(flat=True)
202         values = self._initalpacket.evaluate_at(X, prefactor=True)
203         values = tuple([val.reshape(self._grid.get_number_nodes()) for val in values])
204         self._psi.set_values(values)
205
206         self._grid.rotate(R)
207
208         #calculate the necessary timesteps
209         for j in range(nsteps):
210             for i in range(len(self._a)):
211                 # Integral -\int_{tspan[0]}^{tspan[1]}B^2(s)ds und zugehÃ¶rige
    Propagation
212                 # (siehe Paper, Remark 3.1)
213                 minus_B_squared = lambda t: (-1.0) * dot(B(t), B(t))
214                 A = 1.0 / 8.0 * MagneticField.matrix_quad(minus_B_squared, t_a, t_a +
    self._a[i]*self._dt)
215
216                 X = self._grid.get_nodes(flat=True)
217                 VB = sum(X * dot(A, X))
218                 VB = VB.reshape(self._grid.get_number_nodes())
```

```
                prop = exp(-1.0j / self._eps**2 * VB) # ev. -0.5j durch -1j ersetzen...

                values = self._psi.get_values()
                values = [prop * component for component in values]

                self._potential.calculate_exponential(-1.0j *  self._a[i]*self._dt /self
    ._eps**2)

                self._grid.rotate(R.T)
                VE = self._potential.evaluate_exponential_at(self._grid)
                self._VE = tuple([ve.reshape(self._grid.get_number_nodes()) for ve in VE
    ])

                #apply it
                values = [self._VE * component for component in values]
                self._grid.rotate(R)

                t_interval[0] = t_b
                t_interval[1] = t_b + self._b[i]*self._dt

                U = (FourierMagneticPropagator._Magnus_CF4(t_interval, B, 1)).T
                R = dot(R , U)
                if(R.shape != U.shape):
                    raise ValueError("Shapes of R and U do not match")

                #check for obsolete splitting steps
                if(self._b[i] != 0):
                    values = [fftn(component) for component in values]

                    # Apply the kinetic operator
                    self._KO = KineticOperator(self._grid, self._eps)
                    self._KO.calculate_exponential(-0.5j * self._eps**2 * self._b[i]*
    self._dt)

                    TE = self._KO.evaluate_exponential_at()
                    values = [TE * component for component in values]

                    # Go back to real space
                    values = [ifftn(component) for component in values]

                #Apply
                self._psi.set_values(values)

                #update t_a and t_b
                t_a = t_a + self._a[i]*self._dt
                t_b = t_b + self._b[i]*self._dt

        return tspan[0]


    def propagate(self, tspan):
        r"""This method does nothing.
        """
```

References

[1] *V. Gradinaru and O. Rietmann. A High-Order Integrator for the Schrödinger Equation with Time-Dependent, Homogeneous Magnetic Field. Unpublished article, ETH Zürich, submitted for publication, 2020.*

[2] *B. Simon and M. Reed. Fourier Analysis, Self-Adjointness, Volume 2 of Methods of Modern Mathematical Physics. Academic Press, Boston, 1975.*

[3] *S. Blanes and P.C. Moan. Fourth- and sixth-order commutator-free Magnus integrators for linear and non-linear dynamical systems. Applied Numerical Mathematics, 56(12):1519 - 1537, 2006.*

[4] *R. Bourquin and V. Gradinaru. WaveBlocks: Reusable building blocks for simulations with semi-classical wavepackets, GitHub, URL:* `https://github.com/WaveBlocks/WaveBlocksND`*, 2010 - 2016.*