Time-Dependent Schrödinger Equation with Magnetic Field

Etienne Corminboeuf

9th July 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 given by the Pauli equation

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

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

where $V_{ext}(x,t)$ is some external potential, $A_k(x,t)$ the k-th component of the magnetic vector potential $A(x,t)$ and $\phi(x,t)$ the electric potential. Because of the homogeneity of $B(t)$ the magnetic field 2-form dA associated with $B(t)$ is independent of x and we can rewrite the magnetic vector potential to be

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

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

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

1.2 Numerical model

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

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

where $V(x,t) := \frac{1}{2m}\frac{e^2}{4}\|B(t)x\|_{\mathbb{R}^d}^2 + e\phi(x,t) + V_{ext}(x,t)$ 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 $U(t, t_0)$ can be approximated by the 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

$$i\epsilon^2 \frac{\partial}{\partial t}\Phi_{-\Delta}(t, t_0) = -\Delta\Phi_{-\Delta}(t, t_0), \quad \Phi_{-\Delta}(t_0, t_0) = id, \tag{10}$$

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).
To simplify the numerical calculations the propagator for the potential part $\Phi_V = \exp(-i\int_{t_0}^{t} V(x, s)ds)$ can be rewritten using the definition $V(x, t) = \|B(t)x\|^2_{\mathbb{R}^d} + \phi(x, t) + V_{ext}(x, t)$ to

$$\int_{t_0}^{t} V(x, s)ds = \left\langle x, \left(-\int_{t_0}^{t} B^2(s)ds\right) x \right\rangle + \int_{t_0}^{t} \left(\phi(x, s) + V_{ext}(x, s)\right) ds. \tag{14}$$

class WaveBlocksND.**FourierMagneticPropagator**(*parameters*, *potential*, *initial_values*)

This class can numerically propagate given initial values $\Psi(x_0, t_0)$ on a potential hyper[source]
surface $V(x)$, in presence of a magnetic field. The propagation is done with a splitting of the time propagation operator $\exp(-\frac{i}{\varepsilon^2}\tau H)$. Available splitting schemes are implemented in SplittingParameters.

Figure 1: Documentation-Header of the FourierMagneticPropagator Class.

2 Implementation

The considerations made above were implemented into the WaveBlocksND project, developed by Bourquin and Gradinaru [4]. To that avail we created a new *FourierMagneticPropagator*-class, based on the existing *FourierPropagator*-class. The full class code can be found in Appendix A. The *FourierMagneticPropagator*-class carries the header portrayed in figure 1.

In this section, we present the *postpropagate*-function which carries the implementation of section 1 and explain its approach.

The code of the *postpropagate*-function is summarised in Algorithm 1. It is based on Algorithm 1 from [1]. Its header is visible in figure 2.

post_propagate(*tspan*) [source]

Given an initial wavepacket Ψ_0 at time $t = 0$, calculate the propagated wavepacket Ψ at time $tspan[0]$. We perform $n = \lceil tspan[0]/dt \rceil$ steps of size dt.

Parameters: **tspan** – ndarray consisting of end time at position 0, other positions are irrelevant.

Figure 2: Header of the postpropagate-function.

The flow map ρ from equation 8 effectively acts as a rotation. These rotations could not directly be applied onto the potential V or the wavefunction Ψ but had to be realised via a rotation of the underlying grid X. More precisely, the operation $(\rho(R)A)(x) = A(R^{-1}x)$ requires us to first rotate the grid X by R^{-1} and then evaluate quantity A on the rotated grid. Because X is a class member in WaveBlocksND, after the evaluation of A on $R^{-1} \cdot X$ the above rotation needs to be reversed to return to the original state. Otherwise subsequent evaluations would be made on the wrong grid.
Because of the rotational invariance of the first term in equation 14, in the term $\Phi_{\rho(U(t_0+Nh,t_i+jh))V}$ only the electric potential ϕ needs to be subjected to such a rotation.

Algorithm 1: PostPropagate Function

Data: *As arguments to the function:* class instance $self$; end time t.
As arguments to the class instance: step width dt; meshgrid X; initial wave function Ψ_0; potential V; magnetic field B; number of components of the wavefunction N; scaled Planck constant ϵ; splitting method with coefficients $(a_i, b_i)_{i=1}^{n}$; end time of simulation T; dimension d; frequency of writing to disk w_n.

Result: Computes wavepacket at time t and saves it to class member. **Returns:** End time t.

define stepwidth: $n_{steps} = \lceil t/dt \rceil$;
define time grids: $t_a = t_0$, $t_b = t_0$;
calculate flow map $R = U(t_0 + N \cdot dt, t_0)$;
rotate the grid by R^T ;
evaluate initial data on rotated grid and save to wavefunction Ψ;
rotate grid back ;
for *j = 0 to nsteps* **do**
  for *i = 0 to dim(a)* **do**
    potential propagator $\Phi_V = \Phi_B \cdot \Phi_\phi$:
      calculate magnetic field propagator $\Phi_B = \exp(-i\langle x, (\int_{t_a}^{t_a + a_i \cdot dt} B^2(s)ds)x\rangle)$;
      apply propagator to Ψ ;
      rotate grid by R^T ;
      calculate electric and external potential propagator $\Phi_\phi = \exp(-i \int_{t_a}^{t_a + a_i \cdot dt} (\phi(x, s) + V_{ext})ds)$;
      apply propagator to Ψ on rotated grid ;
      rotate grid back ;
    calculate flow map $R = R \cdot U^{-1}(t_b + b_i \cdot dt, t_b)$;
    kinetic propagator $\Phi_{-\Delta}$:
      Fast-Fourier-Transform Ψ to Fourier space ;
      calculate the kinetic propagator $Phi_{-\Delta}$;
      apply propagator to $\mathcal{FFT}(\Psi)$;
      Inverse-FFT Ψ to real space ;
    update time grids: $t_a = t_a + a_i \cdot dt$, $t_b = t_b + b_i \cdot dt$;
  end
end
return t

3 Results

In order to investigate the results of the *FourierMagneticPropagator*, we will present two examples and analyse several important metrics, mainly those of energy and norm conservation and convergence. Additionally, the time evolution is shown in appendix **??**

q	$\begin{pmatrix} 1.0 & 0.0 \end{pmatrix}$
p	$\begin{pmatrix} 0.0 & 0.0 \end{pmatrix}$
Q	$\begin{pmatrix} \sqrt{2.0 \cdot 0.56} & 0.0 \\ 0.0 & \sqrt{2.0 \cdot 0.24} \end{pmatrix}$
P	$\begin{pmatrix} i/\sqrt{2.0 \cdot 0.56} & 0.0 \\ 0.0 & i/\sqrt{2.0 \cdot 0.24} \end{pmatrix}$
S	(0.0)

Table 1: Parameters for the initial wavepacket Ψ_0.

3.1 Example: Threefold Morse Potential

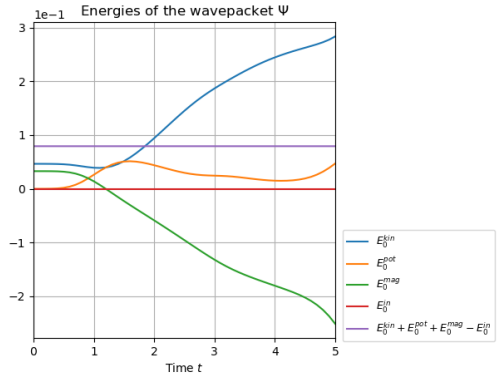
Consider the threefold morse potential for $x \in \mathbb{R}^2$:

$$V_{ext}(x) = 8\left(1 - \exp\left(-\frac{\|x\|^2_{\mathbb{R}^2}}{32}(1 - \cos(3arctan2(x_2, x_1)))^2\right)\right)^2. \tag{15}$$

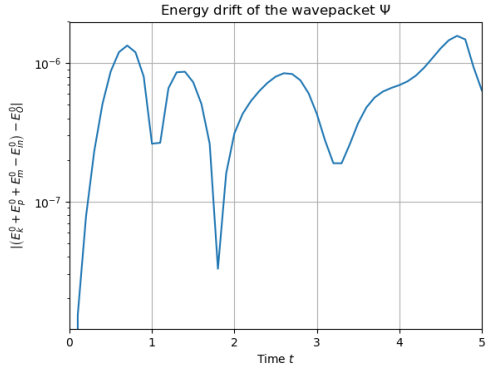
Consider the inital data $\Psi_0 = SOMETHING$ with the parameters from table 1, step width $dt = 0.01$, start time $t_0 = 0$, end time $T = 5$ and the homogeneous, time-independent magnetic field $B = \begin{pmatrix} 0 & -0.5 \\ 0.5 & 0 \end{pmatrix}$. As the splitting method we chose Strang Splitting [6].

3.1.1 Energy and Norm conservation

The evolution of the energies is visible in figure 3, the evolution of the norms in figure 4. We found that both energies and norms are approximately constant.



(a) Energy evolution.



(b) Energy drift evolution.

Figure 3: Energy and energy drift.

3.1.2 Resource Consumption

We ran the simulation on a CPU consisting of 2x AMD Opteron(tm) Processor 6174 with 24 Cores and recorded the resource consumption using Linux' pidstat and time commands. The simulation lasted a total of 5:50:44 (h:min:s) consuming on average 1068% CPU. The usage of RSS, VSZ and CPU over time is visible in figure 5.

3.1.3 Convergence

3.2 Example: Torsional Potential

Consider the torsional potential from [5] for $x \in \mathbb{R}^2$:

$$V_{ext}(x) = \sum_{i=1}^{2} 1 - \cos(x_i). \tag{16}$$

//applications of this potential?
Consider Ψ_0 to be as in the example above with the parameters from table 1 and the splitting to be Strang Splitting.

3.2.1 Energy and Norm conservation

The energy and norm evolution over time is depicted in figures 6 and **??** respectively.

3.2.2 Time Evolution

3.2.3 Resource Consumption

We ran the simulation on a setup identical to the one mentioned above and again recorded the resource consumpion using Linux' time and pidstat commands. The simulation lasted for a total of 7:45:18 (h:min:s) and consumed 247% CPU. The usage of RSS, VSZ and CPU over time is visible in figure 7.

3.2.4 Convergence

4 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)`.

@author: R. Bourquin
```

```
@copyright: Copyright (C) 2012, 2016 R. Bourquin
@license: Modified BSD License
"""

from numpy import array, complexfloating, dot, exp, eye, zeros, shape
from numpy.fft import fftn, ifftn
from scipy.linalg import expm

from WaveBlocksND.BlockFactory import BlockFactory
from WaveBlocksND.Propagator import Propagator
from WaveBlocksND.KineticOperator import KineticOperator
from WaveBlocksND.MagneticField import MagneticField
from WaveBlocksND.SplittingParameters import SplittingParameters

__all__ = ["FourierMagneticPropagator"]


class FourierMagneticPropagator(Propagator, SplittingParameters):
    r"""This class can numerically propagate given initial values :math:`\Psi(x_0, t_0)` on
    a potential hyper surface :math:`V(x)`, in presence of a magnetic field. The propagation is done with a splitting
    of the time propagation operator :math:`\exp(-\frac{i}{\varepsilon^2} \tau H)`.
    Available splitting schemes are implemented in :py:class:`SplittingParameters`.
    """

    def __init__(self, parameters, potential, initial_values):
        r"""Initialize a new :py:class:`FourierMagneticPropagator` instance. Precalculate the
        the kinetic operator :math:`T_e` and the potential operator :math:`V_e`
        used for time propagation.

        :param parameters: The set of simulation parameters. It must contain at least
                           the semi-classical parameter :math:`\varepsilon` and the
                           time step size :math:`\tau`.
        :param potential: The potential :math:`V(x)` governing the time evolution.
        :type potential: A :py:class:`MatrixPotential` instance.
        :param initial_values: The initial values :math:`\Psi(\Gamma, t_0)` given
                               in the canonical basis.
        :type initial_values: A :py:class:`WaveFunction` instance.

        :raise: :py:class:`ValueError` If the number of components of :math:`\Psi` does not match the
                           number of energy surfaces :math:`\lambda_i(x)` of the potential.

        :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)`.

        :raise: :py:class:`ValueError` If the dimensions of the splitting scheme parameters :math:`a` and :math:`b` are not equal.
        """
        # The embedded 'MatrixPotential' instance representing the potential 'V'.
        self._potential = potential

        # The initial values of the components '\psi_i' sampled at the given grid.
        self._psi = initial_values

```

```
59          if self._potential.get_number_components() != self._psi.get_number_components
    ():
60              raise ValueError("Potential dimension and number of components do not
    match.")
61
62          # The time step size.
63          self._dt = parameters["dt"]
64
65          # Final time.
66          self._T = parameters["T"]
67
68          # The model parameter '\varepsilon'.
69          self._eps = parameters["eps"]
70
71          # Spacial dimension d
72          self._dimension = parameters["dimension"]
73
74          # The position space grid nodes '\Gamma'.
75          self._grid = initial_values.get_grid()
76
77          # The kinetic operator 'T' defined in momentum space.
78          self._KO = KineticOperator(self._grid, self._eps)
79
80          # Exponential '\exp(-i/2*eps^2*dt*T)' used in the Strang splitting.
81          # not used
82          self._KO.calculate_exponential(-0.5j * self._dt * self._eps**2)
83          self._TE = self._KO.evaluate_exponential_at()
84
85          # Exponential '\exp(-i/eps^2*dt*V)' used in the Strang splitting.
86          # not used
87          self._potential.calculate_exponential(-0.5j * self._dt / self._eps**2)
88          VE = self._potential.evaluate_exponential_at(self._grid)
89          self._VE = tuple([ve.reshape(self._grid.get_number_nodes()) for ve in VE])
90
91          # The magnetic field
92          self._B = MagneticField(parameters["B"])
93          # check if magnetic field and potential are of same dimension
94          if self._B.get_dimension() != self._dimension:
95              raise ValueError("Spacial dimension of potential and magnetic field must
    be the same")
96
97          #precalculate the splitting needed
98          self._a, self._b = self.build(parameters["splitting_method"])
99          if shape(self._a) != shape(self._b):
100             raise ValueError("Splitting scheme shapes must be the same")
101
102         # Get inital data as function
103         packet_descr = parameters["initvals"][0]
104         self._initalpacket = BlockFactory().create_wavepacket(packet_descr)
105
106
107     # TODO: Consider removing this, duplicate
108     def get_number_components(self):
109         r"""Get the number :math:`N` of components of :math:`\Psi`.
110
111         :return: The number :math:`N`.
112         """
113         return self._potential.get_number_components()
```

```
114
115
116     def get_wavefunction(self):
117         r"""Get the wavefunction that stores the current data :math:`\Psi(\Gamma)`.
118
119         :return: The :py:class:`WaveFunction` instance.
120         """
121         return self._psi
122
123
124     def get_operators(self):
125         r"""Get the kinetic and potential operators :math:`T(\Omega)` and :math:`V(\
    Gamma)`.
126
127         :return: A tuple :math:`(T, V)` containing two ``ndarrays``.
128         """
129         # TODO: What kind of object exactly do we want to return?
130         self._KO.calculate_operator()
131         T = self._KO.evaluate_at()
132         V = self._potential.evaluate_at(self._grid)
133         V = tuple([v.reshape(self._grid.get_number_nodes()) for v in V])
134         return (T, V)
135
136
137     @staticmethod
138     def _Magnus_CF4(tspan, B, N, *args):
139         r"""Returns the  Fourth Order Magnus integrator :math:`\Omega(A)` according to
     [#]_.
140
141         :param tspan: Full timespan of expansion.
142
143         :param B: Magnetic field matrix :math:`B(t) = (B_{j,k}(t))_{1 \leq j, k \leq d
    }`.
144
145         :param N: Number of timesteps for the expansion.
146
147         :param *args: Additional arguments for the magnetic field :math:`B(t, *args)`
148
149         .. [#] 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.
150         """
151         # Magnus constants
152         c1 = 0.5*(1.0 - 0.5773502691896258)
153         c2 = 0.5*(1.0 + 0.5773502691896258)
154         a1 = 0.5*(0.5 - 0.5773502691896258)
155         a2 = 0.5*(0.5 + 0.5773502691896258)
156
157         R = 1.*eye( len( B(1.*tspan[0], *args) ) )
158         h = (tspan[1]-tspan[0]) / (1.*N)
159         for k in range(N):
160             t0 = k*h + tspan[0]
161             t1 = t0 + c1*h
162             t2 = t0 + c2*h
163             B1 = B(t1, *args)
164             B2 = B(t2, *args)
165             R = dot(expm(a1*h*B1+a2*h*B2), dot(expm(a2*h*B1+a1*h*B2), R))
166
```

```
        return R


    def post_propagate(self, tspan):
        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`.

        :param tspan: :py class:`ndarray` consisting of end time at position 0, other
    positions are irrelevant.
        """

        # (ignoriere tspan[0])
        nsteps = int(tspan[0] / self._dt + 0.5)
        print("Perform " + str(nsteps) + " steps from t = 0.0 to t = " + str(tspan[0])
    )


        # Magnetfeld Matrix B(t)
        B = lambda t: self._B(t)

        #how many components does Psi have
        N = self._psi.get_number_components()

        #start time t_0 = 0?
        t0 = 0
        t_a = t0
        t_b = t0

        #calculate R = U(t0 + N*h, t0)
        #Use N = n_steps to account for large time difference
        t_interval = array([t0, tspan[0]])
        R = FourierMagneticPropagator._Magnus_CF4(t_interval, B, nsteps)

        # rotate the grid by the transpose of R
        self._grid.rotate(R.T)

        # Compute rotated initial data
        X = self._grid.get_nodes(flat=True)
        values = self._initalpacket.evaluate_at(X, prefactor=True)
        values = tuple([val.reshape(self._grid.get_number_nodes()) for val in values])
        self._psi.set_values(values)

        self._grid.rotate(R)

        #calculate the necessary timesteps
        for j in range(nsteps):
            for i in range(len(self._a)):
                # Integral -\int_{tspan[0]}^{tspan[1]}B^2(s)ds und zugehÃ¶rige
    Propagation
                # (siehe Paper, Remark 3.1)
                minus_B_squared = lambda t: (-1.0) * dot(B(t), B(t))
                A = 1.0 / 8.0 * MagneticField.matrix_quad(minus_B_squared, t_a, t_a +
    self._a[i]*self._dt)

                X = self._grid.get_nodes(flat=True)
                VB = sum(X * dot(A, X))
                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.
        """
```

B Time evolution

B.1 Threefold Morse Potential

The time evolution of the wavepacket Ψ is portrayed in figure 8. At a position x the color encodes the phase of $\Psi(x)$, the brightness of the pixel encodes the intensity $|\Psi(x)|$.

B.2 Torsional Potential

The time evolution of the wavepacket Ψ is portrayed in figure 9. At a position x the color encodes the phase of $\Psi(x)$, the brightness of the pixel encodes the intensity $|\Psi(x)|$.

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.

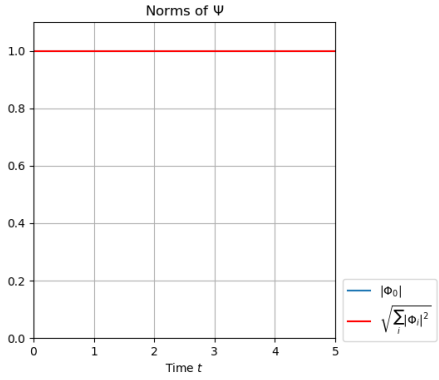
[2] B. Simon and M. Reed. *Fourier Analysis, Self-Adjointness, Volume 2* of *Methods of Modern Mathematical Physics*. Academic Press, Boston, 1975.

[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.

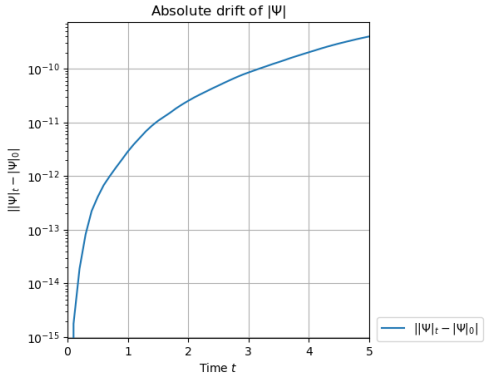
[4] R. Bourquin and V. Gradinaru. *WaveBlocks: Reusable building blocks for simulations with semiclassical wavepackets*, GitHub, URL: `https://github.com/WaveBlocks/WaveBlocksND`, 2010 - 2016.

[5] E. Faou, V. Gradinaru and C. Lubich. *Computing semiclassical quantum dynamics with Hagedorn wavepackets*, SIAM J. Scientific Computing, 31:3027-3041, 01, 2009.

[6] G. Strang. *On the construction and comparison of difference schemes*, SIAM Journal on Numerical Analysis, 5:506-517, 1968.

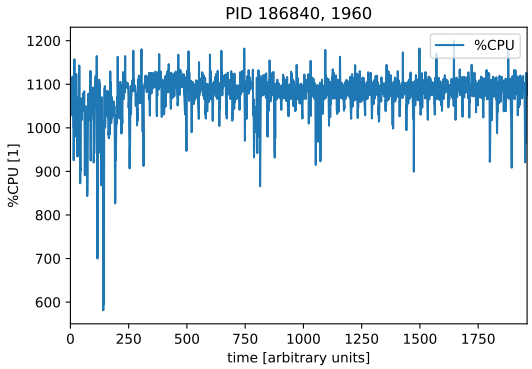


(a) Norm evolution.

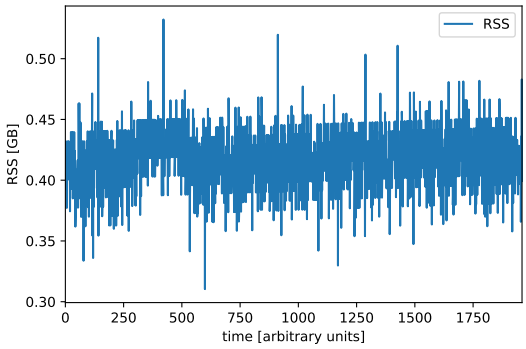


(b) Norm drift evolution.

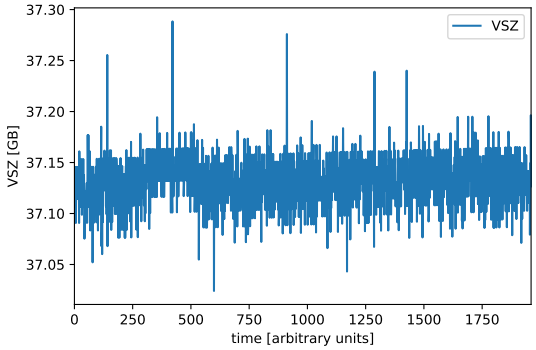
Figure 4: Norm and norm drift.



(a) Percentage of CPU used by the simulation.

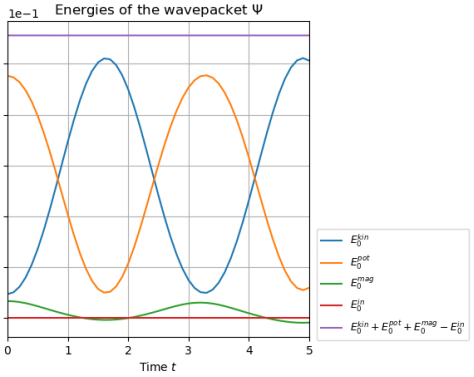


(b) Amount of RSS (Resident Set Size) used by the simulation.

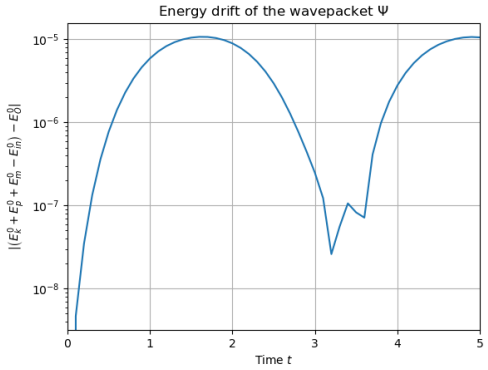


(c) Amount of VSZ (Virtual Memory Size) used by the simulation.

Figure 5: Resource consumption of the simulation with the threefold morse potential.

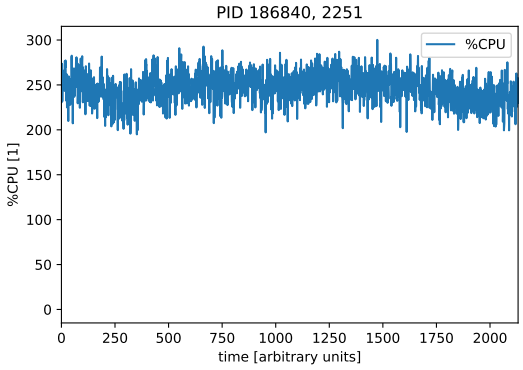


(a) Energy evolution.

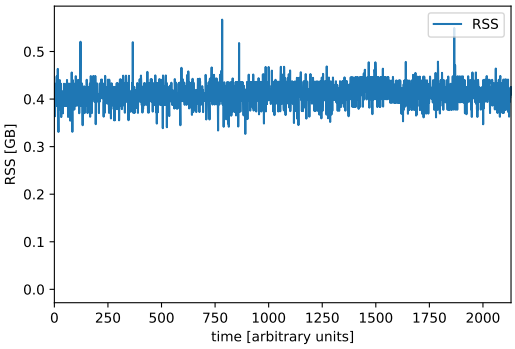


(b) Energy drift evolution.

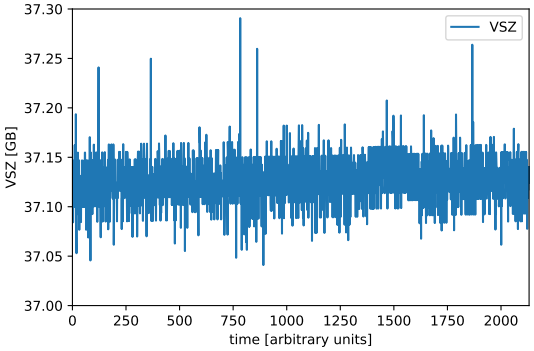
Figure 6: Energy and energy drift.



(a) Percentage of CPU used by the simulation.



(b) Amount of RSS (Resident Set Size) used by the simulation.



(c) Amount of VSZ (Virutal Memory Size) used by the simulation.

Figure 7: Resource consumption of the simulation with the torsional potential. The difference in CPU percentage between the torsional potential and the morse potential is mostly due to different loads on the machines at the time of simulation.

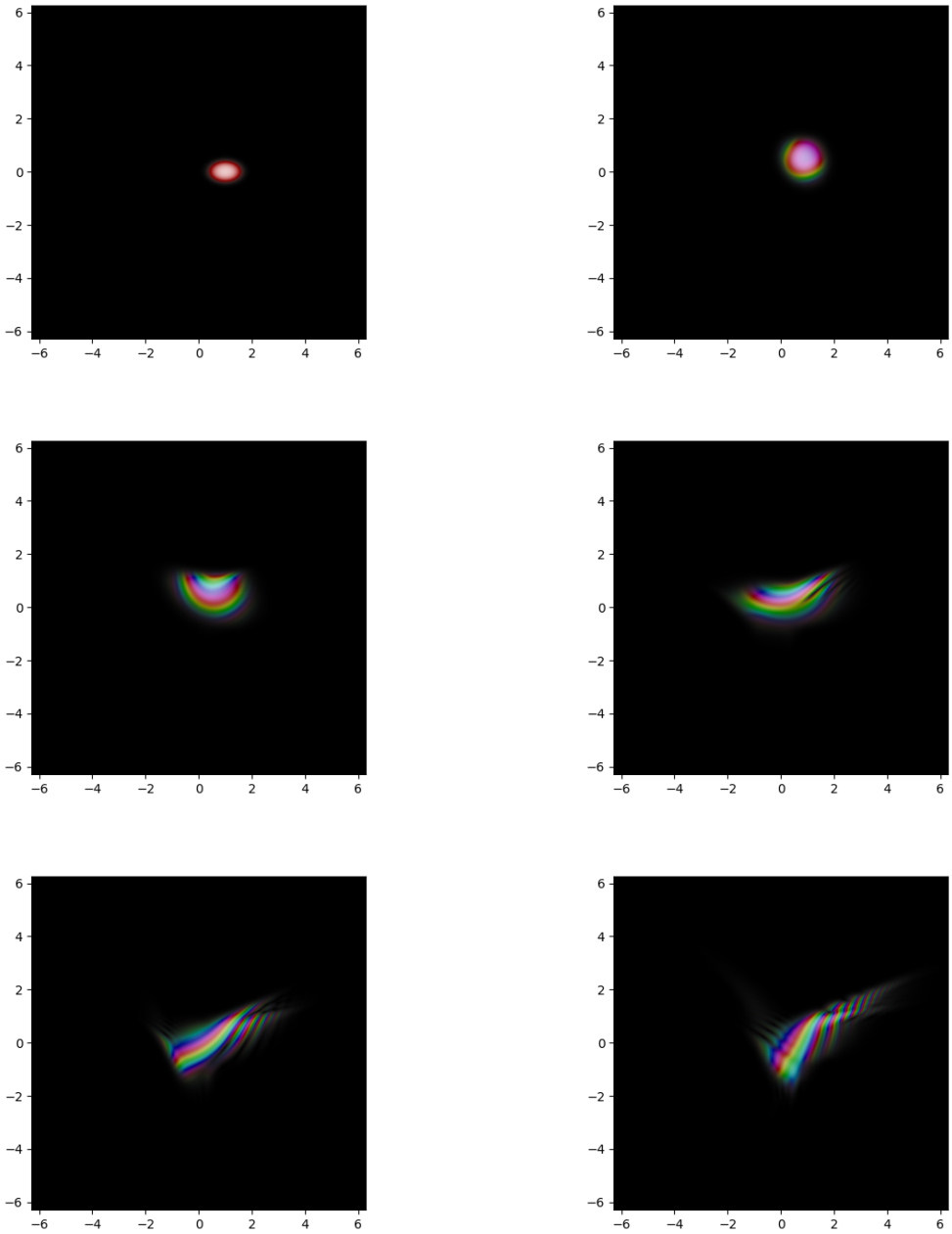


Figure 8: Time evolution of the wavepacket Ψ with initial data Ψ_0 in the threefold morse potential and a homogeneous, time-independent magnetic field.

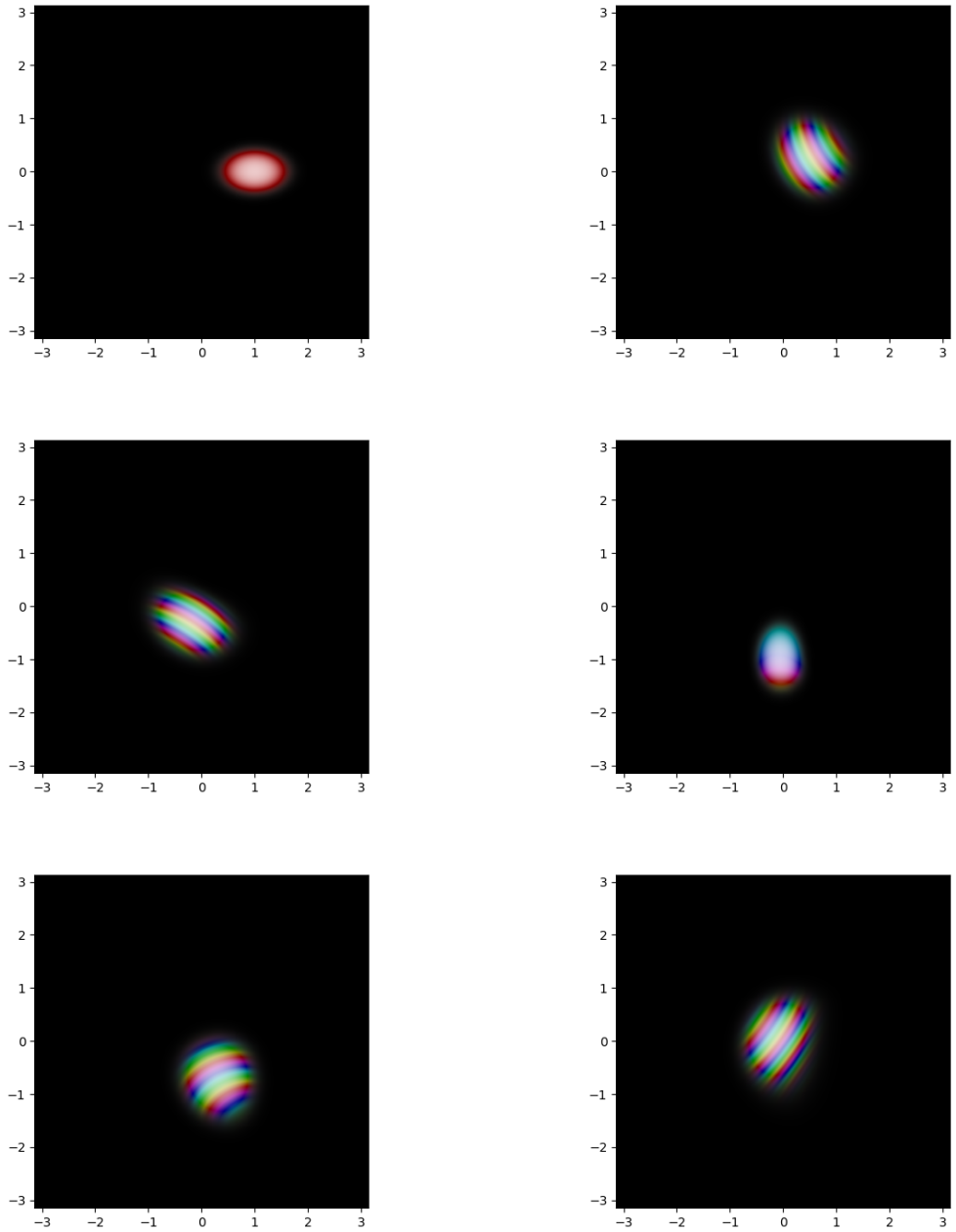


Figure 9: Time evolution of the wavepacket Ψ with initial data Ψ_0 in the torsional potential and a homogeneous, time-independent magnetic field.