A semester thesis written at the
Eidgenössische Technische Hochschule Zürich
on the

Implementation of an Integrator for the Schrödinger Equation with Time-Dependent, Homogeneous Magnetic Field

by Etienne Corminboeuf
under supervision by
Dr. V. Gradinaru and O. Rietmann in
Zurich, 15th August 2020.

Contents

1 Introduction

We consider a spinless particle in $\mathbb{R}^d$ with mass $m > 0$ and charge $e \in \mathbb{R}$ in a homogeneous magnetic field $B(t)$. We follow the notation introduced by Gradinaru and Rietmann in [5] and quickly recap the important elements. For a full derivation please consult [5].

1.1 Mathematical model

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

$$i\hbar\partial_t\Psi(x,t) = H_P(t)\Psi(x,t) \tag{1.1}$$

$$H_P(t) := \frac{1}{2m}\sum_{k=1}^{d}(p_k - eA_k(x,t))^2 + e\phi(x,t) + V_{ext}(x,t) \tag{1.2}$$

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 homogenity of the magnetic field $B(t)$, the 2-form dA associated with $B(t)$ is independent of x and we can rewrite the magnetic vector potential as

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

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

$$L_{jk} := x_j p_k - x_k p_j \tag{1.3}$$

$$H_B(t) := -\sum_{j,k=1}^{d} B_{jk}(t)L_{jk} \tag{1.4}$$

the Pauli-Hamiltonian takes the form

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

1.2 Numerical model

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

$$H_P(t) = -\Delta + H_B(t) + V(x,t)$$

where $V(x,t) := \frac{1}{2m}\frac{e^2}{4}\|B(t)x\|^2_{\mathbb{R}^d} + e\phi(x,t) + V_{ext}(x,t)$ 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 the following three separate parts that can easily be solved numerically:

$$i\epsilon^2 \partial_t \Psi = -\Delta\Psi, \tag{K}$$
$$i\epsilon^2 \partial_t \Psi = H_B(t)\Psi, \tag{M}$$
$$i\epsilon^2 \partial_t \Psi = V(x,t)\Psi. \tag{P}$$

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

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

Consider the flow map $U(t,t_0)$ associated with (B). The unitary representation

$$\rho : SO(d) \longrightarrow U(L^2(\mathbb{R}^d)) \tag{1.5}$$
$$R \longmapsto (\rho(R)\Psi)(x) = \Psi(R^{-1}x) \tag{1.6}$$

maps the solution $U(t,t_0)$ of (B) to a solution of (M). The proof of this statement can be found in [5]. The exact flow map $U(t,t_0)$ can be approximated by the Magnus expansion proposed by Blanes and Moan in [2]. 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. The flow map $\Phi_{-\Delta+H_B}$ which is a solution to (K+M) is

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

A splitting scheme combining the solutions of (K+M) and (P) leads to a numerical 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

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

with an initial time t_0. [5] derives an explicit expression of the solution to (H):

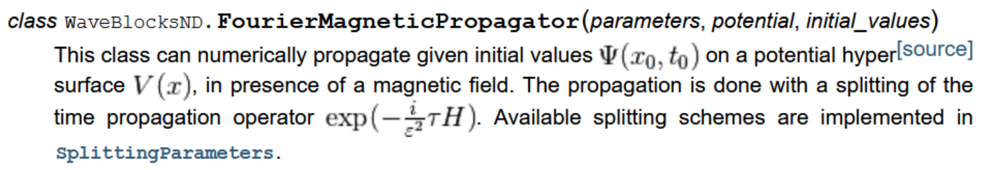
$$\Phi_H(t_0+Nh,t_0) \approx \left(\prod_{j=0}^{N-1}\prod_{i=0}^{n-1}\Phi_{-\Delta}(t_{i+1},t_i)\,\Phi_{\rho(U(t_0+Nh,t_i+jh))V}(s_{i+1}+jh,s_i+jh)\right) \times \Phi_{H_B}(t_0+Nh,t_0) \quad (1.7)$$

where h is the splitting time step, ρ is the representation defined in Equation (1.5) and $U(t,t_0)$ is the exact flow map to (B).
Note also that for $V(x,t) = \|B(t)x\|^2_{\mathbb{R}^d} + \phi(x,t) + V_{ext}(x,t)$ we have:

$$\int_{t_0}^{t} V(x,s)ds = \left\langle x, \left(-\int_{t_0}^{t} B^2(s)ds\right)x\right\rangle + \int_{t_0}^{t}(\phi(x,s)+V_{ext}(x,s))\,ds. \quad (1.8)$$

2 Implementation

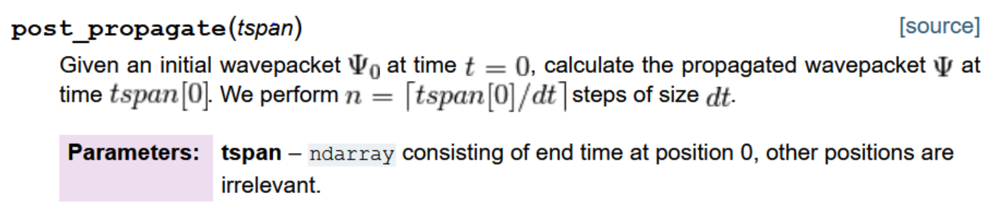
We implemented the method described above into the WaveBlocksND project, developed by Bourquin and Gradinaru [3]. To that avail we created a new *FourierMagneticPropagator*-class, based on the existing *FourierPropagator*-class. The full class code can be found in Appendix A. The *FourierMagneticPropagator*-class carries the header visible below.


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`.

In this section, we present the *postpropagate*-function which carries the implementation of Section 1 and explain its approach.
Note that using the *postpropagate*-function to simulate the time evolution contrasts with the other time propagation algorithms implemented in WaveBlocksND which make use of the *propagate*-function. This difference is due to the structure of our method. WaveBlocksND calls the *propagate*-function for every time step, the *postpropagate*-function on the other hand only when data is actually saved to disk. This happens at specified times, for example after a given time interval $\Delta t = n \cdot dt$. The *FourierMagneticPropagator* does not allow intermediate results to be used to develop the simulation further, rather it requires a complete simulation from start time t_0 to the desired time $t_0 + k\cdot\Delta t$ for some $k \in \mathbb{N}$. The *propagate*-function cannot be adapted to fulfill this requirement, thus the *postpropagate*-function has to be used.
The code of the *postpropagate*-function is summarised in Alg. (1). It is based on the algorithm from [5]. Its header is visible below.

A few additional remarks to Alg. 1: The flow map ρ from Equation (1.5) effectively acts as a rotation. This rotation cannot be applied directly onto the potential V or the



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.

wavefunction Ψ but has 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 in order to return to the original state. Otherwise subsequent evaluations would be conducted on the wrong grid.
Because of the rotational invariance of the first term in Equation (1.8), only the second term $\phi(x, s) + V_{ext}(x, s)$ 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 $\Psi_0$; potential $V$; magnetic field $B$; number of components
      of the wavefunction $N$; $\epsilon$; 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: The wavepacket at time $t$ is saved into the corresponding class member.
        It returns end time $t$.
1  define stepwidth: $n_{steps} = \lceil t/dt \rceil$;
2  define time grids: $t_a = t_0$, $t_b = t_0$ ;
3  calculate flow map $R = U(t_0 + N \cdot dt, t_0)$ ;
4  rotate the grid by $R^T$ ;
5  evaluate initial data on rotated grid and save to wavefunction $\Psi$;
6  rotate grid by $R$ ;
7  for $j$ = 0 to nsteps do
8      for $i$ = 0 to dim(a) do
9          potential propagator $\Phi_V = \Phi_B \cdot \Phi_\phi$ :
10             calculate magnetic field propagator
                $\Phi_B = \exp(-i\langle x, (\int_{t_a}^{t_a+a_i\cdot dt} B^2(s)ds)x\rangle)$ ;
11             apply propagator to $\Psi$ ;
12             rotate grid by $R^T$ ;
13             calculate electric and external potential propagator
                $\Phi_\phi = \exp(-i \int_{t_a}^{t_a+a_i\cdot dt}(\phi(x,s) + V_{ext})ds)$ ;
14             apply propagator to $\Psi$ on rotated grid ;
15             rotate grid back ;
16         update flow map $R = R \cdot U^{-1}(t_b + b_i \cdot dt, t_b)$ ;
17         kinetic propagator $\Phi_{-\Delta}$:
18             Fast-Fourier-Transform $\Psi$ to Fourier space ;
19             calculate the kinetic propagator $\Phi_{-\Delta}$ ;
20             apply propagator to $\mathcal{FFT}(\Psi)$ ;
21             Inverse-FFT $\Psi$ to real space ;
22         update time grids: $t_a = t_a + a_i \cdot dt$, $t_b = t_b + b_i \cdot dt$ ;
23     end
24 end
25 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 conservation, norm conservation and convergence. Additionally, the time evolution is shown in appendix B.

3.1 Example: Threefold Morse Potential

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

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

and the inital data

$$\Psi_0^\epsilon[q, p, Q, P] = \left(\pi\epsilon^2 Q^2\right)^{-\frac{1}{4}} \exp\left(\frac{i}{2\epsilon^2} PQ^{-1}(x-q)^2 + \frac{i}{\epsilon^2} p(x-q)\right)$$

with the parameters from Table 1. The initial data is visible in Figure 3.1.

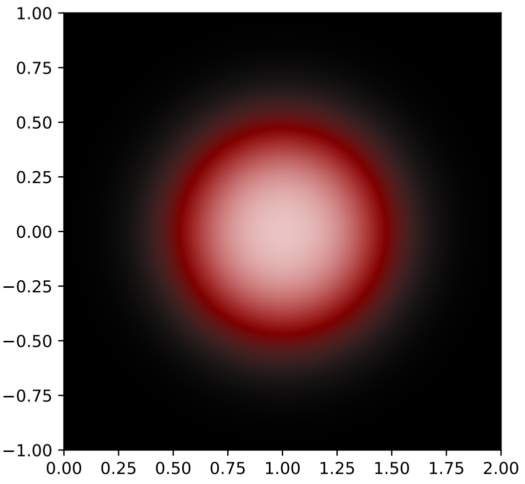


Figure 3.1: Initial Data Ψ_0 with the parameters from Table 1. At a position x, the color of the corresponding pixel encodes the phase of $\Psi_0(x)$, the brightness encodes the intensity $|\Psi_0(x)|$.

Note that this corresponds to a wavefunction concentrated in position around q and in momentum around p with uncertainties $\epsilon|Q|/\sqrt{2}$ and $\epsilon|P|/\sqrt{2}$. Additionally consider the 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 splitting method we chose Strang Splitting [6]. We set $\epsilon = 0.25$. The simulation data is sampled on a spatial grid with

1024×1024 nodes that cover the simulation domain, implemented as a meshgrid from python's numpy package.

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	$\begin{pmatrix} i/\sqrt{2.0 \cdot 0.56} & 0.0 \\ 0.0 & i/\sqrt{2.0 \cdot 0.24} \end{pmatrix}$

Table 1: Parameters for the initial wavepacket Ψ_0.

3.1.1 Energy and Norm conservation

The evolution of the energies is visible in Figure 3.2, the evolution of the norms in Figure 3.3. We see that both energies and norms are approximately constant.

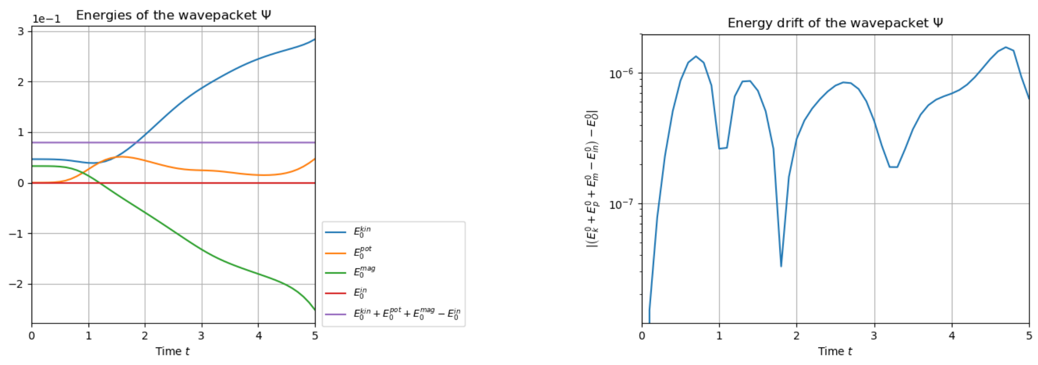


Figure 3.2: Energy and energy drift, $\epsilon = 0.25$.

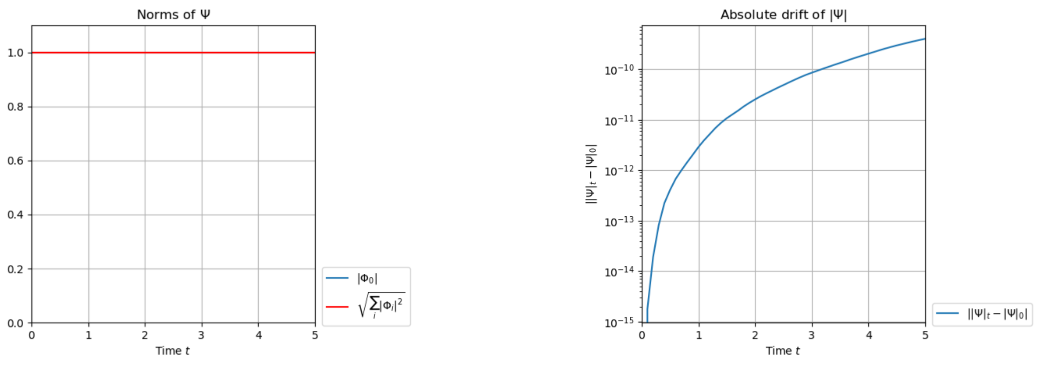
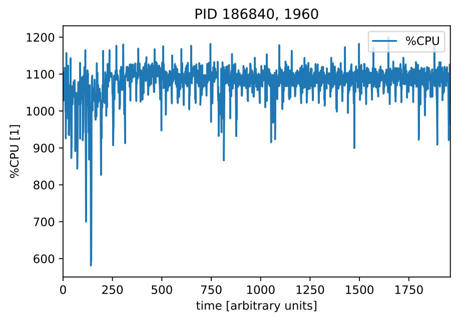


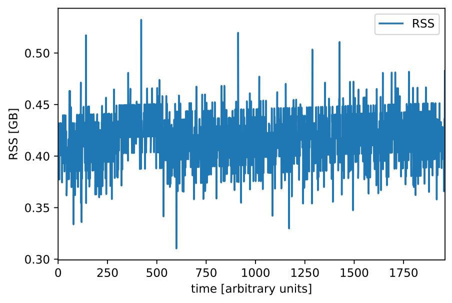
Figure 3.3: Norm and norm drift, $\epsilon = 0.25$.

3.1.2 Resource Consumption

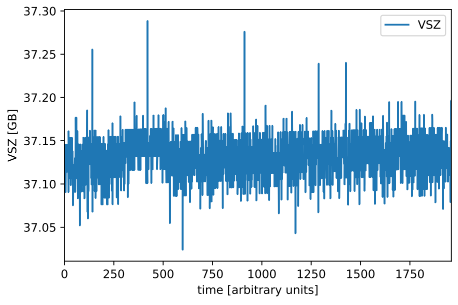
We ran the simulation on a CPU consisting of 2x AMD Opteron(tm) Processor 6174 with 24 Cores and recorded the resource consumption using Linux' pidstat and time



(a) Percentage of CPU.



(b) RSS (Resident Set Size).



(c) VSZ (Virtual Memory Size).

Figure 3.4: Resource consumption of the simulation with the threefold morse potential.

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 3.4.

3.1.3 Convergence

To determine performance for different ϵ, we ran a reference simulation using a splitting of order 6 proposed by Blanes and Moan [1] and a sample simulation using Strang splitting [6], which is of order 2. The difference of those simulations for several values of ϵ is portrayed in Figure 3.5.

3.2 Example: Torsional Potential

Consider the torsional potential from [4] for $x \in \mathbb{R}^2$:

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

and let Ψ_0 be as in the example above with the parameters from Table 1, $\epsilon = 0.25$ and the splitting Strang Splitting again.

3.2.1 Energy and Norm conservation

The energy and norm evolution over time is depicted in Figures 3.6 and 3.7.

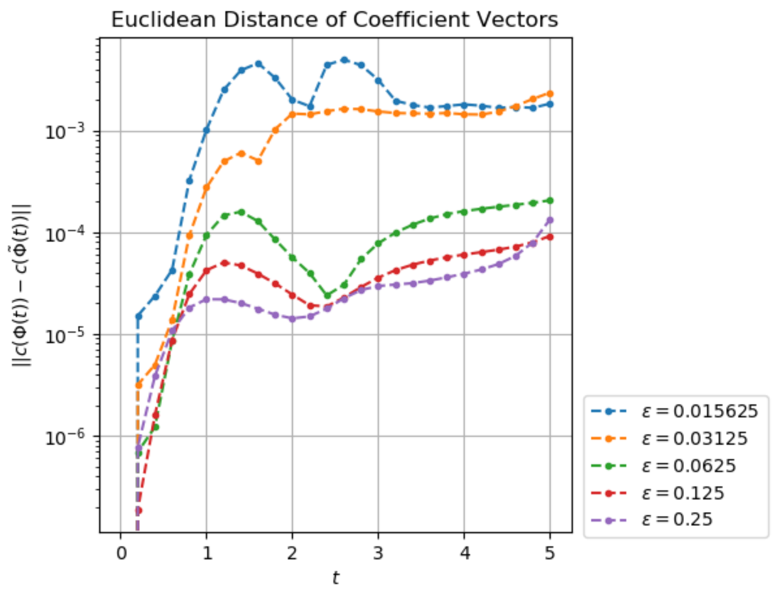


Figure 3.5: Error of the coefficient vectors for different ϵ. Threefold Morse Potential.

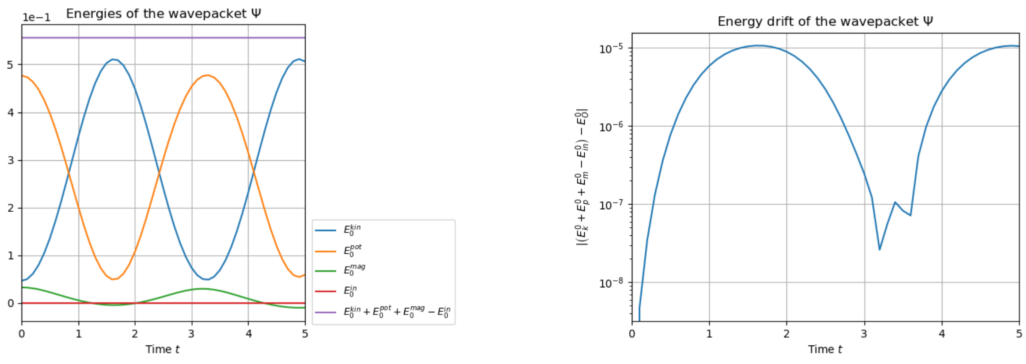


Figure 3.6: Energy and energy drift, $\epsilon = 0.25$.

3.2.2 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 3.8.

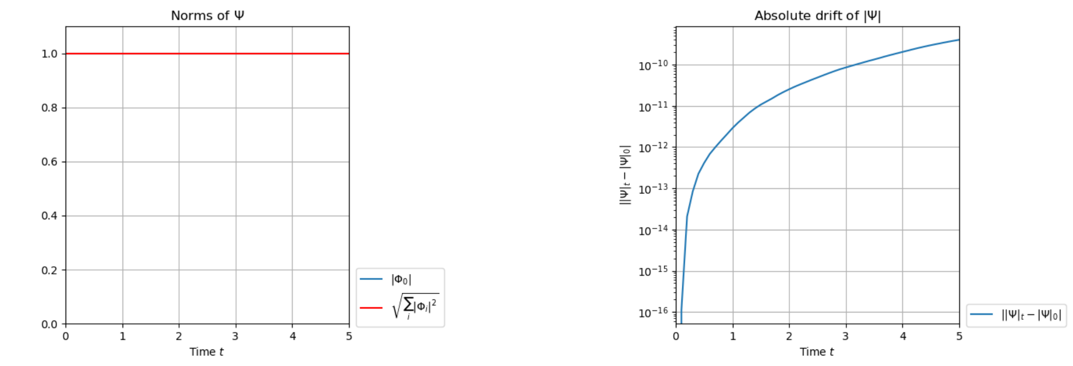
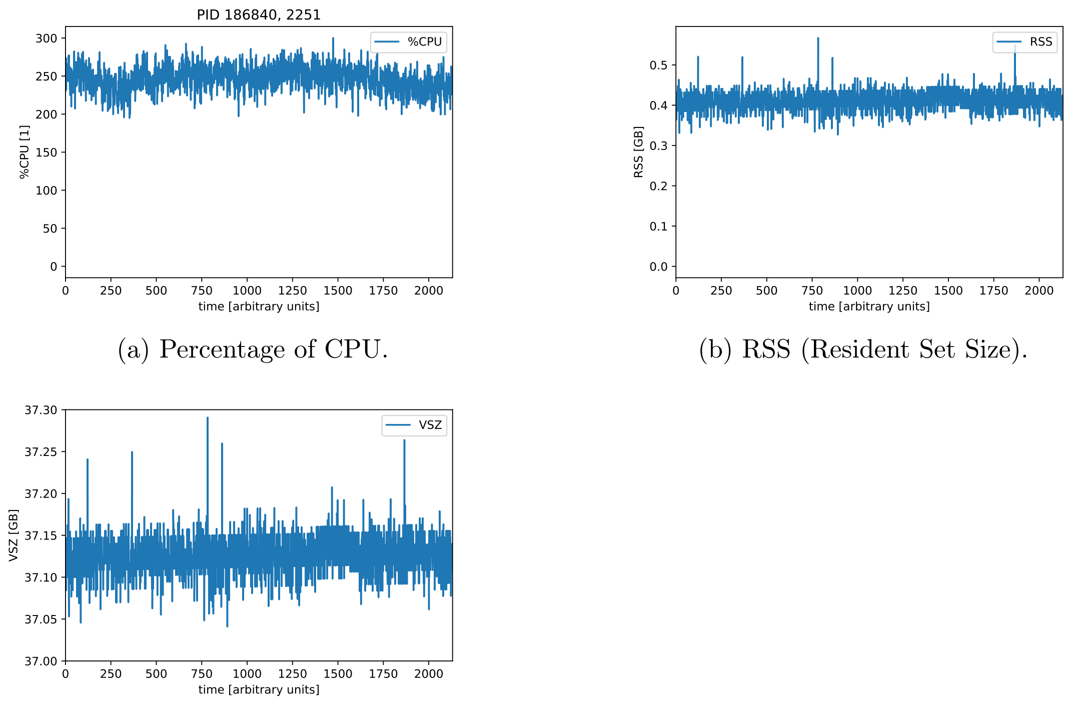


Figure 3.7: Norm and norm drift, $\epsilon = 0.25$.



(a) Percentage of CPU.

(b) RSS (Resident Set Size).

(c) VSZ (Virtual Memory Size).

Figure 3.8: 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.

3.2.3 Convergence

The same considerations as in Section 3.1.3 were made for the torsional potential. The difference of those simulations for various values of ϵ is portrayed in Figure 3.9.

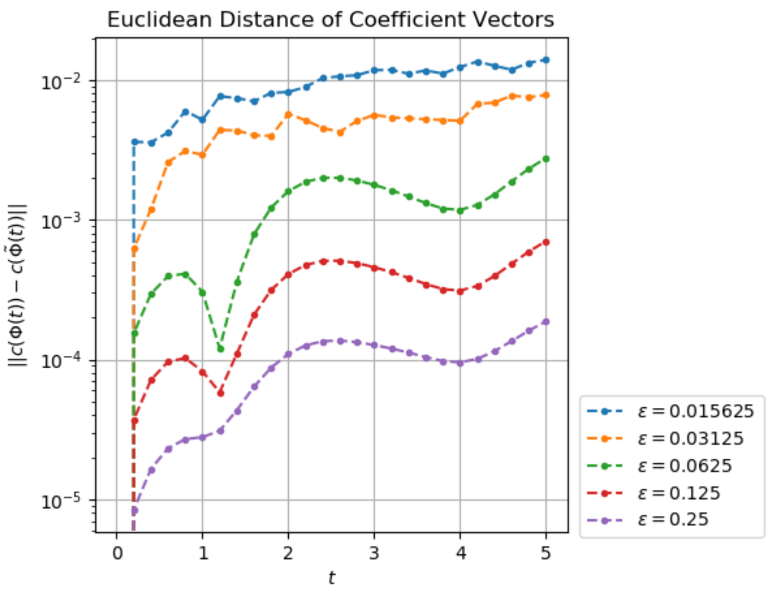


Figure 3.9: Error of the coefficient vectors for different ϵ. Torsional Potential.

4 Discussion and Conclusion

We implemented the method developed by Rietmann and Gradinaru in [5] into the WaveblocksND framework. We found that energies and norms are conserved well in both of the cases we explored. From Figures 3.5 and 3.9 we can conclude that the method is well suited for values of $\epsilon > 0.125$, but is subject to increasing errors for values of ϵ smaller than that.
The major disadvantage of this method is its speed. The section above shows that simulations are time- and resource intensive. Additionally the method is structured such that time-steps are not consecutive but have to be calculated individually. This means that in order to write data to the disk after the 10th and 20th time step (for example to calculate the energy at those times) the simulation needs to run the first 10 time-steps, write to disk, restart at t_0 and run the first 20 time-steps. This doesn't present itself as a problem when only the state at a certain end-time T is needed but quickly becomes costly if a high resolution of the evolution itself is required.

One could further investigate the performance of this method for a particle in a time-dependent magnetic field as well as in dimensions $d > 2$. Additionally performance could be compared to the semi-classical wavepacket approach from [4] also implemented in WaveBlocksND.

Appendices

A Code

```python
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

        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.
```

```python
        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))

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

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


        # Define magnetic field matrix B(t)
        B = lambda t: self._B(t)

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

        #set start time and time grids
        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)

        # rotate grid back to original orientation
        self._grid.rotate(R)

        #calculate timesteps
        # each j is an individual time steps
        for j in range(nsteps):
            # each i is an intermediate time step in the splitting
            for i in range(len(self._a)):

```

```python
            ### Calculate potential flow map \Phi_V = \Phi_B * \Phi_{\phi}###

            # Integral -\int_{tspan[0]}^{tspan[1]}B^2(s)ds and its associated propagation
            # does not need to be rotated, as B^2(s) is independent of rotations
            # (see [Gradinaru and Rietmann, 2020]. 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())
            #define the magnetic field propagator \Phi_B = \int{ -i/eps^2 * \int{ B^2}}
            prop = exp(-1.0j / self._eps**2 * VB)

            #apply the propagator
            values = self._psi.get_values()
            values = [prop * component for component in values]

            #define the propagator \Phi_{\phi} (for electric and spatial potential)
            # these potential are not independent of rotations and need to be rotated
            self._potential.calculate_exponential(-1.0j *  self._a[i]*self._dt /self._eps**2)

            # rotate and evaluate potentials
            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 the propagator
            values = [self._VE * component for component in values]
            self._grid.rotate(R)

            # define time step for Magnus Integrator
            t_interval[0] = t_b
            t_interval[1] = t_b + self._b[i]*self._dt

            # calculate Magnus Integrator
            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):
                    ### calculate kinetic flow map \Phi_{-\Delta} ###

                    #go to fourier space
                    values = [fftn(component) for component in values]

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

                    #calculate and apply the kinetic propagator
                    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]

                # save data
                self._psi.set_values(values)

                #update time grids
                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 B.1. 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 B.2. At a position x the color encodes the phase of $\Psi(x)$, the brightness of the pixel encodes the intensity $|\Psi(x)|$.

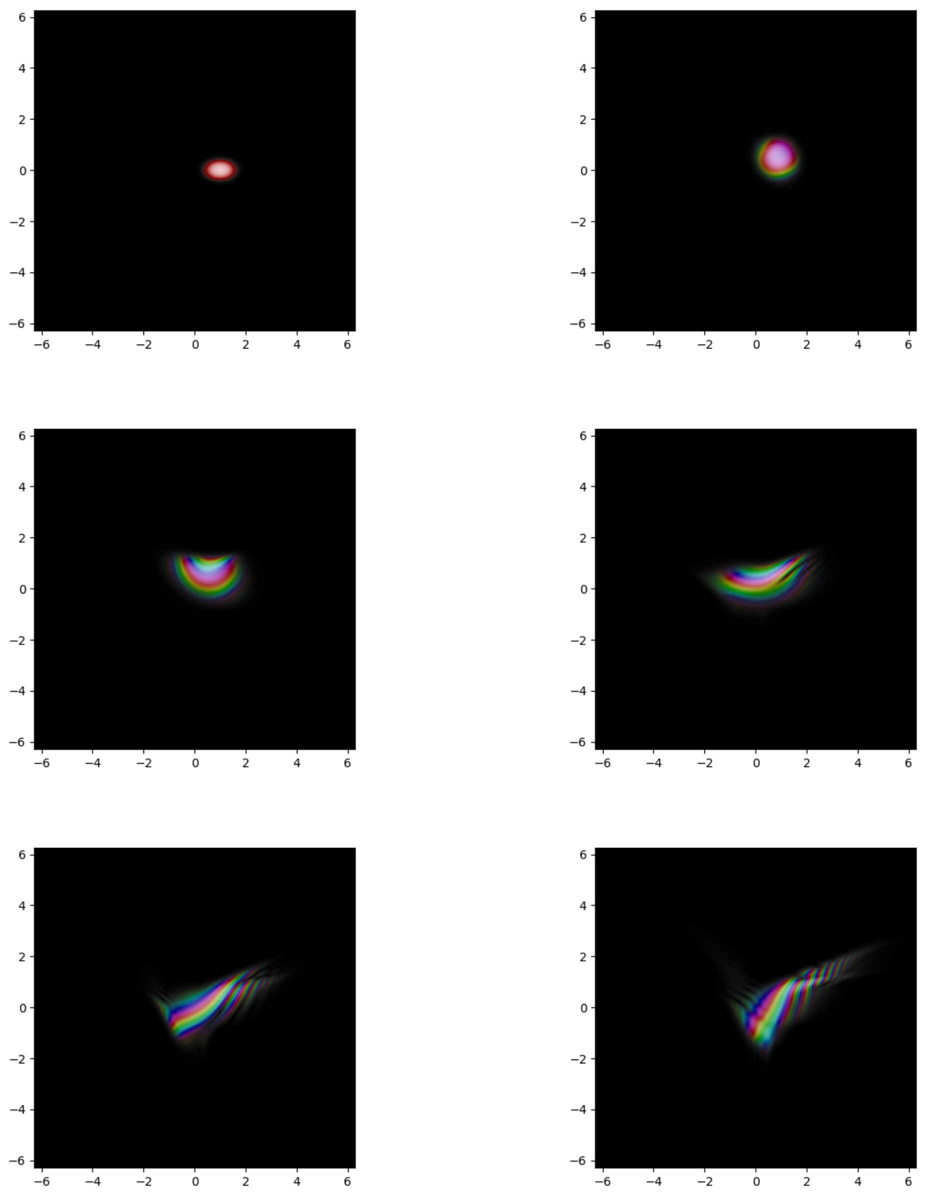


Figure B.1: Time evolution of the wavepacket Ψ with initial data Ψ_0 in the threefold morse potential and a homogeneous, time-independent magnetic field.

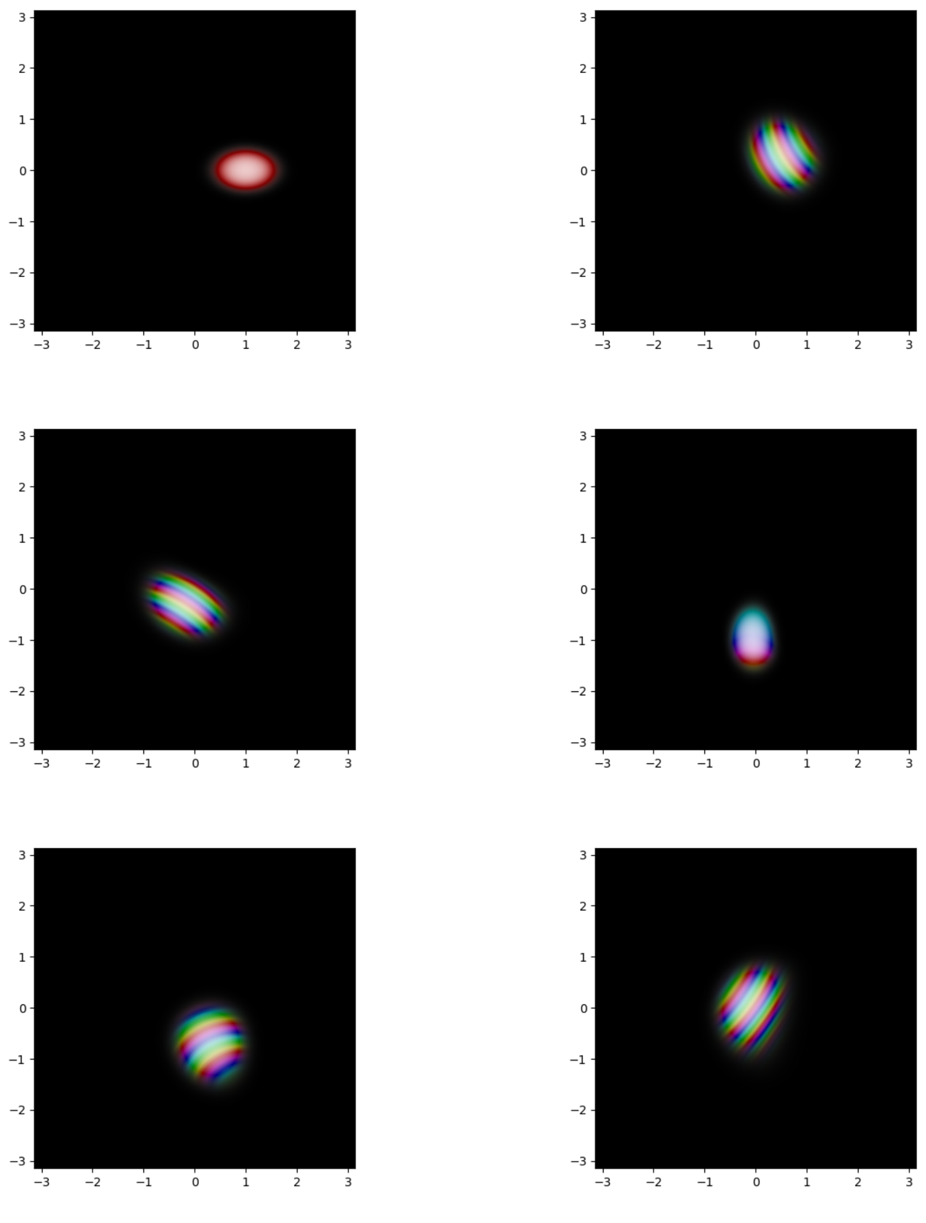


Figure B.2: Time evolution of the wavepacket Ψ with initial data Ψ_0 in the torsional potential and a homogeneous, time-independent magnetic field.

References

[1] S. Blanes and P.C. Moan. Practical symplectic partitioned Runge-Kutta and Runge-Kutta-Nyström methods. *Journal of Computational and Applied Mathematics*, 142(2):313 – 330, 2002.

[2] Sergio Blanes and Per Christian Moan. Fourth-and sixth-order commutator-free Magnus integrators for linear and non-linear dynamical systems. 2006.

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

[4] Erwan Faou, Vasile Gradinaru, and Christian Lubich. Computing Semiclassical Quantum Dynamics with Hagedorn Wavepackets. *SIAM J. Scientific Computing*, 31:3027–3041, 01 2009.

[5] Vasile Gradinaru and Oliver Rietmann. A High-Order Integrator for the Schrödinger Equation with Time-Dependent, Homogeneous Magnetic Field. Technical Report 2018-47, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2018.

[6] Gilbert Strang. On the construction and comparison of difference schemes, 1968.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

IMPLEMENTATION OF AN INTEGRATOR FOR THE SCHRÖDINGER EQUATION WITH TIME-DEPENDENT, HOMOGENEOUS MAGNETIC FIELD

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
CORMINBOEUF	ETIENNE

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
ZÜRICH, 14.08.2020	E. Corminboeuf

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.