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

# Implementation of a Method solving the Time-Dependent Schrödinger Equation with Magnetic Field

by **Etienne Corminboeuf**
under supervision by
Dr. V. Gradinaru and O. Rietmann in
Zurich, July 2020.

# Contents

# 1 Introduction

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 by Gradinaru and Rietmann in [4] and quickly recap the important elements. For a full derivation please consult [4].

## 1.1 Mathematic model

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

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

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

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 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

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

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

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

## 1.2 Numerical model

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

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

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 three separate parts that can be solved numerically.

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

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

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

[5] proves the existence of a flow map $U(t,t_0)$ which is a solution to eq. (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 eq. (B) to a solution of eq. (M). The proof of this statement can be found in [4]. The exact flow map $U(t,t_0)$ can be approximated by the Magnus expansion proposed by Blanes and Moan [1]. Direct calculation shows $[-\Delta, H_B(t)] = 0$. Thus the flow maps solving eq. (K) and eq. (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 eq. (K) and eq. (M) by $\Phi_{-\Delta}$ and $\Phi_{H_B}$ respectively. The flow map $\Phi_{-\Delta+H_B}$ which is a solution to eq. (K+M) then follows as a simple multiplication

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

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

### 1.3 Splitting

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

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

to an initial time $t_0$. [4] derives an explicit expression of the solution to eq. (H) as follows:

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

where $h$ is the splitting time step, $\rho$ the representation defined in eq. (1.5) and $U(t,t_0)$ the exact flow map to eq. (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

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

## 2 Implementation

We implemented the method described above into the WaveBlocksND project, developed by Bourquin and Gradinaru [2]. 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 fig. 2.1.

*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 2.1: Header of the FourierMagneticPropagator Class.

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 Alg. (1). It is based on the algorithm from [4]. Its header is visible in fig. 2.2.

**post_propagate**(*tspan*) [source]
Given an initial wavepacket $\Psi_0$ at time $t=0$, calculate the propagated wavepacket $\Psi$ 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 2.2: Header of the postpropagate-function.

The flow map $\rho$ from eq. (1.5) effectively acts as a rotation. These rotations could not directly be applied onto the potential $V$ or the wavefunction $\Psi$ 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 conducted on the wrong grid.
Because of the rotational invariance of the first term in eq. (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 $N$; scaled Planck constant $\epsilon$; 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:** 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 $\Psi$;
rotate grid by $R$ ;
**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 $\Psi$ ;
            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 $\Psi$ 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 $\Psi$ to Fourier space ;
            calculate the kinetic propagator $\Phi_{-\Delta}$ ;
            apply propagator to $\mathcal{FFT}(\Psi)$ ;
            Inverse-FFT $\Psi$ 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 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 $x \in \mathbb{R}^2$:

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

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. 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 the splitting method we chose Strang Splitting [6].

| | |
|---|---|
| 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 | $(0.0)$ |

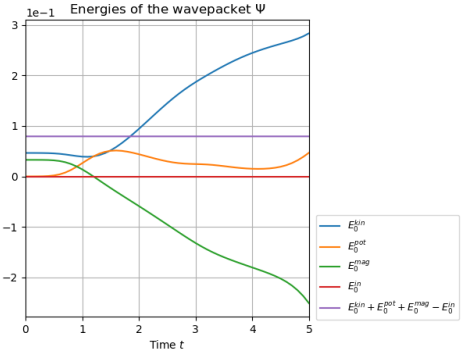
Table 1: Parameters for the initial wavepacket $\Psi_0$.

### 3.1.1 Energy and Norm conservation

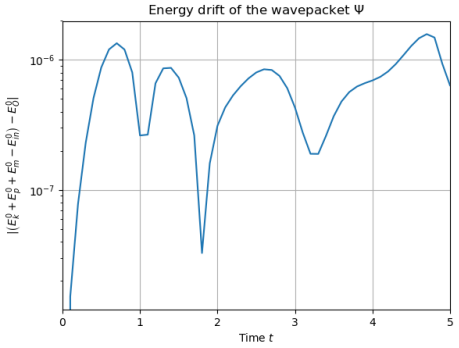
The evolution of the energies is visible in fig. 3.1, the evolution of the norms in fig. 3.2. We found that both energies and norms are approximately constant.

### 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

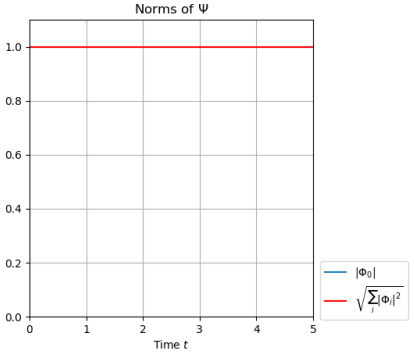



(a) Energy evolution.

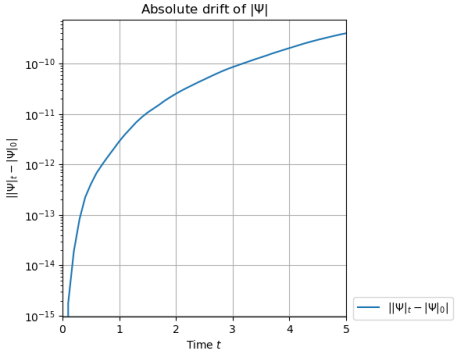



(b) Energy drift evolution.

Figure 3.1: Energy and energy drift.




(a) Norm evolution.




(b) Norm drift evolution.

Figure 3.2: Norm and norm drift.

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 fig. 3.3.

### 3.1.3 Convergence

## 3.2 Example: Torsional Potential

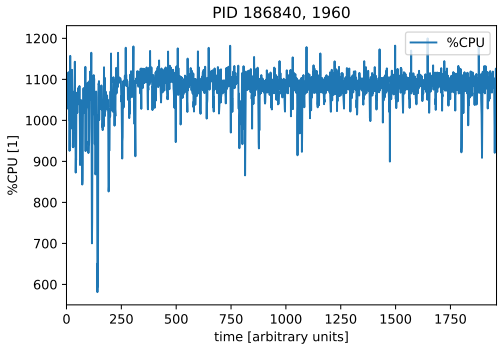
Consider the torsional potential from [3] for $x \in \mathbb{R}^2$:

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

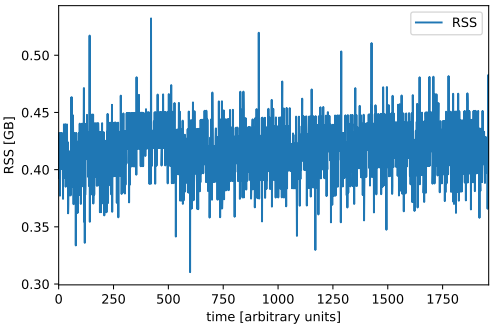
Consider $\Psi_0$ to be as in the example above with the parameters from table 1 and the splitting to be Strang Splitting [6].

### 3.2.1 Energy and Norm conservation

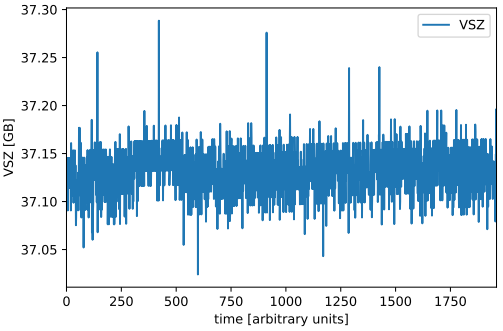
The energy and norm evolution over time is depicted in fig. 3.4 and fig. 3.5 respectively.




(a) Percentage of CPU.




(b) RSS (Resident Set Size).




(c) VSZ (Virtual Memory Size).

Figure 3.3: Resource consumption of the simulation with the threefold morse potential.

### 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 fig. 3.6.

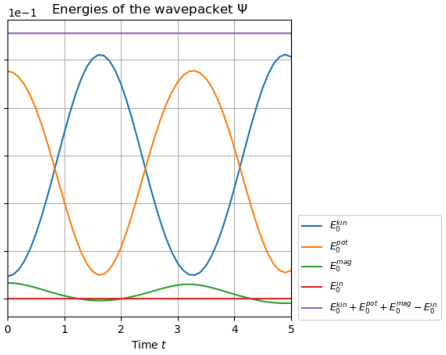
### 3.2.3 Convergence

# 4 Summary and Conclusion

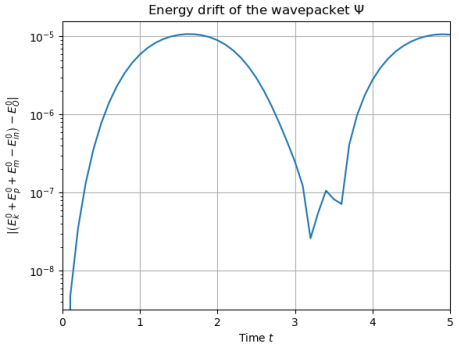
# Appendices

# A Code

```
r"""The WaveBlocks Project

This file contains the Fourier Magnetic Propagator class. The
    wavefunction
```

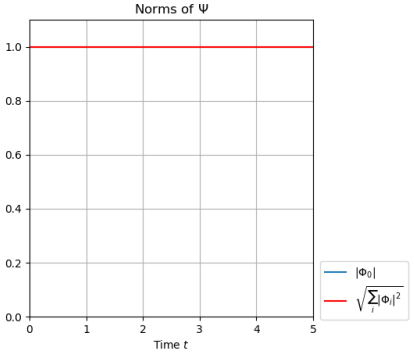



(a) Energy evolution.

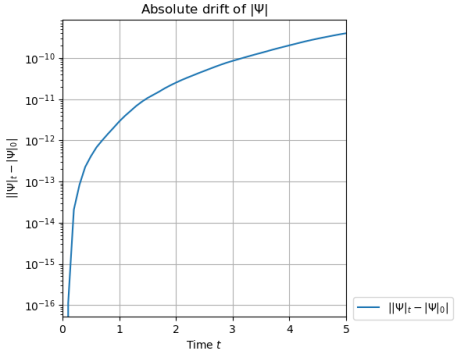



(b) Energy drift evolution.

Figure 3.4: Energy and energy drift.



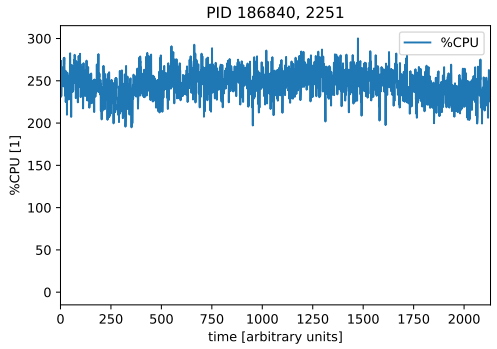

(a) Norm evolution.



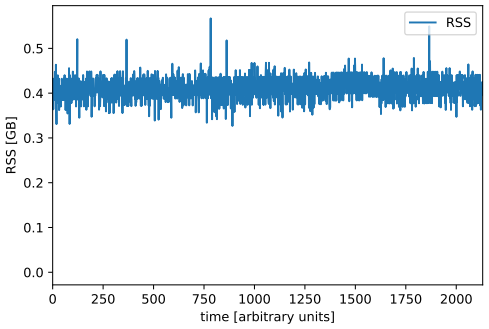

(b) Norm drift evolution.

Figure 3.5: Norm and norm drift.

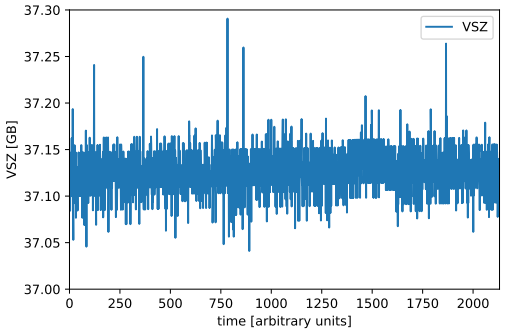
```
:math:‘\Psi‘ is propagated in time with a splitting of the
exponential :math:‘\exp(-\frac{i}{\varepsilon^2} \tau H)‘.

@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"]


```




(a) Percentage of CPU.




(b) RSS (Resident Set Size).




(c) VSZ (Virtual Memory Size).

Figure 3.6: 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.

```
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.
57          self._psi = initial_values
58
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 
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: ``ndarray`` consisting of end time at position 0,
    other positions are irrelevant.
174         """
175
176         #Define stepwidth
177         nsteps = int(tspan[0] / self._dt + 0.5)
178         #Console output
179         print("Perform " + str(nsteps) + " steps from t = 0.0 to t = " +
    str(tspan[0]))
180
181
182         # Define magnetic field matrix B(t)
183         B = lambda t: self._B(t)
184
185         #how many components does Psi have
186         N = self._psi.get_number_components()
187
188         #set start time and time grids
189         t0 = 0
190         t_a = t0
191         t_b = t0
192
193         #calculate R = U(t0 + N*h, t0)
194         #Use N = n_steps to account for large time difference
195         t_interval = array([t0, tspan[0]])
196         R = FourierMagneticPropagator._Magnus_CF4(t_interval, B, nsteps)
197
198         # rotate the grid by the transpose of R
199         self._grid.rotate(R.T)
200
201         # Compute rotated initial data
202         X = self._grid.get_nodes(flat=True)
203         values = self._initalpacket.evaluate_at(X, prefactor=True)
204         values = tuple([val.reshape(self._grid.get_number_nodes()) for
    val in values])
205         self._psi.set_values(values)
206
207         # rotate grid back to original orientation
208         self._grid.rotate(R)
209
210         #calculate timesteps
211         # each j is an individual time steps
212         for j in range(nsteps):
213             # each i is an intermediate time step in the splitting
214             for i in range(len(self._a)):
215
```

```
216                 ### Calculate potential flow map \Phi_V = \Phi_B * \Phi_
    {\phi}###
217 
218                 # Integral -\int_{tspan[0]}^{tspan[1]}B^2(s)ds and its
    associated propagation
219                 # does not need to be rotated, as B^2(s) is independent
    of rotations
220                 # (see [Gradinaru and Rietmann, 2020]. Remark 3.1)
221                 minus_B_squared = lambda t: (-1.0) * dot(B(t), B(t))
222                 A = 1.0 / 8.0 * MagneticField.matrix_quad(minus_B_squared
    , t_a, t_a + self._a[i]*self._dt)
223 
224                 X = self._grid.get_nodes(flat=True)
225                 VB = sum(X * dot(A, X))
226                 VB = VB.reshape(self._grid.get_number_nodes())
227                 #define the magnetic field propagator \Phi_B = \int{ -i/
    eps^2 * \int{ B^2}}
228                 prop = exp(-1.0j / self._eps**2 * VB)
229 
230                 #apply the propagator
231                 values = self._psi.get_values()
232                 values = [prop * component for component in values]
233 
234                 #define the propagator \Phi_{\phi} (for electric and
    spatial potential)
235                 # these potential are not independent of rotations and
    need to be rotated
236                 self._potential.calculate_exponential(-1.0j *  self._a[i
    ]*self._dt /self._eps**2)
237 
238                 # rotate and evaluate potentials
239                 self._grid.rotate(R.T)
240                 VE = self._potential.evaluate_exponential_at(self._grid)
241                 self._VE = tuple([ve.reshape(self._grid.get_number_nodes
    ()) for ve in VE])
242 
243                 # apply the propagator
244                 values = [self._VE * component for component in values]
245                 self._grid.rotate(R)
246 
247                 # define time step for Magnus Integrator
248                 t_interval[0] = t_b
249                 t_interval[1] = t_b + self._b[i]*self._dt
250 
251                 # calculate Magnus Integrator
252                 U = (FourierMagneticPropagator._Magnus_CF4(t_interval, B,
     1)).T
253                 R = dot(R , U)
254                 if(R.shape != U.shape):
255                     raise ValueError("Shapes of R and U do not match")
256 
257                 # check for obsolete splitting steps
```

```
258                 if(self._b[i] != 0):
259                     ### calculate kinetic flow map \Phi_{-\Delta} ###
260 
261                     #go to fourier space
262                     values = [fftn(component) for component in values]
263 
264                     # calculate the kinetic operator
265                     self._KO = KineticOperator(self._grid, self._eps)
266                     self._KO.calculate_exponential(-0.5j * self._eps**2 *
    self._b[i]*self._dt)
267 
268                     #calculate and apply the kinetic propagator
269                     TE = self._KO.evaluate_exponential_at()
270                     values = [TE * component for component in values]
271 
272                     # Go back to real space
273                     values = [ifftn(component) for component in values]
274 
275                 # save data
276                 self._psi.set_values(values)
277 
278                 #update time grids
279                 t_a = t_a + self._a[i]*self._dt
280                 t_b = t_b + self._b[i]*self._dt
281 
282         return tspan[0]
283 
284 
285     def propagate(self, tspan):
286         r"""This method does nothing.
287         """
```

# B Time evolution

## B.1 Threefold Morse Potential

The time evolution of the wavepacket $\Psi$ is portrayed in fig. 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 $\Psi$ is portrayed in fig. B.2. At a position $x$ the color encodes the phase of $\Psi(x)$, the brightness of the pixel encodes the intensity $|\Psi(x)|$.

## References

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

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

[3] Erwan Faou, Vasile Gradinaru, and Christian Lubich. Computing semiclassical quantum dynamics with hagedorn wavepackets. *SIAM J. Scientific Computing*, 31:3027–3041, 01 2009.

[4] 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.

[5] Michael Reed and Barry Simon. *Fourier Analysis, Self-Adjointness, Volume 2.* Academic Press, Boston, 1975.

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

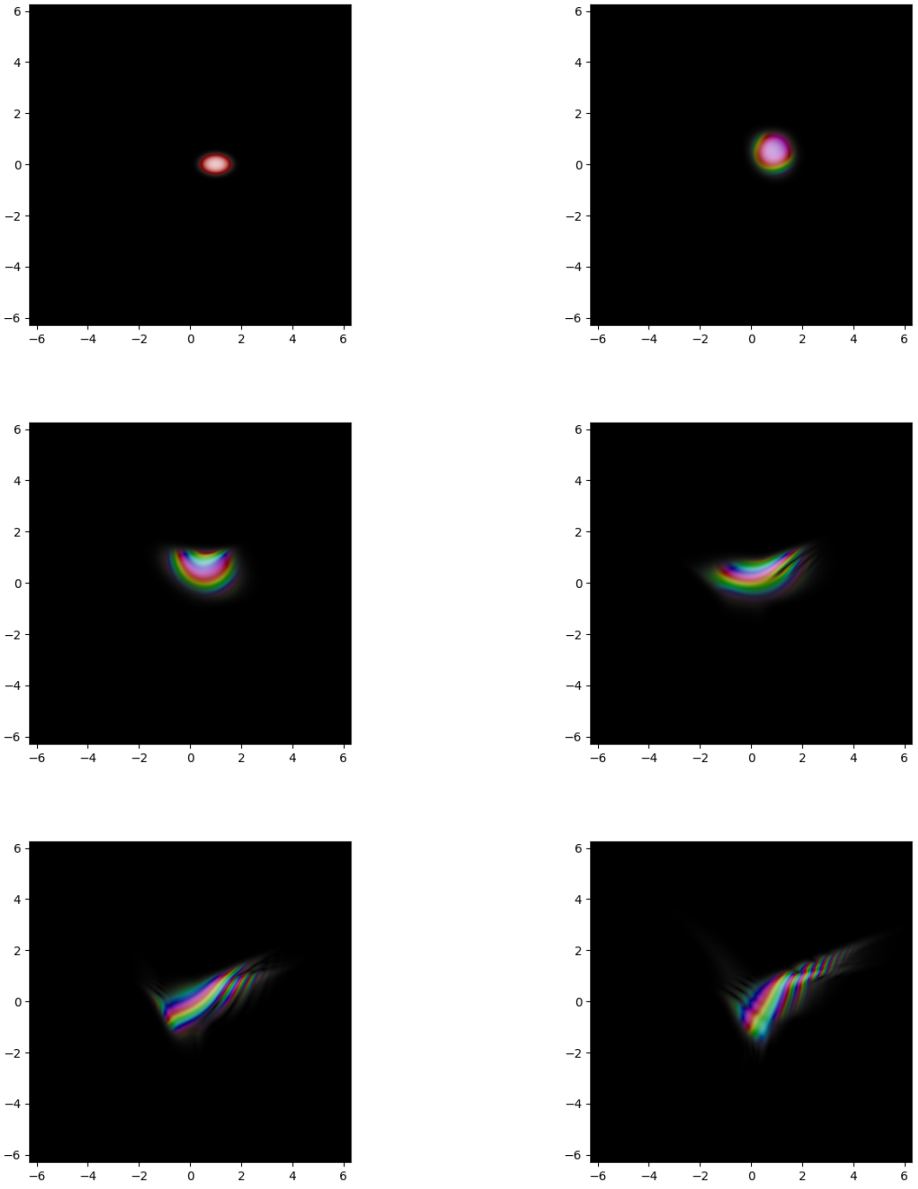


Figure B.1: Time evolution of the wavepacket $\Psi$ with initial data $\Psi_0$ in the threefold morse potential and a homogeneous, time-independent magnetic field.

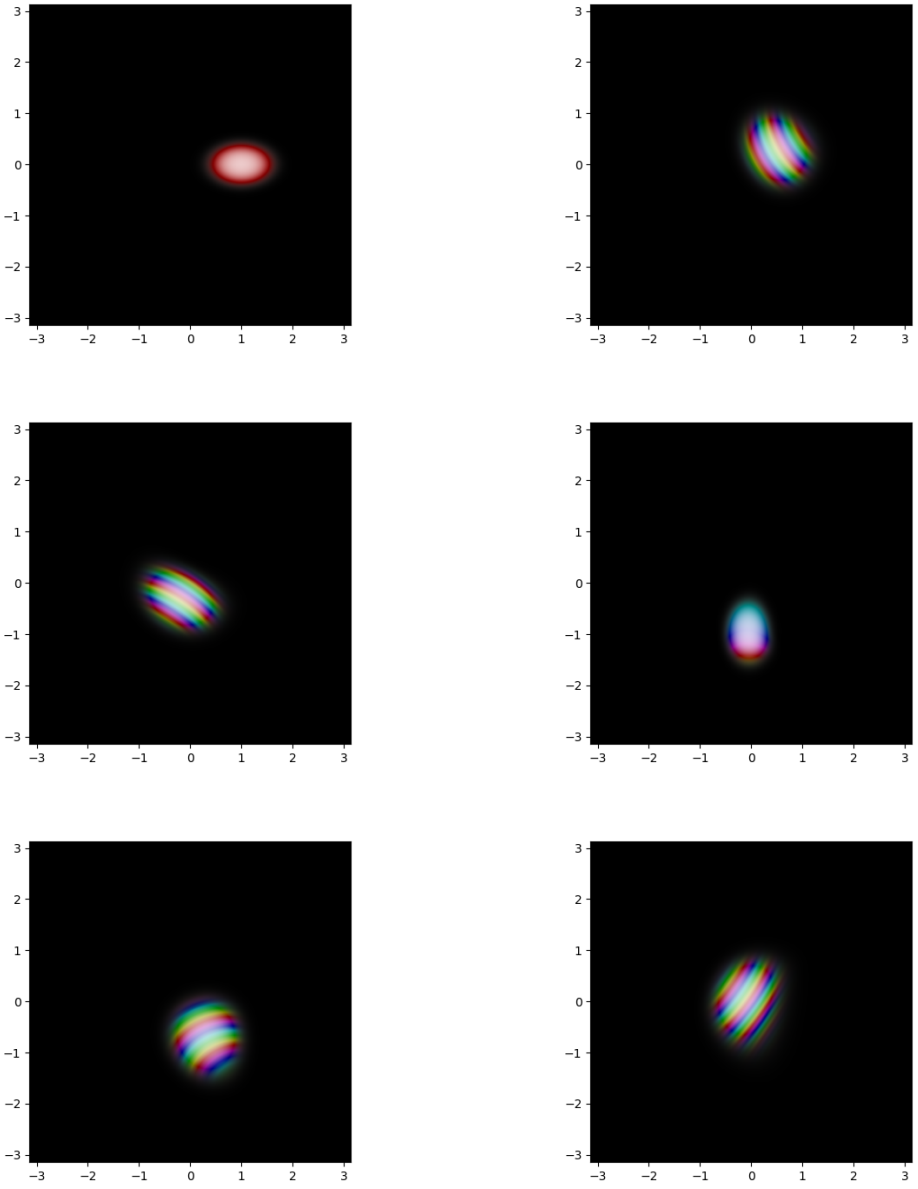


Figure B.2: Time evolution of the wavepacket $\Psi$ with initial data $\Psi_0$ in the torsional potential and a homogeneous, time-independent magnetic field.