ALGORITHMS FOR NON-ADIABATIC TRANSITIONS WITH ONE-DIMENSIONAL WAVEPACKETS

Bachelor Thesis

written by
Raoul Bourquin

supervised by
Dr. Vasile Grădinaru
and
Prof. Dr. Ralf Hiptmair

Seminar for Applied Mathematics
ETH Zurich

Spring semester 2010

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In this chapter we introduce the fundamental physical and mathematical ideas and structures on which the other chapters build. The central objects here are the time-dependent Schrödinger equation and a non-adiabatic potential.

1.1 The time-dependent Schrödinger equation

Time-dependent problems in quantum physics are governed by the time-dependent Schrödinger equation:

$$i\hbar\frac{\partial}{\partial t}\left|\varphi\right\rangle = H\left|\varphi\right\rangle \ . \tag{1.1}$$

The Hamiltonian of the system in consideration is given by H, and the function $\varphi\left(x,t\right)$ represents the wave function dependent on position x and time t. In d space dimensions this is

$$\begin{aligned}\varphi : \mathbb{R}^d \times \mathbb{R} &\to \mathbb{C}\\ (x,t) &\mapsto \varphi\left(x,t\right) \ .\end{aligned}$$

There are various mathematical restrictions on what is a valid wave-function. For example φ has to be square-integrable. Most of these preconditions have little importance for us.

1.2 Semiclassical scaling

We use the semiclassical scaling, where $\varepsilon > 0$ is a real parameter [1]. The equation still keeps its mathematical form:

$$i\varepsilon^2 \frac{\partial}{\partial t} |\psi\rangle = H |\psi\rangle \ . \tag{1.2}$$

It's well known that we get the classical dynamics from the limit $\hbar \to 0$. The same holds of course for the semiclassical parameter ε and for bigger ε we get an increasing amount of quantum effects.

The Hamiltonian operator H is composed of two parts, the kinetic operator T and the potential operator V. Thus we can split H as $H = T + V$ with the following definitions for both operators. The mass m which is present in the common definition of the kinetic operator is included in the parameter ε.

$$\begin{aligned} T &:= -\frac{1}{2}\varepsilon^4 \frac{\partial^2}{\partial x^2} \\ V &:= V(x) \end{aligned} \tag{1.3}$$

The potential is a real valued function depending only on space but not on time. This static potential results from the Born-Oppenheimer approximation for the electronic structure problem. For a more detailed theoretical background see for example reference [16]. Assume the potential is given by:

$$\begin{aligned} V : \mathbb{R}^d &\to \mathbb{R} \\ x &\mapsto V(x) \end{aligned} \tag{1.4}$$

then this allows us to solve the Schrödinger equation by separation of variables and obtain an analytical result for the time propagation of a quantum state $|\psi(t)\rangle$

$$|\psi(t)\rangle = e^{-\frac{i}{\varepsilon^2}Ht} |\psi(0)\rangle \ . \tag{1.5}$$

The solutions to this time propagation have fine details. A typical wavepacket is highly oscillatory with a wavelength $\mathcal{O}(\varepsilon^2)$ localised in space with $\mathcal{O}(\varepsilon^2)$ and moving with a velocity of $\mathcal{O}(1)$. This tiny structures are a challenge for the algorithms simulating them. We would need a very fine grid and thus a huge bunch of grid nodes.

[1]Other authors use ε or even $\hbar$ (without its physical meaning and value) for the quantity we denote by ε^2.

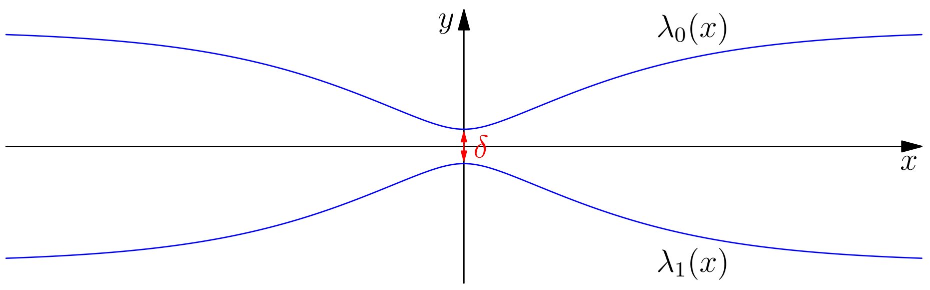


Figure 1.1: Example of an avoided crossing of two energy levels.

1.3 Non-adiabatic potentials, avoided crossings

Non-adiabatic potentials are potentials that consist of multiple energy levels. These energy levels may intersect each other, but we are interested in a situation that is called an *avoided crossing*. That is, the two energy surfaces always stay in a minimal distance. We call this distance the energy gap and denote it by δ. A very simple example of such an avoided crossing with only two energy levels is shown in figure 1.1.

Based on the class of physical potential in consideration we have a strict monotone order of the eigenvalues for all x in our space [2].

$$\lambda_0(x) > \lambda_1(x) > \ldots > \lambda_{N-1}(x) \qquad \forall x \tag{1.6}$$

This global consistent order allows us to sort the eigenvalues and the corresponding eigenvectors uniquely in decreasing order.

For a more elaborate study of the mathematical details and a classification of different types of avoided crossings see reference [8].

We are now interested what happens with an incoming wave-packet $|\psi\rangle$ while it traverses the narrow part. The magnitude δ of the energy gap plays an important role in this process.

1.4 A vector of states

For the study of avoided crossings of the energy levels we are interested in vector valued states $|\Psi\rangle$. Each component of this vector represents a part of the wave function being on the corresponding energy surface.

To describe the dynamics of these states, we need to generalize the Schrödinger equation to a vector valued version. This is not difficult to do, basically the

[2] This is a fairly strong assumption that can be replaced by much weaker formulations more suitable for mathematical analysis. But for our purpose it is sufficient and these details don't really matter.

extended equation looks exactly like (1.2) but with the difference that H is a matrix now. Let's write down this in more detail because we will refer to it over and over again.

Assume we deal with N different states hence $|\Psi\rangle$ consists of N components $\varphi_i(x)$. And the Hamiltonian becomes a real valued symmetric $N \times N$ matrix. This gives the following expression for the time-dependent Schrödinger equation

$$i\varepsilon^2 \frac{\partial}{\partial t} \left| \begin{pmatrix} \varphi_0 \\ \vdots \\ \varphi_{N-1} \end{pmatrix} \right\rangle = \begin{pmatrix} & H & \end{pmatrix} \underbrace{\left| \begin{pmatrix} \varphi_0 \\ \vdots \\ \varphi_{N-1} \end{pmatrix} \right\rangle}_{|\Psi\rangle} . \tag{1.7}$$

1.5 The potential

In the case of a non-adiabatic potential with multiple energy levels, the potential V becomes a matrix. We assume that V depends on space x but not on time t, thus it is time-independent.

The matrix representing V is symmetric and with entries $v_{i,j} \equiv v_{j,i} \in \mathbb{R}$. We may write a general unspecified potential as

$$V(x) =: \begin{pmatrix} v_{0,0}(x) & \cdots & v_{0,N-1}(x) \\ \vdots & & \vdots \\ v_{N-1,0}(x) & \cdots & v_{N-1,N-1}(x) \end{pmatrix} \tag{1.8}$$

where each of the matrix entries $v_{i,j}(x)$ is a real valued function

$$\begin{aligned} v_{i,j} : \mathbb{R}^d &\rightarrow \mathbb{R} \\ x &\mapsto v_{i,j}(x) \end{aligned}$$

on its own. These functions are assumed to be smooth.

1.5.1 Diagonalization of the potential

We are much more interested in the potential's eigenvalues which are the energy levels of our system. A well known result from linear algebra tells us that symmetric matrices always have only real eigenvalues. Therefore we can diagonalize this matrix and obtain pure real eigenvalues $\lambda_i(x)$ that depend on the space variable x.

The diagonalization itself is performed by orthogonal matrices, the same theorem as above guarantees that we have a full set of orthogonal eigenvectors $\nu_i(x)$

which depend of course on x too.

Given the full set of eigenvalues $\lambda_0(x), \ldots, \lambda_{N-1}(x)$ and the corresponding eigenvectors of $V(x)$ denoted by $\nu_0(x), \ldots, \nu_{N-1}(x)$ the spectral decomposition of the potential's matrix reads

$$\Lambda(x) = M^{-1}(x) V(x) M(x) \tag{1.9}$$

where the matrix Λ is diagonal with the eigenvalues λ_i on its diagonal. The transformation matrix M is orthogonal and contains the eigenvectors as columns:

$$M := \begin{pmatrix} \nu_0(x) & , \ldots , & \nu_{N-1}(x) \end{pmatrix} . \tag{1.10}$$

The special case for 2 energy levels

A general potential that only contains two energy levels is given by a symmetric two by two matrix. In this case we can write down the eigenvalues for the potential in closed form. The following formulae are defined in more detail in [11]. Suppose the potential matrix is given by

$$V(x) := \begin{pmatrix} v_1 & v_2 \\ v_2 & -v_1 \end{pmatrix} \tag{1.11}$$

with trace $\operatorname{Tr}(V) = 0$. Then we define a θ as

$$\theta := \frac{1}{2} \arctan\left(\frac{v_2}{v_1}\right) . \tag{1.12}$$

For the numerical computation we have to use the `atan2` function to get the signs correct. Finally we can write the two eigenvectors as

$$\nu_0 := \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} \quad \nu_1 := \begin{pmatrix} -\sin(\theta) \\ \cos(\theta) \end{pmatrix} . \tag{1.13}$$

Obviously they are orthogonal and normed. Remember that if we sort the λ_i we have to change the order of the eigenvectors as well. It's not guaranteed that ν_0 always belongs to λ_0.

1.5.2 Basis transformations of states

For various calculations later on we need to be able to transform states from and to the eigenbasis. This is in principle a trivial process of linear algebra, but lets briefly note the important points.

The transformation from the eigenbasis to the canonical basis will be important when we set up the initial values for a simulation. Assume we have a wave function $|\varphi^e\rangle$ given in the eigenbasis. The transformed state in the canonical basis is given by

$$|\varphi^c\rangle = M\,|\varphi^e\rangle \tag{1.14}$$

where M contains the column vectors ν_i.

The opposite transformation becomes important when evaluating observables. Given a state $|\varphi^c\rangle$ in the canonical basis, the image of the transformation into the eigenbasis is

$$|\varphi^e\rangle = M^{\mathrm{T}}\,|\varphi^c\rangle \tag{1.15}$$

where we simplified $M^{-1} = M^{\mathrm{T}}$ for real orthogonal matrices.

1.6 The matrix exponential

The exact time propagator for the Schrödinger equation is $e^{-\frac{i}{\varepsilon^2}Ht}$ as shown in equation (1.2). If the Hamiltonian is matrix valued then this expression becomes a matrix exponential. For that reason we need to think about some aspects of this topic too.

1.6.1 Symbolic matrix exponential

Given a general 2×2 square matrix $M \in \mathbb{C}^{2\times 2}$ we can derive an analytical closed form expression for its exponential. We now try to find an explicit formula for the exponential $\exp(M)$ of this matrix. For brevity we just show the formula. A detailed derivation and the proof can be found in reference [2].

Assume that our matrix looks like

$$M := \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{1.16}$$

with four possibly complex entries. Then

$$e^M = e^{\frac{a+d}{2}} \begin{pmatrix} \cosh(\Delta) + \frac{a-d}{2}\frac{\sinh(\Delta)}{\Delta} & b\frac{\sinh(\Delta)}{\Delta} \\ c\frac{\sinh(\Delta)}{\Delta} & \cosh(\Delta) - \frac{a-d}{2}\frac{\sinh(\Delta)}{\Delta} \end{pmatrix} \tag{1.17}$$

where Δ is a discriminant of the form

$$\Delta := \frac{1}{2}\sqrt{(a-d)^2 + 4bc}\,. \tag{1.18}$$

In the case of $\Delta = 0$ we need to be careful because of a possible division by zero and therefore consider a special case

$$e^M = e^{\frac{a+d}{2}} \begin{pmatrix} 1 + \frac{a-d}{2} & b \\ c & 1 - \frac{a-d}{2} \end{pmatrix} . \tag{1.19}$$

1.6.2 Numerical matrix exponential

For matrices of size bigger than 2 by 2 there is no applicable symbolic expression we could use. Hence the only possible solution is a numerical approximation. There are many such approximations known in literature, see for example the excellent survey of standard methods in reference [14, 15].

In our implementation we just use the primitive function `expm` based on Padé approximation available in scipy [13]. But any other method like for example Krylov methods can be used.

The general process of computing the matrix exponential of $V(x)$ numerically for all grid nodes γ is shown in figure 1.2. We start with evaluating the matrix V of scalar functions at all grid nodes γ. These values are stored in a suitable data structure. Then we slice this block to get the matrix that belongs to a single node γ_l. The exponential of this small $N \times N$ matrix is computed numerically, for example by `expm`. After we did this for all grid nodes we can glue together all the exponentials and arrive at a data structure identical to what we began with.

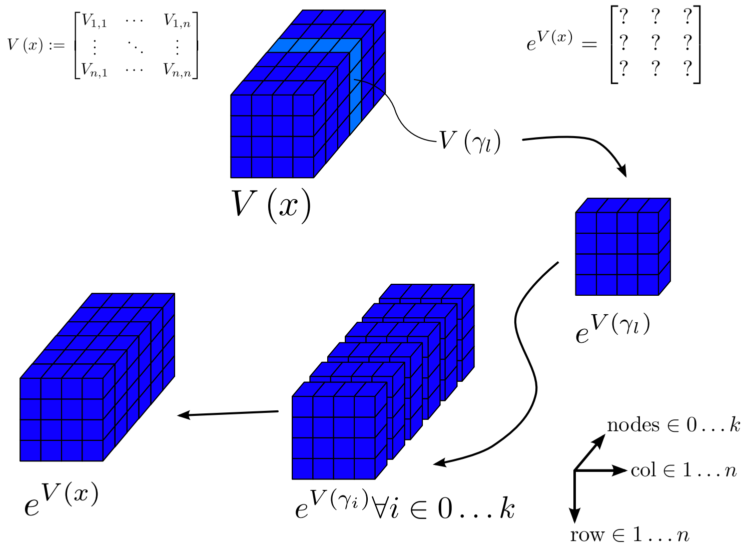

$V(x) := \begin{bmatrix} V_{1,1} & \cdots & V_{1,n} \\ \vdots & \ddots & \vdots \\ V_{n,1} & \cdots & V_{n,n} \end{bmatrix}$
$e^{V(x)} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$
$V(\gamma_l)$
$V(x)$
$e^{V(\gamma_l)}$
$e^{V(x)}$
$e^{V(\gamma_i)} \forall i \in 0 \ldots k$
nodes $\in 0 \ldots k$
col $\in 1 \ldots n$
row $\in 1 \ldots n$

Figure 1.2: Computation of the matrix exponential.

Chapter 2

Time propagation by operator splitting

In this chapter we will present an algorithm based on an analytical approximation of the matrix exponential in the time propagation operator.

The analytic explicit time propagation for an initial value of the time-dependent Schrödinger equation (1.2) can be written as

$$|\psi(x,t)\rangle = \exp\left(-\frac{i}{\varepsilon^2}Ht\right)|\psi(x,0)\rangle \tag{2.1}$$

In our case the Hamiltonian H is a square matrix of small size. Let's write this expression in a more detailed fashion

$$\exp\left(-\frac{i}{\varepsilon^2}Ht\right) = \exp\left(-\frac{i}{\varepsilon^2}(T+V)\,t\right) = \exp\left(\frac{-i}{\varepsilon^2}Tt + \frac{-i}{\varepsilon^2}Vt\right) \tag{2.2}$$

where the kinetic and potential operators T and V are given by (1.3).

2.1 Operator splitting

Given a linear ordinary differential equation

$$\begin{aligned} \dot{u} &= (A+B)\,u \\ u(0) &= u_0 \end{aligned} \tag{2.3}$$

with explicit solution

$$u\left(t\right)=e^{(A+B)t}\cdot u_0 \tag{2.4}$$

If A and B were matrices it's not possible to calculate the exponential in general. Only if A and B commute we can apply the Baker-Campbell-Hausdorff formula and write $e^{(A+B)}$ as $e^A e^B$. However in our application the two operators T and V do never commute.

A possible approximation is an operator splitting. See reference [12] and [5] for the details of this ansatz. A time-stepping scheme for small but finite time steps τ is:

$$u\left(t+\tau\right)\approx e^{\frac{1}{2}\tau B}e^{\tau A}e^{\frac{1}{2}\tau B}u\left(t\right)\ . \tag{2.5}$$

This is called a symmetric Lie-Trotter splitting. Now we apply this scheme to the equation (2.1) with $A := -\frac{i}{\varepsilon^2}T$ and $B := -\frac{i}{\varepsilon^2}V$. This yields for the propagation operator

$$e^{\frac{1}{2}\tau\frac{-i}{\varepsilon^2}V}e^{\tau\frac{-i}{\varepsilon^2}T}e^{\frac{1}{2}\tau\frac{-i}{\varepsilon^2}V}+\mathcal{O}\left(\tau^3\right) \tag{2.6}$$

which is of locally third order. The approximative time propagation of the Schrödinger equation reads then

$$\left|\psi\left(t+\tau\right)\right\rangle = e^{-\frac{i}{2\varepsilon^2}\tau V}e^{-\frac{i}{\varepsilon^2}\tau T}e^{-\frac{i}{2\varepsilon^2}\tau V}\left|\psi\left(t\right)\right\rangle \tag{2.7}$$

2.2 The propagation algorithm

With the operator splitting we gained the chance to perform each of these three propagation steps individually. This comes in handy, we can utilize the fact that the kinetic operator T is diagonal in momentum space while the potential V is diagonal in position space. Thus we switch to momentum space for the propagation by $e^{-\frac{i}{\varepsilon^2}\tau T}$ and back to position space afterwards. This change of basis is done by a Fourier transformation and can be performed efficiently by a fast Fourier transform algorithm. With all these parts put together the time propagation now reads

$$\left|\psi\left(x,t+\tau\right)\right\rangle = e^{-\frac{i}{2\varepsilon^2}\tau V(x)}\mathcal{F}^{-1}\left(e^{-\frac{i}{\varepsilon^2}\tau T(\omega)}\mathcal{F}\left(e^{-\frac{i}{2\varepsilon^2}\tau V(x)}\left|\psi\left(x,t\right)\right\rangle\right)\right) \tag{2.8}$$

where $\mathcal{F}\left(\cdot\right)$ denotes a formal Fourier transformation. This formula describes how to advance a single time step of duration τ. The whole algorithm consists of as many iterations of this formula as desired.

A further simplification of (2.8) is possible when we take into account the exact expression for T given by equation (1.3) with the mass m set to 1. The result is $e^{-\frac{i}{\varepsilon^2}\tau T(\omega)} = e^{-i\varepsilon^2\tau\widetilde{T}(\omega)}$ with $\widetilde{T} := -\frac{1}{2}\frac{\partial^2}{\partial x^2}$.

Both the kinetic as well as the potential operator are time-independent and we can precalculate their exponentials. This reduces the per iteration cost of the time- stepping algorithm by a big amount.

The expression $-\frac{i}{2\varepsilon^2}\tau V(x)$ easily evaluates to a scalar for any given x. This won't cause any troubles in the exponential. For the kinetic operator we need to go to Fourier space. Using the linearity of $\mathcal{F}$ and the above definition for $\widetilde{T}$:

$$\mathcal{F}\left(-i\varepsilon^2\tau\widetilde{T}\right) = -i\varepsilon^2\tau\mathcal{F}\left(\widetilde{T}\right) = -i\varepsilon^2\tau\mathcal{F}\left(-\frac{1}{2}\frac{\partial^2}{\partial x^2}\right)$$

As known from Fourier theory it holds that $\mathcal{F}\left(f^{(n)}(x)\right) = (i\omega)^n \mathcal{F}(f(x))$ for the n^{th} derivatives of a suitable function f and with ω being the Fourier variable. Use of this identity gives

$$\mathcal{F}\left(-\frac{1}{2}\frac{\partial^2}{\partial x^2}f(x)\right) = -\frac{1}{2}(i\omega)^2\mathcal{F}(f(x))$$

If we now simplify this result further, we get $\mathcal{F}\left(\widetilde{T}\right) = -\frac{1}{2}(i\omega)^2$ and thus

$$\mathcal{F}\left(-i\varepsilon^2\tau\widetilde{T}\right) = -\frac{i}{2}\varepsilon^2\tau\omega^2 \tag{2.9}$$

We got rid of the partial derivatives in the exponent at the cost of an additional real scalar ω. Finally we introduce the following notation

$$\begin{aligned} T_e(\omega) &:= \exp\left(-\frac{1}{2}i\varepsilon^2\tau\omega^2\right) \\ V_e(x) &:= \exp\left(-\frac{i}{2\varepsilon^2}\tau V(x)\right) \end{aligned} \tag{2.10}$$

for the precalculated propagation operators. Both exponentials only contain numbers. With these definitions the algorithm given by equation (2.8) becomes

$$\psi_{n+1}(x) = V_e(x)\mathcal{F}^{-1}\Big(T_e(\omega)\underbrace{\mathcal{F}\big(V_e(x)\cdot\psi_n(x)\big)}_{\widehat{\psi}_n(\omega)}\Big) \tag{2.11}$$

where ψ_n is a short notation for $\psi(x, t = n\tau)$ with n denoting the number of the current time step.

2.2.1 The discretized space

For the numerical simulation we first define the computational domain $\Omega \in \mathbb{R}^d$. In our case where we deal with only one space dimension and $d = 1$, this domain is simply an interval on the real line. Without loss of generality we can assume Ω to be centred around the origin and bounded by α. To simplify further, also with respect to the Fourier transformation, we write the constant α as a multiple of π and denote the scaling factor by $f \in \mathbb{R}$.

$$\Omega := [-f\pi, f\pi] \tag{2.12}$$

The domain is still part of the continuum. For the numerical computation we need to discretize the space and introduce a grid on Ω. Denote the number of grid nodes by n and the grid spacing by h. Then a grid Γ is given by

$$\Gamma := \{\gamma_0 < \gamma_1 < \ldots < \gamma_{N-1}\} \tag{2.13}$$
$$\gamma_0 \equiv -f\pi \quad \text{and} \quad \gamma_{N-1} \equiv f\pi \tag{2.14}$$

with equidistant grid nodes $\gamma_i \in \Omega$. Additionally to the grid in position space we need a grid in the Fourier space for computing $T_e(\omega)$. Suppose the number n of grid nodes is given as a power of 2, then we have $n = 2^k$. Now the grid $\widehat{\Gamma}$ in Fourier space is given as

$$\widehat{\Gamma} := \{\omega_0 < \omega_1 < \ldots < \omega_{N-1}\} \tag{2.15}$$
$$\omega_0 := 1 - 2^{k-1} \quad \text{and} \quad \omega_{N-1} := 2^{k-1} \tag{2.16}$$

Depending on the implementation of the discrete Fourier transform we want to use we have to shift the nodes with negative sign and reorder the set $\widehat{\Gamma}$.

2.2.2 Discretized time evolution operators

The time propagation operator exponentials T_e and V_e given by equation (2.10) become

$$\begin{aligned} \widetilde{T}_e(\widetilde{\omega}) &:= \exp\left(-\frac{1}{2} i \varepsilon^2 \tau \widetilde{\omega}^2\right) \\ \widetilde{V}_e(\gamma) &:= \exp\left(-\frac{i}{2\varepsilon^2} \tau V(\gamma)\right) \end{aligned} \tag{2.17}$$

where $\widetilde{\omega} := \frac{\omega}{n}$ and $\omega \in \widehat{\Gamma}$ when transformed to discretized space.

2.2.3 Pseudo code

With the results from the last sections we are ready to write down a pseudo code for the time propagation of a $|\psi\rangle$ which solves the scalar semiclassical Schrödinger equation (1.2). The algorithm 1 shows a straight forward implementation of (2.11). The exponentials of the operators are evaluated and stored in a vectorized fashion simultaneously for all grid nodes γ_i.

Algorithm 1 Time propagation with operator splitting of H for $|\psi\rangle$

```
Require: The precalculated operator exponentials from (2.10)
  // Propagate with the potential $V$
  $\psi' := V_e \cdot \psi$
  // Fourier transform
  $\widehat{\psi'} := \mathcal{F}(\psi')$
  // Propagate with the kinetic operator $T$
  $\widehat{\psi''} := T_e \cdot \widehat{\psi'}$
  // Apply inverse Fourier transform
  $\psi'' := \mathcal{F}^{-1}\left(\widehat{\psi''}\right)$
  // Propagate again with the potential $V$
  $\psi^{(k+1)} := V_e \cdot \psi''$
  return $\psi^{(k+1)}$
```

2.3 Vector valued states

In the last section we only considered the simpler case with a potential function and a scalar state $|\psi\rangle$. Now we want to extend the theory to matrix valued potentials like (1.8) together with vector valued states $|\Psi\rangle$ defined by (1.7). The goal of this effort is to get an extended version of algorithm 1 that handles this more general case.

The question is now which parts we have to generalize. Because we defined the Lie-Trotter splitting in an abstract context, the formula (2.7) stays the same. The only things that change are the exponentials therein that become now matrix exponentials. Hence we have to derive new formulae analogous to the ones of definition (2.10).

With the generalized definitions of T_e and V_e the core of the time propagation algorithm as given in (2.11) is still valid and can be reused.

2.3.1 Propagation operator exponentials

Assume our state $|\Psi\rangle$ consists of N components $\varphi_0, \ldots, \varphi_{N-1}$. Thus the Hamiltonian operator $\mathbf{H}$ has to be a $N \times N$ matrix [1]. We may split $\mathbf{H} = \mathbf{T} + \mathbf{V}$ and

[1]Don't confuse this with the matrix representation $H_{i,j} := \langle \phi_i \,|\, H \,|\, \phi_j \rangle$ of a Hamiltonian operator H for a given set of basis functions $\phi_0, \ldots, \phi_k$.

write

$$\mathbf{H} = \begin{pmatrix} T_0 & & \\ & \ddots & \\ & & T_{N-1} \end{pmatrix} + \begin{pmatrix} v_{0,0}(x) & \cdots & v_{0,N-1}(x) \\ \vdots & & \vdots \\ v_{N-1,0}(x) & \cdots & v_{N-1,N-1}(x) \end{pmatrix} \tag{2.18}$$

where we used the definition of T given by (1.3) and the potential matrix introduced in (1.8). We can simplify the first matrix by assuming that all the T_i are identical operators.

With this last step, the exponential $e^{-\frac{i}{\varepsilon^2}\tau\mathbf{T}}$ of the diagonal kinetic operator matrix $\mathbf{T}$ becomes rather easy and the problem reduces to what we did in the last section. The solution for a single component is given by T_e of (2.10). Therefore we get

$$\mathbf{T_e} := \exp\left(-\frac{i}{\varepsilon^2}\tau\mathbf{T}\right) = \begin{pmatrix} T_e & & \\ & \ddots & \\ & & T_e \end{pmatrix} \tag{2.19}$$

For the potential operator matrix the process is much more difficult as $V(x)$ is in general not diagonal. Here we really need full-fledged matrix exponentials for $e^{-\frac{i}{2\varepsilon^2}\tau V(x)}$ without any possibilities for simplification. For the special case where $N = 2$ we can use the analytical formula for the matrix exponential given by (1.17) and adapt it for symmetric matrices. Otherwise where $N > 2$ numerical techniques have to be used. For the sake of completeness we note:

$$\mathbf{V_e} := \exp\left(-\frac{i}{2\varepsilon^2}\tau V(x)\right) = \exp\left(-\frac{i}{2\varepsilon^2}\tau \begin{pmatrix} & V(x) & \end{pmatrix}\right) \tag{2.20}$$

Both, $\mathbf{T_e}$ and $\mathbf{V_e}$ are again matrices of the same dimension as $\mathbf{H}$. Putting all the parts together we arrive at

$$\Psi_{n+1}(x) = \mathbf{V_e}(x)\,\mathcal{F}^{-1}\Big(\mathbf{T_e}(\omega)\,\underbrace{\mathcal{F}\big(\mathbf{V_e}(x)\cdot\Psi_n(x)\big)}_{\widehat{\Psi}_n(\omega)}\Big) \tag{2.21}$$

which is a time-stepping scheme analogous to (2.11) but applicable to the Schrödinger equation (1.7) with vectorial states $|\Psi\rangle$.

2.3.2 Pseudo code

With the last result we can start to write down a pseudo code for the time propagation of a $|\Psi\rangle$. The algorithm 2 shows a straight forward implementation of (2.21). The values for $\mathbf{T_e}$ and $\mathbf{V_e}$ are precomputed for all grid nodes γ_i. We need to be careful with the formal matrix multiplications this time. Of course we do not build the matrix for $\mathbf{T_e}$ but rather multiply by the same T_e all the time. This is equivalent to the redefinition $\mathbf{T_e} := T_e$.

Algorithm 2 Time propagation with operator splitting of H for $|\Psi\rangle$

```
Require: The precalculated operator exponentials from (2.19) and (2.20)
  $\Psi^{(k)} = \{\varphi_0, \dots, \varphi_{N-1}\}$
  // Propagate with the potential $V$
  $\Psi' := \{0, \dots, 0\}$
  for $r := 0$ to $N-1$ do
    for $c := 0$ to $N-1$ do
      $\Psi'_r := \Psi'_r + \mathbf{Ve}_{r,c} \cdot \Psi^{(k)}_c$
    end for
  end for
  // Fourier transform the components
  $\widehat{\Psi}' := \{0, \dots, 0\}$
  for $r := 0$ to $N-1$ do
    $\widehat{\Psi}'_r := \mathcal{F}(\Psi'_r)$
  end for
  // Propagate with the kinetic operator $T$
  for $r := 0$ to $N-1$ do
    $\widehat{\Psi}''_r := \mathbf{T_e} \cdot \widehat{\Psi}'_r$
  end for
  // Apply inverse Fourier transform to the components
  $\Psi'' := \{0, \dots, 0\}$
  for $r := 0$ to $N-1$ do
    $\Psi''_r := \mathcal{F}^{-1}\left(\widehat{\Psi}''_r\right)$
  end for
  // Propagate again with the potential $V$
  $\Psi^{(k+1)} := \{0, \dots, 0\}$
  for $r := 0$ to $N-1$ do
    for $c := 0$ to $N-1$ do
      $\Psi^{(k+1)}_r := \Psi^{(k+1)}_r + \mathbf{Ve}_{r,c} \cdot \Psi''_c$
    end for
  end for
  return $\Psi^{(k+1)}$
```

2.4 Initial values

We just finished building an algorithm that can perform the time evolution of any given initial state. It's time to pay attention to these initial values and elaborate how we have to define them properly.

In the case of multiple energy levels λ_i we need to specify the initial values with extra care. In most cases we want to start with a single Gaussian wavepacket on only one energy level and nothing on the others. This Gaussian wavepacket

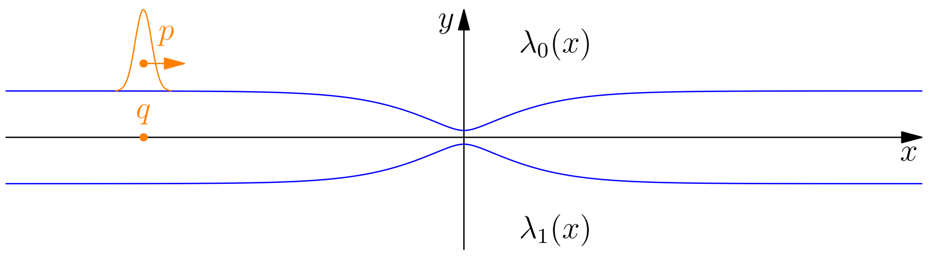


Figure 2.1: Example of Gaussian initial values on the upper energy level.

we start with is localized around position q and may have a momentum that is localized in momentum space around p. For an example how this could look like see figure 2.1 where we start with a right travelling packet on the left side of the avoided crossing.

It's apparent that the initial values are given in the eigenbasis of the potential V. But the simulation takes place in the canonical basis thus the initial values have to be transformed. This happens with a simple linear basis transformation from the eigenbasis into the canonical basis. The orthogonal matrix M that performs this task is given by the eigenvectors of our potential. We defined this transformation together with the inverse in the section 1.5.2.

As starting point of the time-stepping algorithm (2) we write now for the values $\Psi_0(x)$ just before our iteration first takes place

$$\Psi_0(x) := M\,|\Psi_{\mathrm{IV}}\rangle \tag{2.22}$$

where $|\Psi_{\mathrm{IV}}\rangle$ are the given initial values in the eigenbasis. Even if we start in most situations with a single Gaussian packet, the initial values can be arbitrary and different on each energy surface. In general they don't have to be localized wavepackets at all.

2.5 Observables

In this section we will look into the calculation of observables like the different energies and norms. Note that we always use the complex scalar product $\langle\cdot,\cdot\rangle$ here.

2.5.1 The norm of a wavepacket

The norm of a wavepacket is particularly interesting because it can be interpreted as the probability density for finding the particle in a specified infinitesimal volume element.

To calculate the norm of a wavepacket $|\Phi\rangle$ we start from the definition and use again a transformation to Fourier space and Parseval's identity. This gives us

$$\|\Psi(x)\|_{L^2}^2 := \langle \Psi \mid \Psi \rangle = \left\langle \widehat{\Psi} \,\middle|\, \widehat{\Psi} \right\rangle = \|\widehat{\Psi}(\omega)\|_{L^2}^2 \tag{2.23}$$

In the discretized space we evaluate Ψ at the grid nodes Γ and get a vector with n entries.

$$\begin{aligned}
\|\Psi(\Gamma)\|_2^2 &= \langle \Psi(\Gamma), \Psi(\Gamma) \rangle \\
&= \sum_i^n \overline{\Psi(\gamma_i)} \Psi(\gamma_i) \\
&= \frac{2\pi f}{n^2} \sum_k^n \overline{\widehat{\Psi}(\omega_k)} \widehat{\Psi}(\omega_k) \\
&= \frac{2\pi f}{n^2} \langle \widehat{\Psi}(\omega_k), \widehat{\Psi}(\omega_k) \rangle \\
&= \|\widehat{\Psi}(\widehat{\Gamma})\|_2^2
\end{aligned} \tag{2.24}$$

where we used a discrete Fourier transformation. Further, with the square removed by taking the root, we get

$$\|\widehat{\Psi}(\widehat{\Gamma})\|_2 = \frac{\sqrt{2\pi f}}{n} \|\widehat{\Psi}(\omega)\|_2 \tag{2.25}$$

This is the formula we use in the code because it's much cheaper to calculate the norm in Fourier space.

Notice that for the case of vector valued wave functions the norm of a single component φ_i is exactly the probability for finding the particle on the corresponding energy level E_i.

2.5.2 Energy of a wavepacket

The energy of a quantum wavepacket $|\Psi\rangle$ is given by the following integral

$$E = \langle \Psi \mid H \mid \Psi \rangle \tag{2.26}$$

Hence we investigate the detailed structure of this expression with the aim to easily calculate it's value. First we explicitly split the expression into the parts for potential and kinetic energy

$$\langle \Psi \mid H \mid \Psi \rangle = \langle \Psi \mid T \mid \Psi \rangle + \langle \Psi \mid V \mid \Psi \rangle \tag{2.27}$$

and then we plug in the operators' definitions given by (1.3). This yields

$$\langle \Psi \,|\, H \,|\, \Psi \rangle = \left\langle \Psi \,\middle|\, -\frac{1}{2}\varepsilon^4 \frac{\partial^2}{\partial x^2} \,\middle|\, \Psi \right\rangle + \langle \Psi \,|\, V(x) \,|\, \Psi \rangle \tag{2.28}$$

The first summand which represents the kinetic energy E_{kin} can be simplified using linearity

$$\begin{aligned} E_{\text{kin}} &= -\frac{1}{2}\varepsilon^4 \left\langle \Psi \,\middle|\, \frac{\partial^2}{\partial x^2} \,\middle|\, \Psi \right\rangle \end{aligned}$$

and by transformation to Fourier space

$$\begin{aligned} &= -\frac{1}{2}\varepsilon^4 \left\langle \mathcal{F}(\Psi) \,\middle|\, (i\omega)^2 \,\middle|\, \mathcal{F}(\Psi) \right\rangle \\ &= \frac{1}{2}\varepsilon^4 \left\langle \Psi(\omega) \,\middle|\, \omega^2 \,\middle|\, \Psi(\omega) \right\rangle \end{aligned}$$

Finally we get the following formula for the kinetic energy of a wavepacket $|\Psi\rangle$

$$E_{\text{kin}} = \frac{1}{2}\varepsilon^4 \int_\omega \overline{\widehat{\Psi}(\omega)} \cdot \omega^2 \cdot \widehat{\Psi}(\omega) \, d\omega \tag{2.29}$$

While we were able to simplify the expression for the kinetic energy quite a lot this is not possible in the same manner for a general potential $V(x)$.Thus we just apply Parseval's equality and write

$$\begin{aligned} E_{\text{pot}} &= \langle \Psi \,|\, V(x) \,|\, \Psi \rangle \\ &= \langle \mathcal{F}(\Psi(x)) \,|\, \mathcal{F}(V(x)\Psi(x)) \rangle \\ &= \int_\omega \mathcal{F}(\Psi(x)) \, \mathcal{F}(V(x)\Psi(x)) \, d\omega \end{aligned} \tag{2.30}$$

2.5.3 Energy of a vector valued wavepacket

In the case of vector valued wavepackets $|\Psi\rangle$ we have to think carefully what to compute. The basic equation (2.27) is still valid but we have to use the matrix valued operators $\mathbf{T}$ and $\mathbf{V}$

$$E = \langle \Psi \,|\, \mathbf{T} \,|\, \Psi \rangle + \langle \Psi \,|\, \mathbf{V} \,|\, \Psi \rangle \tag{2.31}$$

The computation of both energies has to be carried out in the canonical basis because we need the operators $\mathbf{T}$ and $\mathbf{V}$ whose representations we only know there. On the other hand we are interested in the energy of the wavepacket $|\Psi\rangle$

and its components φ_i measured in the eigenbasis. Thus there is an additional basis transformation M involved which drops out in the scalar case. Of course the overall energy is independent of any particular basis, but the energy of a component φ_i is not. In analogy to (2.29) and (2.30) we can write

$$E_{\text{kin}} = \left\langle \begin{pmatrix} \varphi_0 \\ \vdots \\ \varphi_{N-1} \end{pmatrix} \middle| M^{\mathrm{T}} \begin{pmatrix} T & & \\ & \ddots & \\ & & T \end{pmatrix} M \middle| \begin{pmatrix} \varphi_0 \\ \vdots \\ \varphi_{N-1} \end{pmatrix} \right\rangle \tag{2.32}$$

and

$$E_{\text{pot}} = \left\langle \begin{pmatrix} \varphi_0 \\ \vdots \\ \varphi_{N-1} \end{pmatrix} \middle| M^{\mathrm{T}} \begin{pmatrix} & & \\ & V(x) & \\ & & \end{pmatrix} M \middle| \begin{pmatrix} \varphi_0 \\ \vdots \\ \varphi_{N-1} \end{pmatrix} \right\rangle \tag{2.33}$$

For the calculation of the energy of a single component φ_i of $|\Psi\rangle$ we transform and measure according to this formula

$$E^i_{\text{pot}} = \left\langle \begin{pmatrix} 0 \\ \varphi_i \\ 0 \end{pmatrix} \middle| M^{\mathrm{T}} \mathbf{V} M \middle| \begin{pmatrix} 0 \\ \varphi_i \\ 0 \end{pmatrix} \right\rangle \tag{2.34}$$

for the potential energy or its identical counterpart with the operator $\mathbf{V}$ replaced by $\mathbf{T}$ for kinetic energy. In any case it holds that

$$\begin{aligned} E_{\text{total}} &= E_{\text{kin}} + E_{\text{pot}} \\ &= \sum_{i=0}^{N-1} E^i_{\text{kin}} + \sum_{i=0}^{N-1} E^i_{\text{pot}} \\ &= \text{constant} \end{aligned} \tag{2.35}$$

We have conservation of energy as the system is self-contained.

2.5.4 Energy computations in discretized space

For the energies we get within discretized space the following formulae where we again build a vector from evaluating the function ψ on the grid nodes Γ:

$$E_{\text{kin}} = \frac{2\pi f}{n^2} \frac{1}{2} \varepsilon^4 \langle \overline{\mathcal{F}(\psi(\Gamma))}, \omega^2 \mathcal{F}(\psi(\Gamma)) \rangle \tag{2.36}$$

and

$$E_{\text{pot}} = \frac{2\pi f}{n^2} \langle \overline{\mathcal{F}\left(\psi(\Gamma)\right)}, V\left(x\right) \mathcal{F}\left(\psi(\Gamma)\right) \rangle \tag{2.37}$$

For a vectorial wavepacket $|\Psi\rangle$ we have

$$E_{\text{pot}} = \frac{2\pi f}{n^2} \langle \mathcal{F}\left(\varphi_i\right), \mathcal{F}\left(\widetilde{\varphi}_i\right) \rangle \tag{2.38}$$

where $\left(\widetilde{\varphi}_0, \ldots, \widetilde{\varphi}_{N-1}\right)^{\mathrm{T}} := \widetilde{\Psi} = \mathbf{V}\Psi$ for the potential energy. To calculate the kinetic energy we apply the above formula (2.36) to each component ψ_i of Ψ separately.

Of course the Fourier transformation $\mathcal{F}\left(\cdot\right)$ has to be interpreted as a discrete Fourier transformation. The discrete Fourier transformation is then implemented as an `FFT` algorithm.

Chapter 3

Semiclassical wavepackets

3.1 Definition of semiclassical wavepackets

In this section we present a particular form of wavepackets defined by G. Hagedorn, see for example [6, 7] and particularly [9].

These wavepackets are a general class of orthonormal basis functions for an $L^2\left(\mathbb{R}^d\right)$ space. For the d dimensional space they are defined as follows. Let $q \in \mathbb{R}^d$ be the position and $p \in \mathbb{R}^d$ the momentum vector of the package. Further there are complex matrices $P, Q \in \mathbb{C}^{d\times d}$ which obey the following important relations

$$\begin{aligned} Q^{\mathrm{T}} P - P^{\mathrm{T}} Q &= 0 \\ Q^{\mathrm{H}} P - P^{\mathrm{H}} Q &= 2i\mathbb{1}\,. \end{aligned} \tag{3.1}$$

With these parameters we can now define the ground state wave function ϕ_0 depending on arbitrary but fixed parameters as

$$\begin{aligned} \phi_0\left[P, Q, p, q\right](x) := \left(\pi\varepsilon^2\right)^{-\frac{d}{4}} \det\left(Q\right)^{-\frac{1}{2}} \\ \cdot \exp\left(\frac{i}{2\varepsilon^2}\langle(x-q), PQ^{-1}(x-q)\rangle + \frac{i}{\varepsilon^2}\langle p, (x-q)\rangle\right) \end{aligned} \tag{3.2}$$

where $x \in \mathbb{R}^d$. Also ε enters this equation with a constant numerical value during all computations[1].

The eigenfunctions of the harmonic oscillator are contained as special cases in the more general formulae for these wavepackets.

[1] In contrast to some other authors we use the notation of $P := iB$ and $Q := A$ and $q := a$ for the position and $p := \eta$ for the momentum. The motivation for this change are the equations of motion of P and Q that become the classical equations.

For the semiclassical wavepackets we can define and use ladder operators in the same manner as one does for the harmonic oscillator. This analogy builds on the fact that these wavepackets diagonalize the general quadratic Hamiltonian. Before we define these operators, let's restrict the dimension of the position space to one. This way we can avoid some difficulties that have no relevance for us now.

3.1.1 Restriction to one space dimension

The restriction to one space dimension where $d = 1$ simplifies things a lot because the vectors p and q and especially the matrices P and Q all reduce to scalar values. Further we don't need to bother with multi-index notation for k.

First we simplify the ground state (3.2) to the one dimensional case:

$$\phi_0 \left[P, Q, p, q\right](x) := \left(\pi\varepsilon^2\right)^{-\frac{1}{4}} Q^{-\frac{1}{2}} \exp\left(\frac{i}{2\varepsilon^2} P Q^{-1} (x-q)^2 + \frac{i}{\varepsilon^2} p (x-q)\right) . \tag{3.3}$$

3.1.2 Ladder operators

Now let's take a closer look at the ladder operators. As mentioned above there exists a lowering operator $\mathcal{L}$ and a raising operator $\mathcal{R}$ for semiclassical wavepackets. We will use these ladder operators later for defining the wave functions ϕ_k of higher states $k \geq 1$. The ladder operators are defined as:

$$\begin{aligned} \mathcal{R} &= \frac{i}{\sqrt{2\varepsilon^2}} \left(\overline{P}(x-q) - i\varepsilon^2 \overline{Q}\left(\frac{\partial}{\partial x} - p\right)\right) \\ \mathcal{L} &= -\frac{i}{\sqrt{2\varepsilon^2}} \left(P(x-q) - i\varepsilon^2 Q\left(\frac{\partial}{\partial x} - p\right)\right) . \end{aligned} \tag{3.4}$$

It exists a lowest state which can not be lowered further by $\mathcal{L}$. This state is the zero state and acts as the bottom of this ladder:

$$\mathcal{L}\phi_0 = 0 . \tag{3.5}$$

On the other hand we can apply the raising operator to the ground state and create ϕ_1:

$$\phi_1 = \mathcal{R}\phi_0 . \tag{3.6}$$

In much the same way we can create ϕ_k for arbitrary k by applying $\mathcal{R}$ multiple times. To be more concrete, the following formulae bring the different states into relation:

$$\begin{aligned} \phi_{k+1} &= \frac{1}{\sqrt{k+1}} \mathcal{R} \phi_k \\ \phi_{k-1} &= \frac{1}{\sqrt{k}} \mathcal{L} \phi_k \,. \end{aligned} \tag{3.7}$$

To get the state ϕ_k from the given ϕ_0 we have to let $\mathcal{R}$ act k times. Together with the prefactors this yields

$$\phi_k := \frac{1}{\sqrt{k!}} \mathcal{R}^k \phi_0 \,. \tag{3.8}$$

Finally we can give an analytical closed form for ϕ_k

$$\begin{aligned} \phi_k \left[P, Q, p, q\right](x) := 2^{-\frac{k}{2}} (k!)^{-\frac{1}{2}} \left(\pi \varepsilon^2\right)^{-\frac{1}{4}} Q^{-\frac{k+1}{2}} \overline{Q}^{\frac{k}{2}} \cdot H_k \left(\varepsilon^{-1} \left|Q\right|^{-1} (x-q)\right) \\ \cdot \exp\left(\frac{i}{2\varepsilon^2} P Q^{-1} (x-q)^2 + \frac{i}{\varepsilon^2} p (x-q)\right) \end{aligned} \tag{3.9}$$

where $H_k(\xi)$ is the Hermite polynomial [2] of degree k.

To end this section, let's emphasize again the close relationship to the eigenfunctions of the harmonic oscillator. Suppose $P = i$, $Q = 1$, choose the origin as position and assume the wavepacket has no momentum, thus $p = q = 0$. Further, assume $\varepsilon = 1$. If we now plug these values in (3.9) we get

$$|\varphi_k\rangle = \frac{H_k(x) e^{-\frac{x^2}{2}}}{\pi^{\frac{1}{4}} 2^{\frac{k}{2}} \sqrt{k!}} \tag{3.11}$$

which is exactly the well known expression for the harmonic oscillator.

3.1.3 Numerical evaluation of basis functions

For the numerical simulation we will need to evaluate the functions $\phi_k(x)$ at some discrete grid nodes x_i. This seems to be a trivial task as we have a closed form expression for ϕ_k given by equation (3.9). Although there is this expression for all k it's a bad idea to use it directly. One critical point in this formula is

[2] We use the following definition for the Hermite polynomials:

$$H_k(\xi) := (-1)^k e^{\xi^2} \left(\frac{\partial}{\partial \xi}\right)^k e^{-\xi^2} \tag{3.10}$$

the factorial. It will soon result in a numerical overflow even for relatively small k. Therefore we need a better approach.

An idea is to evaluate the ground state and recursively calculate the higher states based on these values. The essential three term recursion can be obtained as follows. We start with the function for ϕ_0 which we can evaluate numerically without much troubles. It is just a Gaussian exponential. (Note that we omit a factor of $Q^{-\frac{1}{2}}$ for the moment.)

Applying the raising operator $\mathcal{R}$ once results in:

$$\phi_1(x) = Q^{-1}\sqrt{\frac{2}{\varepsilon^2}}(x-q)\cdot\phi_0 \tag{3.12}$$

and for the general case we get the following three term recursion

$$\phi_{k+1}(x) = \sqrt{\frac{2}{\varepsilon^2}}\frac{1}{\sqrt{k+1}}Q^{-1}(x-q)\,\phi_k(x) - \sqrt{\frac{k}{k+1}}Q^{-1}\overline{Q}\phi_{k-1}(x)\ . \tag{3.13}$$

This is exactly how the calculation is implemented in an efficient and numerically stable way. Because later we will need the values for all ϕ_k from $k=0$ up to a maximum $k_{\max} =: K$, it's not a disadvantage but rather a big benefit that we have to evaluate all previous functions for any ϕ_k.

Algorithm 3 Evaluate basis functions $\phi_k(x)$ of semiclassical wavepackets

```
Require: A set of grid or quadrature nodes $x$
Require: A set $\Pi := \{P, Q, p, q\}$ of parameters
  // Base cases
  $\beta_0 := \pi^{-\frac{1}{4}}\varepsilon^{-\frac{1}{2}}\cdot\exp\left(\frac{i}{\varepsilon^2}\left(\frac{1}{2}PQ^{-1}(x-q)^2 + p(x-q)\right)\right)$
  $\beta_1 := Q^{-1}\sqrt{\frac{2}{\varepsilon^2}}(x-q)\cdot\beta_0$
  // Inductive steps
  for $k := 2$ to $K-1$ do
    $\beta_k := Q^{-1}\sqrt{\frac{2}{\varepsilon^2}}\frac{1}{\sqrt{k}}\cdot(x-q)\cdot\beta_{k-1} - Q^{-1}\overline{Q}\sqrt{\frac{k-1}{k}}\cdot\beta_{k-2}$
  end for
  return $B := (\beta_0, \ldots, \beta_{K-1})^{\mathrm{T}}$
```

In praxis we do not call algorithm 3 for each grid node $x_i \in \Gamma$ but use vectorization and calculate B for all nodes simultaneously.

3.2 Definition of scalar wavepackets

After we have defined a basis set $\{\phi_0, \phi_1, \ldots\}$ for the infinite dimensional Hilbert space of states we can now define the exact form of a state $|\Phi\rangle$. Each state is represented by a linear combination of the basis functions ϕ_k. Of course the basis functions are valid states too.

Assume we truncate the Hilbert space and use finite many basis functions. Let K be the maximal number of basis functions. Further let $S \in \mathbb{C}$ be a global phase. Any wavepacket can now be written as

$$|\Phi(x)\rangle := e^{\frac{iS}{\varepsilon^2}} \sum_{k=0}^{K-1} c_k \phi_k(x) \tag{3.14}$$

where $c_k \in \mathbb{C}$ are the coefficients of this linear combination. We can collect them in a vector $c := (c_0, \ldots, c_{K-1})^{\mathrm{T}}$. For later reference, we call the set

$$\Pi := \{P, Q, S, p, q\} \tag{3.15}$$

of variables the *Hagedorn parameters* of a wavepacket $|\Phi\rangle$ of form (3.14). This set includes the four parameters P, Q, p and q that come from the basis functions ϕ_k given by equation (3.9) as well as the global phase that enters the above linear combination. These values play an important role in the wavepacket based algorithms we'll discuss in the next two chapters.

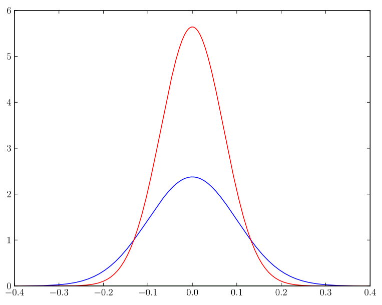
3.3 Definition of vector valued wavepackets

For the quantum dynamics with semiclassical wavepackets in the case of the vector valued Schrödinger equation as defined by formula (1.7) we need a wavepacket $|\Psi\rangle$ that is vector valued as well. Such a packet $|\Psi\rangle$ can be build as a vector of multiple scalar semiclassical packets. Assume that the Hamiltonian H in (1.7) is a $N \times N$ matrix, thus there are N energy levels and $|\Psi\rangle$ needs to have N components too. Formally we define

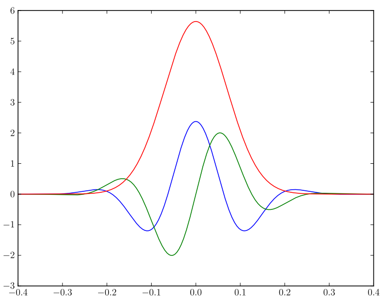
$$|\Psi\rangle := \left| \begin{pmatrix} \Phi_0(x) \\ \vdots \\ \Phi_{N-1}(x) \end{pmatrix} \right\rangle \tag{3.16}$$

where each of the Φ_i is of the form defined in (3.14).

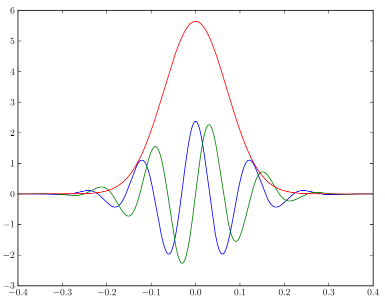
All Φ_i share the same parameters P, Q, global phase S, average momentum q and average position p. We will call a wavepacket that fulfils this condition a *homogeneous wavepacket*. Equivalently we can say the only thing that differs between the Φ_i is the vector of coefficients c. Therefore we add an index i to the notation, c^i stands for the coefficient vector of the component Φ_i. Thus a semiclassical wavepacket suitable for solving (1.7) has the important property that it is fully characterized by



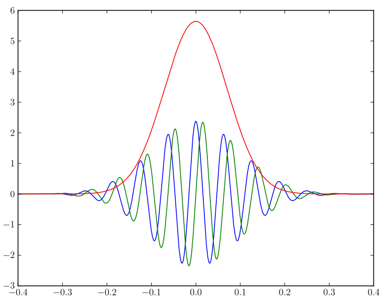
(a)



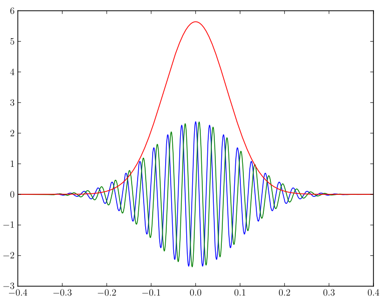
(b)



(c)



(d)



(e)

Figure 3.1: Hagedorn wavepackets Ψ with increasing momentum. The plots show the real (blue), imaginary (green) part and the absolute value (red) of $|\Psi\rangle$. (a) $q = 0.0$ and $p = 0.0$ (b) $q = 0.0$ and $p = 0.25$ (c) $q = 0.0$ and $p = 0.5$ (d) $q = 0.0$ and $p = 1.0$ (e) $q = 0.0$ and $p = 2.0$

- a single set Π of parameters P, Q, S, p and q.
- a vector c^i of coefficients for each component Φ_i.

3.4 Extended vector valued wavepackets

For advanced applications we may extend the definition (3.16) of a state $|\Psi\rangle$ and release the main restriction. In contrast to the homogeneous wavepackets of the last section we allow that each component Φ_i has it's very own set of parameters P, Q, S, p and q. We will call such a wavepacket an *inhomogeneous wavepacket*. To be able to distinguish the different variables, an index i is added also to the parameters. Thus a wavepacket is fully characterized by

- a set Π_i of parameters P_i, Q_i, S_i, p_i and q_i for each component Φ_i.
- a vector c^i of coefficients for each component Φ_i.

3.5 Numerical evaluation of wavepackets

The numerical evaluation of a wavepacket $|\Psi\rangle$ on given grid nodes x_i is not difficult. We just evaluate all the basis functions ϕ_k and assemble the parts. If we are in the case of (3.16) we can do this once and use the values for all N components of $|\Psi\rangle$. Otherwise we have to evaluate the basis functions individually for each component n as they differ by their Hagedorn parameters. Now we multiply these values with the coefficients c^n for each component. Finally we have to multiply with the phase exponential $e^{\left(\frac{iS}{\varepsilon^2}\right)}$. For an extended wavepacket we have to keep in mind that each component n has its own phase S^n. The algorithm 4 shows this procedure in the most general form. The outer for loop iterates over all components of $|\Psi\rangle$ while the inner loop is responsible for evaluating the basis functions ϕ_k^n with the given per component set of Hagedorn parameters. This part can be implemented efficiently according to algorithm 3.

Algorithm 4 Evaluate a vectorial wavepacket $|\Psi\rangle$ on a set of nodes

```
Require: A set of grid or quadrature nodes $x$
Require: An arbitrary (in)homogeneous wavepacket $\Psi$
  // Iterate over all components of $|\Psi\rangle$
  for $n = 0$ to $N - 1$ do
    given $\Pi_n$ as $\{P_n, Q_n, S_n, p_n, q_n\}$
    // Evaluate the basis for component $n$
    for $k = 0$ to $K - 1$ do
      $\beta_k := \phi_k \left[P_n, Q_n, p_n, q_n\right](x)$
    end for
    // Calculate the exponential of the phase
    $\pi_n := \exp\left(\frac{iS_n}{\varepsilon^2}\right)$
    // Assemble the component $\Phi_n$
    $\Phi_n := \pi_n \cdot \sum_{k=0}^{K} c_k^n \beta_k$
  end for
  return $(\Phi_0, \ldots, \Phi_{N-1})^{\mathrm{T}}$
```

Chapter 4

A wavepacket based algorithm

With the detailed insight from the last chapter we now move on and see how we can use semiclassical wavepackets to solve the vector valued Schrödinger equation as defined by (1.7). In this chapter we will study an algorithm that was presented in reference [3]. It uses the semiclassical wavepackets from the last chapter to simulate the time evolution of an initial wavepacket.

4.1 Splitting the Schrödinger equation

The linear time dependent Schrödinger equation as defined by (1.2) with a splitting of the Hamiltonian operator H into a kinetic and a potential part as in equation (1.3) can be written as two separate equations.

The first equation that contains only the kinetic operator T describes the time evolution of a free particle:

$$i\varepsilon^2 \frac{\partial \Psi}{\partial t} = -\frac{1}{2}\varepsilon^4 \frac{\partial^2}{\partial x^2} \Psi \,. \tag{4.1}$$

The other part contains only the potential $V(x)$ and reads:

$$i\varepsilon^2 \frac{\partial \Psi}{\partial t} = V(x)\,\Psi \,. \tag{4.2}$$

4.1.1 Splitting of the potential

The next step is to additionally decompose the potential. We split V into a local quadratic Taylor approximation $U(x)$ and the non-quadratic remainder $W(x)$

as:

$$V(x) = U(x) + W(x)\ . \tag{4.3}$$

This yields two new potential equations that replace (4.2), one with the quadratic part of the potential which, of course, describes an harmonic oscillator:

$$i\varepsilon^2 \frac{\partial \Psi}{\partial t} = U(x)\,\Psi\,. \tag{4.4}$$

The other equation contains all the bulky pieces that remain after the splitting:

$$i\varepsilon^2 \frac{\partial \Psi}{\partial t} = W(x)\,\Psi\,. \tag{4.5}$$

4.2 Propositions about exact propagation

The benefit of this two levels of decomposition becomes obvious once we recall that the aim is to propagate semiclassical wavepackets. It turns out that two out of these three parts can be solved exactly. In this section we closely follow reference [3].

We can solve the free particle equation (4.1) exactly. The important point is that if the wave function has the form of a semiclassical wavepacket defined by equation (3.9) then the coefficients c_i won't change during time propagation. For the time evolution during a single time step τ we get the following equations

$$\begin{aligned} q(t+\tau) &= q(t) + \tau p(t) \\ Q(t+\tau) &= Q(t) + \tau P(t) \\ S(t+\tau) &= S(t) + \frac{1}{2}\tau p(t)^{\mathrm{T}} p(t) \end{aligned} \tag{4.6}$$

with p and P being constant.

In a similar way we can solve the quadratic potential equation (4.4) exactly without changing the packet's coefficients c_i. The time evolution this time reads

$$\begin{aligned} p(t+\tau) &= p(t) - \tau \nabla U(q(t)) \\ P(t+\tau) &= P(t) - \tau \nabla^2 U(q(t))\, Q(t) \\ S(t+\tau) &= S(t) - \tau U(q(t)) \end{aligned} \tag{4.7}$$

where q and Q stay the same.

During these two parts we only change the parameters P, Q and the phase S besides the position q and momentum p of the wavepacket but we never touch the coefficients c_i.

In the third step which deals with equation (4.5) however, we can no longer keep the coefficients c_i constant. But in this turn, we can fix the five parameters and solely update the coefficients. The starting point is the following variational approximation

$$\langle \phi_k, i\varepsilon^2 \frac{\partial u}{\partial t} - Wu \rangle = 0 \quad \forall k \tag{4.8}$$

on the Hilbert space $\mathcal{M}$ defined by

$$\mathcal{M} := \{ v \in L^2\left(R^d\right) : v\left(x\right) = \sum_{k=0}^{K-1} c_k \phi_k\left(x\right) \} . \tag{4.9}$$

This is the space spanned by all functions defined by (3.9) with identical and fixed parameters P, Q, p and q. Equivalently to (4.8) we can solve the following system of linear ordinary differential equations

$$i\varepsilon^2 \frac{dc_k}{dt} = \sum_{l=0}^{K-1} f_{k,l} c_l \tag{4.10}$$

where $f_{k,l} := \langle \phi_k \,|\, W \,|\, \phi_l \rangle$ and we collect all these f into a Hermitian matrix F which is the matrix representation of the non-quadratic remainder W over the basis $\{\phi_i\}_i$. The solution to (4.10) is then simply given by

$$c\left(t+\tau\right) = \exp\left(-\frac{i\tau}{\varepsilon^2} F\right) c\left(t\right) \tag{4.11}$$

and describes the time evolution of the coefficients in the potential's remainder.

For the exact details and the proofs about this splitting and the separate time evolutions we refer to section 2 of [3].

4.2.1 The matrix representation

One last step remains before we can put together the algorithm. In the last paragraph above we need the matrix denoted by F. The definition gives us this equation:

$$f_{k,l} := \langle \phi_k \,|\, W \,|\, \phi_l \rangle = \int_{\mathbb{R}^d} \overline{\phi_k\left(x\right)} W\left(x\right) \phi_l\left(x\right) dx \tag{4.12}$$

which looks like any integration problem. We now want to find an efficient way to calculate this integral for all k and l or, in other words, set up the matrix F. The integral containing W can almost never be solved analytically thus the integral is approximated by quadrature. Because we will need the results at different places, let's look at a much broader setup of this problem.

4.3 Analytical integration and quadrature

Suppose we have the wave function $|\Phi_i\rangle$ of a component as defined by (3.14). Recall then for the basis functions we have orthogonality: $\langle \phi_m \,|\, \phi_n \rangle = \delta_{m,n}$. The two kets $|\Phi_i\rangle$ and $|\Phi_j\rangle$ must have the same Hagedorn parameters for the time being, but of course they can have different coefficients. Thus we add upper indices to the coefficients. For a general but sufficiently smooth function

$$\begin{aligned} f : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto f(x) \end{aligned}$$

we want to transform the following expression into an integral

$$\begin{aligned} \langle \Phi_i \,|\, f \,|\, \Phi_j \rangle &= \left\langle e^{\frac{iS}{\varepsilon^2}} \sum_k c_k^i \phi_k \,\middle|\, f \,\middle|\, e^{\frac{iS}{\varepsilon^2}} \sum_k c_k^j \phi_k \right\rangle \\ &= \underbrace{e^{\frac{-iS}{\varepsilon^2}} e^{\frac{iS}{\varepsilon^2}}}_{=1} \left\langle \sum_k c_k^i \phi_k \,\middle|\, f \,\middle|\, \sum_k c_k^j \phi_k \right\rangle \\ &= \sum_{k,l} \overline{c_k^i} c_l^j \, \langle \phi_k \,|\, f \,|\, \phi_l \rangle \\ &= \sum_{k,l} \overline{c_k^i} c_l^j \int_{\mathbb{R}} \overline{\phi_k(x)} f(x) \, \phi_l(x) \, dx \end{aligned}$$

where we exploited the fact that the complex inner product is sesquilinear and the global phase cancels out.

To deal with the numerics, the integral is approximated by a high order Gauss-Hermite quadrature with weights ω_r and nodes γ_r.

$$\int_{\mathbb{R}} \overline{\phi_k(x)} f(x) \, \phi_l(x) \, dx \approx \sum_r f(\gamma_r) \, \overline{\phi_k(\gamma_r)} \phi_l(\gamma_r) \, \omega_r \tag{4.13}$$

If we now put all the parts together then the whole formula becomes

$$\langle \Phi_i \,|\, f \,|\, \Phi_j \rangle = \sum_{k,l} \overline{c_k^i} c_l^j \sum_r f(\gamma_r) \, \overline{\phi_k(\gamma_r)} \phi_l(\gamma_r) \, \omega_r \, . \tag{4.14}$$

A straight forward implementation to calculate these integrals could look like the code shown in snippet 5. This is very inefficient, we can do much better and replace two of the loops by implicit vectorized calculations.

Algorithm 5 Inefficient version of the quadrature of $I := \langle \Phi_i \,|\, f \,|\, \Phi_j \rangle$

```
Require: Quadrature rule with $\rho$ pairs $(\gamma_r, \omega_r)$.
  $I := 0$
  for $k = 0$ to $K-1$ do
    for $l = 0$ to $K-1$ do
      $I_{k,l} := 0$
      // Iterate over all pairs $(\gamma_r, \omega_r)$
      for $r = 0$ to $\rho - 1$ do
        $I_{k,l} := I_{k,l} + f(\gamma_r)\,\overline{\phi_k(\gamma_r)}\phi_l(\gamma_r)\,\omega_r$
      end for
      // Multiply with the prefactors
      $I := I + \overline{c^i_k} c^j_l \cdot I_{k,l}$
    end for
  end for
  return $I$
```

Suppose all K basis functions are collected in a vector $\varphi := (\phi_0, \ldots, \phi_{K-1})^{\mathrm{T}}$ and the coefficients in the same manner $c := (c_0, \ldots, c_{K-1})^{\mathrm{T}}$. While the vector φ is identical for both $|\Phi_i\rangle$ and $|\Phi_j\rangle$, the vector c is different and we write c^i and c^j respectively. With this data structure we get

$$\begin{aligned}\langle \Phi_i \,|\, f \,|\, \Phi_j \rangle &= \int {\Phi_i}^{\mathrm{H}} f \Phi_j dx = \int \left(c^{i\mathrm{T}} \varphi \right)^{\mathrm{H}} f c^{j\mathrm{T}} \varphi dx \\ &= \int \varphi^{\mathrm{H}} \overline{c^i} f c^{j\mathrm{T}} \varphi dx = \int c^{i\mathrm{H}} \overline{\varphi} f \varphi^{\mathrm{T}} c^j dx \\ &= c^{i\mathrm{H}} \left(\int \overline{\varphi} f \varphi^{\mathrm{T}} dx \right) c^j = c^{i\mathrm{H}} \left(\int f \overline{\varphi} \varphi^{\mathrm{T}} dx \right) c^j \,.\end{aligned} \tag{4.15}$$

Notice that $\overline{\varphi}\varphi^{\mathrm{T}}$ is a $K \times K$ matrix. Further as f is independent of k and l and scalar we can factor it out

$$\widetilde{F}(x) := f \overline{\varphi} \varphi^{\mathrm{T}} = f(x) \begin{pmatrix} & \vdots & \\ \ldots & \overline{\phi_k(x)} \phi_l(x) & \ldots \\ & \vdots & \end{pmatrix} \tag{4.16}$$

so the integral is essentially matrix valued. Let's note again this major result

$$\langle \Phi_i \,|\, f \,|\, \Phi_j \rangle = c^{i\mathrm{H}} \underbrace{\left(\int \widetilde{F}(x)\, dx \right)}_{F} c^j \tag{4.17}$$

If the expression above is approximated by the quadrature it becomes

$$\langle \Phi_i \,|\, f \,|\, \Phi_j \rangle \approx c^{i\mathrm{H}} \left(\sum_r^{\rho} \omega_r \widetilde{F}(\gamma_r) \right) c^j \tag{4.18}$$

with the sum having matrices as summands. This is precisely the way an efficient implementation works. Even if we construct all the matrices $\widetilde{F}(\gamma_i)$ this is not too expensive as each one is just a rank one matrix and the vectors $\varphi(\gamma_i)$ are available already in a vectorized data structure.

4.3.1 Building the matrix

So far we only considered the quadrature of $\langle \Phi \,|\, f \,|\, \Phi \rangle$. But there is an implicit connection to our question on how to build the matrix F. It just drops out as a byproduct of the above improved formulae for integration!

This becomes very obvious once we write $\varphi := (|\phi_0\rangle, \ldots, |\phi_{K-1}\rangle)^{\mathrm{T}}$. If we now plug this into $\overline{\varphi} f \varphi^{\mathrm{T}}$ from above we get

$$\begin{pmatrix} \langle \phi_0 \,|\, f \,|\, \phi_0 \rangle & \ldots & \langle \phi_0 \,|\, f \,|\, \phi_{K-1} \rangle \\ \vdots & & \vdots \\ \langle \phi_{K-1} \,|\, f \,|\, \phi_0 \rangle & \ldots & \langle \phi_{K-1} \,|\, f \,|\, \phi_{K-1} \rangle \end{pmatrix} =: F \tag{4.19}$$

which is the matrix F we wanted. We only need to replace $f(x)$ by the non-quadratic remainder $W(x)$ to get the matrix needed in (4.11).

The procedure 6 shows an implementation for constructing the matrix F. In the step where we retrieve the basis evaluated on the quadrature nodes γ we use algorithm 3. The evaluation of f on all nodes does not require a for loop because f itself is implemented in a way that allows vectorized data processing[1].

Based on algorithm 6 the procedure 7 shows a possible efficient implementation for the quadrature shown in (4.18).

4.3.2 Quadrature in general

In the quadrature introduced in the last section we always assumed that there is a quadrature rule with suitable nodes and weights. But we never specified how to get the nodes. The quadrature rule is a high order Gauss-Hermite quadrature

[1] For vectorized evaluation of a function $f(x)$ the following identity holds

$$f((x_0, \ldots, x_n)) \equiv (f(x_0), \ldots, f(x_n)) \tag{4.20}$$

Algorithm 6 Build the matrix $F := (\langle \phi_i \,|\, f \,|\, \phi_j \rangle)_{i,j}$

```
Require: Quadrature rule with $L$ pairs $(\gamma_l, \omega_l)$.
Require: The scalar function $f(x)$
  // Evaluate the function $f$ for all quadrature nodes $\gamma$
  $(v_0, \ldots, v_{L-1}) := f((\gamma_0, \ldots, \gamma_{L-1}))$
  // Evaluate the basis functions for all quadrature nodes $\gamma$
  $B := (\beta_0, \ldots, \beta_{K-1})$
  // Set up a zero matrix
  $F \in \mathbb{R}^{K \times K}, \quad F := \mathbf{0}$
  // Iterate over all pairs $(\gamma_l, \omega_l)$
  for $l = 0$ to $L - 1$ do
    $F := F + v_l \varepsilon \cdot B^{\mathrm{H}} B \cdot \omega_l$
  end for
  return $F$
```

Algorithm 7 Efficient version of the quadrature of $I := \langle \Phi_i \,|\, f \,|\, \Phi_j \rangle$

```
Require: Quadrature rule with $L$ pairs $(\gamma_l, \omega_l)$.
Require: The scalar function $f(x)$
  Build the matrix $F$ by algorithm 6
  // Multiply by the coefficients
  $I := c^{i\mathrm{H}} F c^j$
  return $I$
```

which is build to exactly integrate expressions of the form

$$\int_{\mathbb{R}} \underbrace{e^{-x^2}}_{w(x)} f(x)\, dx \approx \sum_{i=0}^{n} \omega_i f(\gamma_i) \tag{4.21}$$

The nodes γ_i are the roots of the Hermite polynomial $H_n(x)$ and the weights ω_i are given by

$$\omega_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 H_{n-1}(x_i)^2} \,. \tag{4.22}$$

For the details see [1]. Of course we can never calculate the quadrature nodes this way. The roots of a polynomial are very ill-conditioned even for medium degrees. Hence we need another more stable way for finding the quadrature nodes. The Golub-Welsch algorithm that is superior to the above formula calculates the nodes from the eigenvalues of an orthogonal matrix. This computation is numerically stable also for quadratures of high order that need many nodes. For more details on how this algorithm works see reference [4].

The next problem we face is that we are only implicitly in the case of (4.21). The integrand we want to integrate has the form $g := e^{-x^2} f(x)$, but we only know g as a whole and can not separate the exponential from f as required for

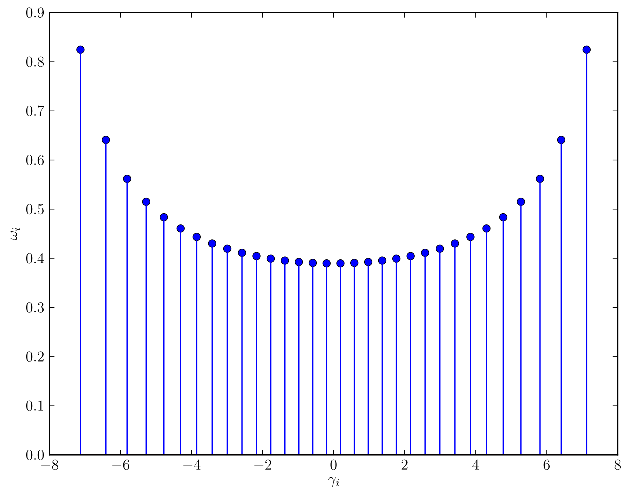


Figure 4.1: Example of transformed Qauss-Hermite quadrature weights.

applying the quadrature rule. Thus our integral looks like the right hand side of (4.23).

$$\int_{\mathbb{R}} e^{-x^2} f(x)\, dx = \int_{\mathbb{R}} g(x)\, dx \tag{4.23}$$

We can never calculate e^{-x^2} explicitly nor divide this factor out. This would result in major numerical issues.

A possible work around is to use Hermite functions[2] $h_n(x)$ and modify the weights ω to fit our purpose. Thus changing the weights like:

$$\omega'_i := \frac{1}{h_n(\gamma_i)^2\, n}\,. \tag{4.25}$$

Figure 4.1 shows an example of all pairs (γ_i, ω'_i) for a given quadrature rule of order $n = 32$.

However, the nodes still deserve our attention. As we see in the figure 4.1 the nodes are centred around 0. This is correct for integrating with a weighting function $w(x) = e^{-x^2}$. But the Gaussian exponential present in our wavepacket's basis functions given by (3.3) is shifted away from 0 by an amount q. Hence we

[2]We use the following definition for the Hermite functions:

$$h_k(\xi) := \frac{1}{\sqrt{\left(2^k k! \sqrt{\pi}\right)}} e^{-\frac{\xi^2}{2}} H_k(\xi) \tag{4.24}$$

of degree k.

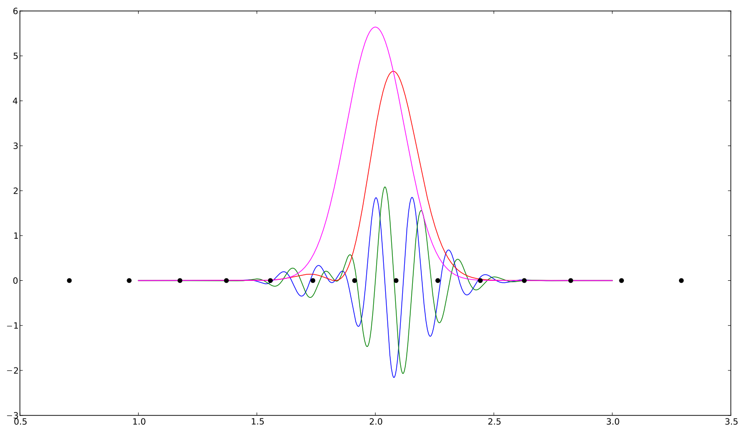


Figure 4.2: Example of a quadrature for a given wavepacket Ψ with. Plotted is the real (blue) and imaginary (green) part as well as the absolute value (red) of the wavepacket. The black dots are the quadrature nodes. The magenta curve shows the Gaussian we get for a wavepacket with the same Hagedorn parameters Π but $c_0 = 1$ and $c_{k>0} = 0$.

have to shift and spread the quadrature nodes as well:

$$\gamma_i' := q + \varepsilon |Q| \gamma_i \,. \tag{4.26}$$

The figure 4.2 shows an example of a quadrature for an arbitrary homogeneous wavepacket $|\Psi\rangle$ with parameters $\Pi = (i, 3, 0, 0.4, 2)$ and coefficients $c_0 = 0.25$, $c_1 = 0.3$ and $c_4 = 0.1$.

Anytime we need quadrature nodes and weights, we'll use the pairs (γ_i', ω_i'). After this lengthy section about integration we return to the propagation algorithm.

4.4 The original time propagation algorithm

Let us now review the time propagation algorithm briefly. The algorithm as defined in section 3.3 of reference [3] is constructed for the propagation of semiclassical wavepackets as defined by (3.14) in an arbitrary number of space dimensions. It integrates the three steps (4.6), (4.7) and (4.11) and combines them into a propagation algorithm suitable for general potentials.

Given the parameters $P^{(j)}$, $Q^{(j)}$, the phase $S^{(j)}$ and the position $q^{(j)}$ and momentum $p^{(j)}$ of the state $|\Phi\rangle$ at time $t^{(j)} := j\tau$ then the algorithm 8 will compute the same values one time step τ later. We omitted the mass matrix M here that is present in ref. [3]. The whole algorithm essentially only propagates the Hagedorn parameters and the coefficients in time.

Algorithm 8 Time propagation of scalar wavepackets $|\Phi\rangle$

```
Require: A semiclassical wavepacket $|\Phi(t)\rangle$ with its Hagedorn parameters
  // Propagate with the kinetic operator
  $q^{\left(j+\frac{1}{2}\right)} := q^{(j)} + \frac{\tau}{2} p^{(j)}$
  $Q^{\left(j+\frac{1}{2}\right)} := Q^{(j)} + \frac{\tau}{2} P^{(j)}$
  $S^{\left(j+\frac{1}{2},-\right)} := S^{(j)} + \frac{\tau}{4} p^{(j)\mathrm{T}} p^{(j)}$
  // Propagate with the local quadratic potential
  $p^{(j+1)} := p^{(j)} - \tau \nabla V \left(q^{(j+1/2)}\right)$
  $P^{(j+1)} := P^{(j)} - \tau \nabla^2 V \left(q^{(j+1/2)}\right) Q^{(j+1/2)}$
  $S^{(j+1/2,+)} := S^{(j+1/2,-)} - \tau V \left(q^{(j+1/2)}\right)$
  // Propagate with the non-quadratic remainder
  // Assemble the matrix $F$
  $F_{k,l} := \langle \phi_k \,|\, W(x) \,|\, \phi_l \rangle \quad \forall k,l \in 0,\ldots,K-1$
  // And propagate the coefficients
  $c^{(j+1)} := \exp\left(-\tau \frac{i}{\varepsilon^2} F^{(j+1/2)}\right) c^{(j)}$
  // Propagate with the kinetic operator again
  $q^{(j+1)} := q^{(j+1/2)} + \frac{\tau}{2} p^{(j+1)}$
  $Q^{(j+1)} := Q^{(j+1/2)} + \frac{\tau}{2} P^{(j+1)}$
  $S^{(j+1)} := S^{(j+1/2,+)} + \frac{\tau}{4} p^{(j+1)\mathrm{T}} p^{(j+1)}$
  return $|\Phi(t+\tau)\rangle$
```

4.5 The propagation algorithm for vector valued wavepackets

In this section we discuss what parts have to be altered in the algorithm 8 such that we can propagate the vector valued wavepackets given by (3.16).

What changes when we plug in the vector $|\Psi\rangle$? First of all, the potential V is now a matrix and we have to define carefully what we mean e.g. by ∇V. The vector of wavepackets defined by (3.16) has N states. Thus we have N vectors $c^0, \ldots, c^{N-1}$ each containing the coefficients c_k^n of the component n. On the other hand we have just a single set of Hagedorn parameters P, Q, S, p and q. From this first glance we can identify the two core points. Who is responsible for propagating the parameters and how do we have to modify the matrix F to fully reflect the existence of multiple coefficient vectors.

4.5.1 Splitting of the potential matrix

First we choose a so called *leading component index* $\chi \in \{0, \ldots, N-1\}$. Then the energy level λ_χ is the one responsible for propagating the Hagedorn parameters of our wavepacket. Thus this energy level $\lambda\chi(x)$ takes over the role of the scalar potential function $V(x)$ from section 4.4.

We split the potential eigenvalue $\lambda\chi(x)$ into a local quadratic part u_χ and a non-quadratic remainder w_χ. This is done via a simple Taylor series expansion up to second order around a given point q.

$$\begin{aligned} u_\chi(x) &= \lambda_\chi(q) + \nabla\lambda_\chi(q)(x-q) + \frac{1}{2}(x-q)^{\mathrm{T}}\nabla^2\lambda_\chi(q)(x-q) \\ w_\chi(x) &= \lambda_\chi(x) - u_\chi(x)\ . \end{aligned} \tag{4.27}$$

For a potential in one space dimension that therefore depends only on one single variable, say x, this reduces to

$$\begin{aligned} u_\chi(x) &= \lambda_\chi|_{x=q} + \frac{d}{dx}\lambda_\chi|_{x=q}(x-q) + \frac{1}{2}\frac{d^2}{dx^2}\lambda_\chi|_{x=q}(x-q)^2 \\ w_\chi(x) &= \lambda_\chi(x) - u_\chi(x)\ . \end{aligned} \tag{4.28}$$

We can now write the potential matrix as a pure quadratic diagonal part U plus a non-quadratic remainder matrix W:

$$V = \begin{pmatrix} u_\chi & & 0 \\ & \ddots & \\ 0 & & u_\chi \end{pmatrix} + \begin{pmatrix} v_{0,0} - u_\chi & \cdots & v_{0,N-1} \\ \vdots & \ddots & \vdots \\ v_{N-1,0} & \cdots & v_{N-1,N-1} - u_\chi \end{pmatrix}. \tag{4.29}$$

The first part U is not used explicitly in the propagation algorithm. We just propagate one set of Hagedorn parameters and this can be done with u_χ solely. But the non-quadratic part W is used for altering and mixing the coefficients c^i of all components Φ_i of $|\Psi\rangle$. This is the answer to the first of the questions posed above.

4.5.2 Extending the handling of coefficients

For finding an answer to the second question too, we elaborate on how to build the matrix F this time. To begin with we construct a new data structure for the coefficients. As said above, our $|\Psi\rangle$ consists of N components each of them having a vector c^i with the K coefficients corresponding to Φ_i. We can stack all these column vectors and build a block vector C with all coefficients of all components inside

$$C := \left(\ c_0^0, \ldots, c_{K-1}^0 \ \middle|\ \cdots \ \middle|\ c_0^{N-1}, \ldots, c_{K-1}^{N-1}\ \right)^{\mathrm{T}}. \tag{4.30}$$

This vector has NK entries. A compatible matrix $\mathbf{F}$ of size $NK \times NK$ is now needed. The matrix is in block form too and we can even calculate all the $K \times K$

blocks individually.

$$\mathbf{F} := \left(\begin{array}{c|c|c} F_{0,0} & \dots & F_{0,N-1} \\ \hline \vdots & & \vdots \\ \hline F_{N-1,0} & \dots & F_{N-1,N-1} \end{array}\right) . \tag{4.31}$$

Each of the $F_{i,j}$ is nothing else than the matrix F known from (4.12). But we have to be careful with the middle part of the braket. This time, our W from (4.29) is a matrix thus we can't simply build $\mathbf{F}$ from identical copies of F even if the basis function ϕ_k are the same for all $F_{i,j}$. Instead we have to distribute the individual entries $W_{i,j}$ of W across the blocks of $\mathbf{F}$. Thus each block $F_{i,j}$ of (4.31) is given by $\int_{\mathbb{R}} \widetilde{F}_{i,j}(x)\, dx$ where

$$\widetilde{F}_{i,j}(x) := W_{i,j}(x) \begin{pmatrix} & \vdots & \\ \dots & \overline{\phi_k(x)}\phi_l(x) & \dots \\ & \vdots & \end{pmatrix} . \tag{4.32}$$

For all the individual blocks we could use algorithm 6, fed with the appropriate entry $W_{i,j}$ of the non-quadratic remainder matrix W, to build the submatrices $F_{i,j}$. A basic explicit implementation of these formulae is given in the algorithm 9 below.

4.5.3 Pseudo code for the time propagation

Now we are ready to write a new code which is able to propagate $|\Psi\rangle$. This algorithm is a generalization of the time propagation given in algorithm 8. The core concepts stay the same, but the details are a little bit more complex as we saw in the last sections.

4.6 Basis transformations

For some calculations we should be able to transform the wavepacket from the canonical basis to the eigenbasis of the given potential V and vice versa. For example we set the initial values in the potentials' eigenbasis but perform the simulation in the canonical basis. So we now investigate how such a basis transformation works.

Given a wavepacket as usual in the form of (3.16). Further assume the matrix $M \in \mathbb{R}^{N\times N}$ to be the matrix that diagonalizes the potential according to (1.9).

Algorithm 9 Build the homogeneous block matrix $\mathbf{F} := (F_{r,c})_{r,c}$

```
Require: A homogeneous wavepacket $\Psi$
Require: $W$ a $N \times N$ matrix of scalar functions
  // Initialize $\mathbf{F}$ as the zero-matrix
  $\mathbf{F} \in \mathbb{R}^{NK \times NK}, \quad \mathbf{F} := \mathbf{0}$
  // Evaluate the basis functions for all quadrature nodes $\gamma$ with algorithm 3
  given $\Pi$ as $\{P, Q, S, p, q\}$
  $B := (\beta_0, \ldots, \beta_{K-1})$
  // Iterate over all row and column blocks of this matrix
  for $r = 0$ to $N - 1$ do
    for $c = 0$ to $N - 1$ do
      // Evaluate the function $W_{r,c}$ for all quadrature nodes $\gamma$
      $(v_0, \ldots, v_{L-1}) := W_{r,c}((\gamma_0, \ldots, \gamma_{L-1}))$
      // Set up a zero matrix
      $F \in \mathbb{R}^{K \times K}, \quad F := \mathbf{0}$
      // Iterate over all pairs $(\gamma_l, \omega_l)$
      for $l = 0$ to $L - 1$ do
        $F := F + v_l \varepsilon \cdot B^{\mathrm{H}} B \cdot \omega_l$
      end for
      // Insert the block $F$ into the block matrix $\mathbf{F}$
      $\mathbf{F}_{r,c} := F$
    end for
  end for
  return $\mathbf{F}$
```

The general form of (1.10) can be written with all the entries as

$$M := \begin{pmatrix} m_{0,0} & \cdots & m_{0,N-1} \\ \vdots & & \vdots \\ m_{N-1,0} & \cdots & m_{N-1,N-1} \end{pmatrix} . \tag{4.33}$$

In the following we try to transform $|\Psi\rangle$ to the eigenbasis. We start with calculating $M|\Psi\rangle$, the action of M on Ψ.

$$\begin{aligned} |\Psi'\rangle &= M|\Psi\rangle \\ &= \left. \begin{pmatrix} m_{0,0} & \cdots & m_{0,N-1} \\ \vdots & & \vdots \\ m_{N-1,0} & \cdots & m_{N-1,N-1} \end{pmatrix} \middle| \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \right\rangle \\ &= \left| \begin{pmatrix} m_{0,0}\Phi_0 + & \cdots & + m_{0,N-1}\Phi_{N-1} \\ & \vdots & \\ m_{N-1,0}\Phi_0 + & \cdots & + m_{N-1,N-1}\Phi_{N-1} \end{pmatrix} \right\rangle . \end{aligned} \tag{4.34}$$

Algorithm 10 Time propagation of a homogeneous wavepacket $|\Psi\rangle$

```
Require: A semiclassical wavepacket $|\Psi(t)\rangle$
Require: The set $\Pi$ of Hagedorn parameters of $\Psi$
  // Propagate with the kinetic operator
  $q^{\left(j+\frac{1}{2}\right)} := q^{(j)} + \frac{\tau}{2} p^{(j)}$
  $Q^{\left(j+\frac{1}{2}\right)} := Q^{(j)} + \frac{\tau}{2} P^{(j)}$
  $S^{\left(j+\frac{1}{2},-\right)} := S^{(j)} + \frac{\tau}{4} p^{(j)\mathrm{T}} p^{(j)}$
  // Propagate with the local quadratic potential
  $p^{(j+1)} := p^{(j)} - \tau \nabla \lambda_\chi \left(q^{(j+1/2)}\right)$
  $P^{(j+1)} := P^{(j)} - \tau \nabla^2 \lambda_\chi \left(q^{(j+1/2)}\right) Q^{(j+1/2)}$
  $S^{(j+1/2,+)} := S^{(j+1/2,-)} - \tau \lambda_\chi \left(q^{(j+1/2)}\right)$
  // Propagate with the non-quadratic remainder
  // Stack the coefficient vectors $c^n$ of all components
  $C^{(j)} := \left(c^0, \ldots, c^{N-1}\right)^{\mathrm{T}}$
  // Assemble the block matrix $\mathbf{F}$ using algorithm 9
  $\mathbf{F}^{(j+1/2)} := \left(F_{r,c}\right)_{r,c} \quad \forall r, c \in 0, \ldots, N-1$
  // Propagate the coefficients
  $C^{(j+1)} := \exp\left(-\tau \frac{i}{\varepsilon^2} \mathbf{F}^{(j+1/2)}\right) C^{(j)}$
  // Split the coefficients
  $\left(c^0, \ldots, c^{N-1}\right) := C^{(j+1)}$
  // Propagate with the kinetic operator again
  $q^{(j+1)} := q^{(j+1/2)} + \frac{\tau}{2} p^{(j+1)}$
  $Q^{(j+1)} := Q^{(j+1/2)} + \frac{\tau}{2} P^{(j+1)}$
  $S^{(j+1)} := S^{(j+1/2,+)} + \frac{\tau}{4} p^{(j+1)\mathrm{T}} p^{(j+1)}$
  return $|\Psi(t+\tau)\rangle$
```

For the next steps we just consider the j^{th} component of this last expression.

$$\begin{aligned} m_{j,0}\Phi_0 + \cdots + m_{j,N-1}\Phi_{N-1} &= m_{j,0} e^{\frac{iS}{\varepsilon^2}} \sum_{k_0} c^0_{k_0} \phi_{k_0} + \cdots + m_{j,N-1} e^{\frac{iS}{\varepsilon^2}} \sum_{k_{N-1}} c^{N-1}_{k_{N-1}} \phi_{k_{N-1}} \\ &= e^{\frac{iS}{\varepsilon^2}} \left(\sum_{k_0} m_{j,0} c^0_{k_0} \phi_{k_0} + \cdots + \sum_{k_{N-1}} m_{j,N-1} c^{N-1}_{k_{N-1}} \phi_{k_{N-1}} \right) \\ &= e^{\frac{iS}{\varepsilon^2}} \sum_k \left(m_{j,0} c^0_k + \cdots + m_{j,N-1} c^{N-1}_k \right) \phi_k \\ &= e^{\frac{iS}{\varepsilon^2}} \sum_k \underbrace{\sum_l^{N-1} m_{j,l} c^l_k}_{c'_k} \phi_k =: e^{\frac{iS}{\varepsilon^2}} \Phi'_j \, . \end{aligned} \tag{4.35}$$

From the last line we see that we can represent the components of the transformed wavepacket $|\Psi'\rangle$ again in Hagedorn form with unchanged basis functions $\phi_i(x)$. However, this is not enough to transform the wavepacket to the potential's eigenbasis as we missed the point that all $m_{i,j}$ depend on x! Hence we additionally need to project the above result to the subspace spanned by the basis functions $\phi_i(x)$. This is done as usual with the inner product.

Denote the coefficients of the j^{th} component of the final wavepacket with d^j_i. Then we can write:

$$\begin{aligned} d^j_p &= \left\langle \phi_p \,\middle|\, \Phi'_j \right\rangle \\ &= \left\langle \phi_p(x) \,\middle|\, \sum_k \sum_l^{N-1} m_{j,l}(x)\, c^l_k \phi_k(x) \right\rangle \\ d^j_p &= \sum_k \sum_l^{N-1} c^l_k \left\langle \phi_p(x) \,\middle|\, m_{j,l}(x)\, \phi_k(x) \right\rangle \, . \end{aligned} \tag{4.36}$$

The part in the braket is just another inner product and, from the definition, we write it as an integral:

$$\left\langle \phi_p(x) \,\middle|\, m_{j,l}(x)\, \phi_k(x) \right\rangle = \int_{\mathbb{R}} \overline{\phi_p(x)} m_{j,l}(x)\, \phi_k(x)\, dx \, . \tag{4.37}$$

We will calculate this integral by means of quadrature again. A straight forward implementation that calculates d^j_p for all $p \in 0 \ldots K-1$ and $j \in 0 \ldots N-1$ could be given by the code snippet (11). Here we assume a basis size of K.

It would be cumbersome and very inefficient to implement the calculation this way. Thus let's try to reformulate the problem as a matrix multiplication

Algorithm 11 Simple version of the basis transformation integral for $|\Psi\rangle$

```
for $j = 0$ to $N-1$ do
  for $p = 0$ to $K-1$ do
    $d_p^j := 0$
    for $k = 0$ to $K-1$ do
      for $l = 0$ to $N-1$ do
        $d_p^j := d_p^j + c_k^l \int_{\mathbb{R}} \overline{\phi_p(x)} m_{j,l}(x) \phi_k(x)\, dx$
      end for
    end for
  end for
end for
```

such that we can perform it efficiently. First, we can interchange the order of summation and notice that m depends on j and l but (of course) not on p and k. This allows us to calculate the vector of the coefficients d^j for fixed j as a sum over matrix multiplications.

$$\begin{pmatrix} \vdots \\ d_p^j \\ \vdots \end{pmatrix} = \sum_{l}^{N-1} \begin{pmatrix} & \vdots & \\ \dots & \langle \varphi_p \,|\, m_{j,l} \varphi_k \rangle & \dots \\ & \vdots & \end{pmatrix} \begin{pmatrix} \vdots \\ c_l \\ \vdots \end{pmatrix} \qquad j = 1 \dots N-1 \,. \tag{4.38}$$

We can do much better and assemble a big matrix to perform the calculations for all the j components of $|\Psi^e\rangle$ simultaneously.

$$\left(\begin{array}{c} d^0 \\ \hline \vdots \\ \hline d^{N-1} \end{array}\right) = \left(\begin{array}{c|c|c} \langle \varphi \,|\, m_{0,0} \,|\, \varphi \rangle & \dots & \langle \varphi \,|\, m_{0,N-1} \,|\, \varphi \rangle \\ \hline \vdots & & \vdots \\ \hline \langle \varphi \,|\, m_{N-1,0} \,|\, \varphi \rangle & \dots & \langle \varphi \,|\, m_{N-1,N-1} \,|\, \varphi \rangle \end{array}\right) \left(\begin{array}{c} c^0 \\ \hline \vdots \\ \hline c^{N-1} \end{array}\right) \tag{4.39}$$

The block matrix, let's call it $\mathbf{F}$, is of size $NK \times NK$ with N^2 individual blocks of size $K \times K$. The vectors φ consist of all the K basis functions. Because the wavepacket is homogeneous all the φ are equivalent thus in fact we have only a single φ.

Finally we see that the transformation to the eigenbasis is nothing else than multiplication with a big matrix where each matrix element is an integral which can easily be carried out by quadrature. We already met matrices of this structure in section 4.5.2 hence we can use the algorithm 9 to construct our matrix $\mathbf{F}$. We only need to call the algorithm with the argument M instead of W.

4.7 Norm calculation for wavepackets

This section deals with the calculation of norms. Suppose we have got a wavepacket as defined by (3.14) and now we want to compute the norm $\|\Phi\|^2$ of this wavepacket. From the definition we derive:

$$\begin{aligned}
\|\Phi\|^2 = \langle \Phi \mid \Phi \rangle &= \left\langle e^{\frac{iS}{\varepsilon^2}} \sum_k c_k \phi_k \,\middle|\, e^{\frac{iS}{\varepsilon^2}} \sum_l c_l \phi_l \right\rangle \\
&= \left\langle \sum_k c_k \phi_k \,\middle|\, \sum_l c_l \phi_l \right\rangle \\
&= \sum_k \overline{c_k} \sum_l c_l \underbrace{\langle \phi_k \mid \phi_l \rangle}_{=\delta_{k,l}} \\
&= \sum_{k,l} \overline{c_k} c_l \delta_{k,l} = \sum_k \overline{c_k} c_k \\
&= \|c\|^2 .
\end{aligned} \tag{4.40}$$

Remember that $\langle \cdot \mid \cdot \rangle$ is sesquilinear. Therefore the global phase cancels out. Additionally we used orthogonality of the basis functions ϕ_i.

4.7.1 Norm calculation for vectorial wavepackets

Now we do the same calculation but for homogeneous vector valued wavepackets of the form defined by (3.16). This case can easily be reduced to the previous one. Again we start from the basic definition of the norm.

$$\begin{aligned}
\|\Psi\|^2 = \langle \Psi \mid \Psi \rangle &= \left\langle \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \,\middle|\, \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \right\rangle \\
&= \sum_i^{N-1} \langle \Phi_i \mid \Phi_i \rangle = \sum_i^{N-1} \|\Phi_i\|^2 \\
&= \sum_i^{N-1} \|c^i\|^2 .
\end{aligned} \tag{4.41}$$

From the last line we see that the norm is nothing else than the sum of the squared norms of each component. This makes the computation as well as the implementation trivial.

4.8 Potential and kinetic energies

In this section we want to have a closer look at the different energies and the calculations thereof.

4.8.1 Potential energy

The potential energy of a wavepacket $|\Psi\rangle$ that feels the potential $V(x)$ is per definition given as $\langle \Psi \,|\, V \,|\, \Psi \rangle$. In our case, $|\Psi\rangle$ is a vector of N components and the potential is a matrix valued function as defined by (1.8) or alternatively a matrix of scalar functions.

If we go back to the definition, the potential energy is expressed by

$$\begin{aligned}
\langle \Psi \,|\, V \,|\, \Psi \rangle &= \left\langle \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \middle| \begin{pmatrix} v_{0,0}(x) & \dots & v_{0,N}(x) \\ \vdots & & \vdots \\ v_{N-1,0}(x) & \dots & v_{N-1,N-1}(x) \end{pmatrix} \middle| \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \right\rangle \\
&= \left\langle \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \middle| \begin{pmatrix} \sum v_{0,i}\Phi_i \\ \vdots \\ \sum v_{N-1,i}\Phi_i \end{pmatrix} \right\rangle
\end{aligned} \tag{4.42}$$

and with a little bit additional linear algebra we get the more handy result

$$\langle \Psi \,|\, V \,|\, \Psi \rangle = \sum_{i}^{N-1} \sum_{j}^{N-1} \langle \Phi_i \,|\, v_{i,j} \,|\, \Phi_j \rangle \tag{4.43}$$

thus effectively splitting the integral into a sum of integrals with only single components involved.

This simple case occurs when both components in the bra and the ket are identical, i.e. $\langle \Phi_i \,|\, v_{i,i} \,|\, \Phi_i \rangle$. And the more difficult case is $\langle \Phi_i \,|\, v_{i,j} \,|\, \Phi_j \rangle$ where the off diagonal terms of V appear. Their effect is obviously some kind of mixing the different components Φ_i and Φ_j and we will find the coefficient vectors of both, c^i and c^j in the resulting formula.

Both integrals can be handled by the algorithm 7 where we have to supply the correct $v_{i,j}(x)$ for f.

Notice that formula (4.43) is valid in the canonical basis. And we are interested in the energies of the components Φ_i as they are in the eigenbasis. To overcome this restriction we just transform to the eigenbasis by the equation (1.9):

$$\begin{aligned}\langle \Psi \,|\, V \,|\, \Psi \rangle &= \left\langle \Psi \,\middle|\, M(x)\,\Lambda(x)\,M^{-1}(x) \,\middle|\, \Psi \right\rangle \\ &= \left\langle \Psi M \,\middle|\, \Lambda(x) \,\middle|\, M^{\mathrm{T}} \Psi \right\rangle \\ &= \langle \Psi' \,|\, \Lambda(x) \,|\, \Psi' \rangle \;.\end{aligned} \tag{4.44}$$

Notice that $|\Psi'\rangle$ is not in the potential's eigenbasis yet because a projection to the subspace spanned by all Hagedorn basis functions ϕ_k is still missing. Thus we need the multiplication by $\mathbf{F}$ that also includes the transformation by M:

$$\begin{aligned}\langle \Psi \,|\, V \,|\, \Psi \rangle &= \langle \mathbf{F}\Psi \,|\, \Lambda(x) \,|\, \mathbf{F}\Psi \rangle \\ &= \langle \Psi^e \,|\, \Lambda(x) \,|\, \Psi^e \rangle \;.\end{aligned} \tag{4.45}$$

This finally results in

$$E_{\text{pot}} = \sum_{i}^{N-1} \langle \Phi_i^e \,|\, \lambda_i \,|\, \Phi_i^e \rangle \tag{4.46}$$

where we are left with just a single sum over the diagonal components. From this transformation we see that the overall potential energy is constant independently of the basis. But of course the potential energies of the components Φ_i are not.

4.8.2 Kinetic energy

The kinetic part of the energy of a homogeneous semiclassical wavepacket is given by

$$E_{\text{kin}} = \langle \Psi \,|\, T \,|\, \Psi \rangle \tag{4.47}$$

where T is the kinetic operator as given by definition (1.3). Recalling that $|\Psi\rangle$ contains several states Φ_i, we have to extend the above expression as follows

$$\begin{aligned}\langle \Psi \,|\, T \,|\, \Psi \rangle &= \left\langle \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \middle| \begin{pmatrix} T & & 0 \\ & \ddots & \\ 0 & & T \end{pmatrix} \middle| \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \right\rangle \\ &= \left\langle \begin{pmatrix} \Phi_0 \\ \vdots \\ \Phi_{N-1} \end{pmatrix} \middle| \begin{pmatrix} T\Phi_0 \\ \vdots \\ T\Phi_{N-1} \end{pmatrix} \right\rangle \\ &= \sum_{i=0}^{N-1} \langle \Phi_i \,|\, T \,|\, \Phi_i \rangle \;.\end{aligned} \tag{4.48}$$

From the fact that T is diagonal it becomes clear that the calculation of the kinetic energy can be achieved by component wise integration for each state.

To finish the calculation of kinetic energies we need to know the action of the kinetic operator T on a wavepacket Φ_i. This is not as easy as for the potential where $V(x)$ just acts by multiplication. In the following we use linearity of the sum and the differential operator

$$\begin{aligned} T\Phi &= -\frac{1}{2}\varepsilon^4 \frac{\partial^2}{\partial x^2}\Phi(x) \\ &= -\frac{1}{2}\varepsilon^4 \frac{\partial^2}{\partial x^2} e^{\frac{iS}{\varepsilon^2}} \sum_{k=0}^{K-1} c_k \phi_k \\ &= -\frac{1}{2}\varepsilon^4 e^{\frac{iS}{\varepsilon^2}} \sum_{k=0}^{K-1} c_k \frac{\partial^2}{\partial x^2}\phi_k(x) \ . \end{aligned} \tag{4.49}$$

With algorithm 12 we can calculate the action of $T' := -i\varepsilon^2 \frac{\partial}{\partial x}$ on Φ. Here the operator T' is a square root of T up to the prefactor $\frac{1}{2}$. We can apply it to both the bra and the ket of (4.48) individually:

$$\begin{aligned} E_{\text{kin}} &= \sum_i \langle \Phi_i \,|\, T \,|\, \Phi_i \rangle \\ &= \sum_i \left\langle \Phi_i \,\middle|\, T'^{\mathrm{H}} \left(\frac{1}{2}\right) T' \,\middle|\, \Phi_i \right\rangle \\ &= \sum_i \frac{1}{2} \langle T'\Phi_i \,|\, T'\Phi_i \rangle \\ &= \frac{1}{2} \sum_i \| T'\Phi_i \|^2 \, . \end{aligned} \tag{4.50}$$

Finally we need to transform the whole calculation to the eigenbasis. This is not difficult. The last step shows that we can calculate the kinetic energy as the norm of the coefficients of a transformed wavepacket $T'\Phi$ individually for each component. This allows us to transfer the wavepacket $|\Psi\rangle$ to the eigenbasis before we apply T'.

Algorithm 12 Calculate the action of T' on a wavepacket Φ

```
Require: A wavepacket $\Phi$ with coefficients $c_k$
  // Initialize a zero vector of length $K+1$
  $d := (0, \dots, 0)$
  // Base cases
  $d_0 := d_0 + pc_0$
  $d_1 := d_1 + \sqrt{\frac{\varepsilon^2}{2}} P c_0$
  // Inductive steps
  for $k := 1$ to $K$ do
    $d_k := d_k + pc_k$
    $d_{k+1} := d_{k+1} + \sqrt{k+1}\sqrt{\frac{\varepsilon^2}{2}} P c_k$
    $d_{k-1} := d_{k-1} + \sqrt{k}\sqrt{\frac{\varepsilon^2}{2}} \overline{P} c_k$
  end for
  return  $c := (d_0, \dots, d_{K-1})$
```

Chapter 5

Generalizing the wavepacket based algorithm

In this chapter we will extend further the ideas from the last chapter. Most concepts carry over if we generalize the necessary details as appropriate. Though some pitfalls appear at the theoretical computations as well as in the implementation.

5.1 Inhomogeneous wavepackets

When we release the restriction of chapter 4 that every wavepacket $|\Psi\rangle$ can only have a single set of Hagedorn parameters, all concepts of the last chapter remain valid. But some of the details become more complicated. In this chapter we'll spot these small differences and present the more general formulae.

So let's take the first step and drop the homogeneity restriction. For the rest of this chapter the wavepackets $|\Psi\rangle$ take the form defined by (3.16) where each component Φ_i is an expression as defined by (3.14). And every Φ_i possesses it's own set of Hagedorn parameters. Hence our wavepackets are now inhomogeneous ones.

5.2 Inner products, integrals and quadrature

5.2.1 An analytical ansatz

The inner product of two semiclassical wavepackets which have different sets of Hagedorn parameters denoted by $\{P_k, Q_k, S_k, p_k, q_k\}$ and $\{P_l, Q_l, S_l, p_l, q_l\}$ respectively is written as usual as $\left\langle \phi^k \,\middle|\, \phi^l \right\rangle$. This is the expression we want to evaluate now and we can even write down a closed form solution based on induction and the recursion relation for Hermite polynomials. The expression for the ground states ϕ_0 acts as induction base and is given by

$$\left\langle \phi_0^k \,\middle|\, \phi_0^l \right\rangle = \sqrt{\frac{-2i}{Q_2\overline{P_1} - P_2\overline{Q_1}}} \cdot \exp\left(\frac{i}{2\varepsilon^2} \frac{Q_2\overline{Q_1}\left(p_2 - p_1\right)^2 + P_2\overline{P_1}\left(q_2 - q_1\right)^2}{\left(Q_2\overline{P_1} - P_2\overline{Q_1}\right)} \right.$$
$$\left. - \frac{i}{\varepsilon^2} \frac{\left(q_2 - q_1\right)\left(Q_2\overline{P_1}p_2 - P_2\overline{Q_1}p_1\right)}{\left(Q_2\overline{P_1} - P_2\overline{Q_1}\right)} \right). \quad (5.1)$$

For the inner product of higher level functions ϕ_i the whole thing gets much more complicated:

$$\left\langle \phi_k^k \,\middle|\, \phi_l^l \right\rangle = \frac{1}{\sqrt{k!l!}} 2^{-\frac{k+l}{2}} \left\langle \phi_0^k \,\middle|\, \phi_0^l \right\rangle \cdot \left(i\overline{P_1}Q_2 - i\overline{Q_1}P_2\right)^{-\frac{k+l}{2}} \cdot$$
$$\sum_{j=0}^{\min(k,l)} \left(\binom{k}{j}\binom{l}{j} j! 4^j \left(iQ_2P_1 - iQ_1P_2\right)^{\frac{k-j}{2}} \left(\overline{iQ_2P_1 - iQ_1P_2}\right)^{\frac{l-j}{2}} \right.$$
$$\cdot H_{k-j}\left(-\frac{1}{\varepsilon} \frac{Q_2\left(p_1 - p_2\right) - P_2\left(q_1 - q_2\right)}{\sqrt{Q_2P_1 - Q_1P_2}\sqrt{\overline{P_1}Q_2 - \overline{Q_1}P_2}} \right)$$
$$\left. \cdot H_{l-j}\left(\frac{1}{\varepsilon} \frac{\overline{P_1}\left(q_1 - q_2\right) - \overline{Q_1}\left(p_1 - p_2\right)}{\sqrt{\overline{Q_2P_1} - \overline{Q_1P_2}}\sqrt{\overline{P_1}Q_2 - \overline{Q_1}P_2}} \right) \right). \quad (5.2)$$

For the proofs of these formulae see reference [10].

Despite we can evaluate the inner product and have a closed form solution for arbitrary wave functions, these formulae are unsuitable for numerical calculation. There are several reasons but for example the factorials and binomial coefficients lead to overflow even for relatively small k and l. Further the sum may be numerically unstable. Thus we need to find a better way to perform these calculations.

5.2.2 Quadrature rules for the product of basis functions

First we notice that each ϕ which is given by (3.9) is represented through a mathematical expression of the general form

$$C \cdot P_n\left(\xi\right) \cdot \exp\left(\theta\right) \quad (5.3)$$

consisting of an arbitrary constant $C \in \mathbb{C}$, a polynomial $P_n\left(\cdot\right)$ of degree n and an exponential $\exp\left(\cdot\right)$.

We try a new Ansatz for calculating the inner product. Evaluating the braket $\langle \phi^k \mid \phi^l \rangle$ results in a multiplication of two expressions of the form (5.3):

$$\begin{aligned} \langle \phi^k \mid \phi^l \rangle &= \int_{\mathbb{R}} \overline{C_k P_{n_k}(\xi_k) \exp(\theta_k)} C_l P_{n_l}(\xi_l) \exp(\theta_l)\, dx \\ &= \int_{\mathbb{R}} \overline{C_k} C_l P_{n_k}\left(\overline{\xi_k}\right) P_{n_l}(\xi_l) \exp\left(\overline{\theta_k}\right) \exp(\theta_l)\, dx\,. \end{aligned} \tag{5.4}$$

The parts in this expression can be combined by same type. With Gauss Hermite quadrature in mind we are especially interested in the exponential parts. They have a general form like

$$\exp(\theta) = \exp\left(s \cdot (x-m)^2 + \cdots\right)\,. \tag{5.5}$$

Therefore lets take a closer look at these parts. Our first step consists of combining the exponentials and distribute the complex conjugate onto the variables affected.

$$\begin{aligned} &\overline{\exp\left(\frac{i}{2\varepsilon^2} P_k Q_k^{-1} (x-q_k)^2 + \frac{i}{\varepsilon^2} p_k (x-q_k)\right)} \cdot \exp\left(\frac{i}{2\varepsilon^2} P_l Q_l^{-1} (x-q_l)^2 + \frac{i}{\varepsilon^2} p_l (x-q_l)\right) \\ =& \exp\left(\overline{\frac{i}{2\varepsilon^2} P_k Q_k^{-1} (x-q_k)^2 + \frac{i}{\varepsilon^2} p_k (x-q_k)} + \frac{i}{2\varepsilon^2} P_l Q_l^{-1} (x-q_l)^2 + \frac{i}{\varepsilon^2} p_l (x-q_l)\right) \\ =& \exp\left(-\frac{i}{2\varepsilon^2} \overline{P_k}\,\overline{Q_k^{-1}} (x-q_k)^2 - \frac{i}{\varepsilon^2} p_k (x-q_k) + \frac{i}{2\varepsilon^2} P_l Q_l^{-1} (x-q_l)^2 + \frac{i}{\varepsilon^2} p_l (x-q_l)\right)\,. \end{aligned} \tag{5.6}$$

For the sake of readability we define the following variables

$$\begin{aligned} r_k &:= P_k Q_k^{-1} \\ r_l &:= P_l Q_l^{-1}\,. \end{aligned} \tag{5.7}$$

Plugging these into the equation (5.6) above and expanding the squares we get for the exponent

$$\frac{i}{\varepsilon^2}\left(-\frac{1}{2}\left(\overline{r_k} - r_l\right) x^2 + \left(\overline{r_k} q_k - r_l q_l\right) x - \frac{1}{2}\left(\overline{r_k} q_k^2 + r_l q_l^2\right) + (p_l - p_k) x + p_k q_k - p_l q_l\right)\,. \tag{5.8}$$

To get back to a form along the lines of (5.5) we have to complete the square

$$\frac{i}{\varepsilon^2}\left(\left(x^2 - 2\underbrace{\frac{\overline{r_k} q_k - r_l q_l}{\overline{r_k} - r_l}}_{q_0} x + q_0^2 - q_0^2\right)\left(-\frac{\overline{r_k} - r_l}{2}\right) - \frac{1}{2}\left(\overline{r_k} q_k^2 + r_l q_l^2\right) + \cdots\right) \tag{5.9}$$

which gives

$$\frac{i}{\varepsilon^2}\left(\frac{1}{2}\left(r_l-\overline{r_k}\right)\left(x-q_0\right)^2+\frac{1}{2}\left(\overline{r_k}-r_l\right)q_0^2-\frac{1}{2}\left(\overline{r_k}q_k^2+r_lq_l^2\right)+\cdots\right). \tag{5.10}$$

Finally we got just a new expression which has the general form shown in (5.5). In this expression, q_0 represents the mean of the Gaussian function while the prefactor defines the variance or spread. With this results we can now go on and adapt the Gauss-Hermite quadrature for the case of unequal Hagedorn parameters and compute the value of an arbitrary integral $\langle \phi^k \mid \phi^l \rangle$. As usual we have to transform the quadrature nodes such that they lie in the important region of space. For this we define the following variables

$$\begin{aligned}
r_l &:= \frac{P_l}{Q_l}\\
r_k &:= \frac{P_k}{Q_k}\\
r_0 &:= \frac{1}{2}\left(r_l-\overline{r_k}\right)\\
q_0 &:= \Re\frac{r_lq_l-\overline{r_k}q_k}{r_l-\overline{r_k}}\\
Q_0 &:= \frac{1}{\sqrt{\Im r_0}}
\end{aligned} \tag{5.11}$$

Now we transform the nodes γ_i according to the weighted position mean q_0 and the parameter Q_0 which changes the spread of the nodes. This yields

$$\gamma_i' := q_0 + \varepsilon\cdot\Re Q_0\cdot\gamma_i \tag{5.12}$$

for the new quadrature nodes which are located in the space around where the product of ϕ_k and ϕ_l is maximal. A procedure that calculates the adapted quadrature results as necessary is given by algorithm 13.

We should get back to the homogeneous case if we choose the sets Π_k and Π_l of Hagedorn parameters identical.

5.2.3 Integrals of whole components

Suppose the wave function $|\Phi_i\rangle$ of a component is given as defined by (3.14). For an arbitrary but sufficiently smooth real valued function

$$\begin{aligned}
f:\mathbb{R}&\rightarrow\mathbb{R}\\
x&\mapsto f\left(x\right)
\end{aligned}$$

Algorithm 13 Mixing two sets Π_r and Π_c of Hagedorn parameters

```
Require: Two sets $\Pi_r$ and $\Pi_c$ of Hagedorn parameters
Require: A quadrature rule $(\gamma_i, \omega_i)$
  // Apply the mixing formula to the parameters
  $r_r := \frac{P_r}{Q_r}$
  $r_c := \frac{P_c}{Q_c}$
  $r_0 := \frac{1}{2}\left(r_r - \overline{r_c}\right)$
  $q_0 := \Re \frac{r_r q_r - \overline{r_c} q_c}{r_r - \overline{r_c}}$
  $Q_0 := \frac{1}{\sqrt{\Im r_0}}$
  // And shift the quadrature nodes
  $\gamma' := q_0 + \varepsilon \Re Q_0 \gamma$
  return $q_0$ and $Q_0$ and $\gamma'$
```

we want to simplify the following expression $\langle \Phi_k \,|\, f \,|\, \Phi_l \rangle$. But in contrast to the derivation (4.13) we have to mind the fact the the different components Φ_i may belong to two different parameter families. That is, they have different Hagedorn parameters. Denote by ϕ^i the functions of the form (3.9) that build the base of Φ_i then we can write:

$$\begin{aligned}
\langle \Phi_k \,|\, f \,|\, \Phi_l \rangle &= \left\langle e^{\frac{iS^k}{\varepsilon^2}} \sum_i c_i^k \phi_i^k \,\middle|\, f \,\middle|\, e^{\frac{iS^l}{\varepsilon^2}} \sum_i c_i^l \phi_i^l \right\rangle \\
&= \underbrace{e^{-\frac{i\overline{S^k}}{\varepsilon^2}} e^{\frac{iS^l}{\varepsilon^2}}}_{\neq 1} \left\langle \sum_i c_i^k \phi_i^k \,\middle|\, f \,\middle|\, \sum_i c_i^l \phi_i^l \right\rangle \\
&= e^{\frac{i}{\varepsilon^2}\left(S^l - \overline{S^k}\right)} \sum_{i,j} \overline{c_i^k} c_j^l \left\langle \phi_i^k \,\middle|\, f \,\middle|\, \phi_j^l \right\rangle \\
&= e^{\frac{i}{\varepsilon^2}\left(S^l - \overline{S^k}\right)} \sum_{i,j} \overline{c_i^k} c_j^l \int_{\mathbb{R}} \overline{\phi_i^k\left(x\right)} f\left(x\right) \phi_j^l\left(x\right) dx\,.
\end{aligned}$$

Notice that the phases don't cancel out anymore as each Φ_i has it's own phase of different magnitude. We finally reduced the calculation of $\langle \Psi \,|\, f \,|\, \Psi \rangle$ to a sum of more simple integrals $\left\langle \phi_i^l \,\middle|\, f \,\middle|\, \phi_j^k \right\rangle$ over products of basis functions. This integral can now be evaluated by the means of numerical quadrature. We can use 7 for this purpose but have to replace the algorithm that is responsible for calculating F.

Integrals of whole wavepackets $|\Psi\rangle$ can now be split into sums of integrals over components. That is, we have

$$\langle \Psi \,|\, M \,|\, \Psi \rangle = \sum_{i,j} \langle \Phi_i \,|\, M_{i,j} \,|\, \Phi_j \rangle \tag{5.13}$$

where M is a $N \times N$ matrix of scalar functions. We will use this identity for simplifying the calculation of energies.

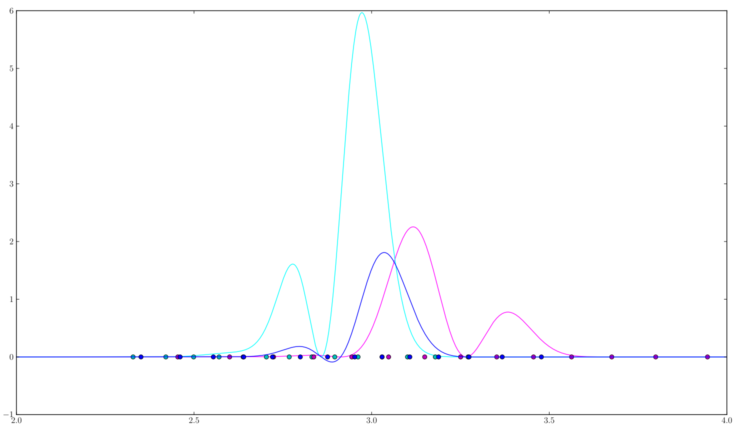


Figure 5.1: Example of a quadrature for a given wavepacket Ψ. The component $\overline{\Phi_0}\Phi_0$ (magenta) and $\overline{\Phi_1}\Phi_1$ (cyan) are shown together with their product $\overline{\Phi_0}\Phi_1$ (blue). The quadrature nodes have the color of the component defining them.

The figure 5.1 shows the quadrature of an inhomogeneous wavepacket $|\Psi\rangle$ that has two components Φ_0 and Φ_1. The Hagedorn parameters are $\Pi_0 = (i, 3, 0, 0.4, 3.2)$ and $\Pi_0 = (i, 1.2, 0, 0.4, 2.8)$ and it's coefficients are $c^0 = (0, 0.3, 0.4, 0, 0, 0.6, 0.2)$ and $c^0 = (0, 0.8, 0.7, 0.4)$. We can see the effect of the algorithm 13 that mixes the cyan and the magenta nodes yielding the blue ones which fit best to the product of the components Φ_i.

5.3 The propagation algorithm for inhomogeneous wavepackets

In this section we will discuss the same two central points that have to be resolved for expanding algorithm 10 to inhomogeneous wavepackets. The most parts are straight forward and obvious but we face some difficulties too. Let's begin with the easy part, the potential splitting.

5.3.1 Splitting of the potential matrix

For an inhomogeneous vector valued wavepacket $|\Psi\rangle$ we drop the concept of a leading index χ. Instead, each energy level λ_i is responsible for propagating the Hagedorn parameters of the packet's component Φ_i. Because of this reason we have to perform the quadratic Taylor approximation for all N eigenvalues. Let's restrict to one space dimension right now. Then the approximations read

$$\begin{aligned} u_i(x) &= \lambda_i|_{x=q} + \frac{d}{dx}\lambda_i|_{x=q}(x-q) + \frac{1}{2}\frac{d^2}{dx^2}\lambda_i|_{x=q}(x-q)^2 \\ w_i(x) &= \lambda_i(x) - u_i(x) \quad \forall i \in 0,\dots,N-1 \end{aligned} \tag{5.14}$$

where q is the point around which the Taylor expansion is centred.

We can write the potential matrix as a pure quadratic diagonal part U plus a non-quadratic remainder matrix W once again:

$$V = \begin{pmatrix} u_0 & & 0 \\ & \ddots & \\ 0 & & u_{N-1} \end{pmatrix} + \begin{pmatrix} v_{0,0} - u_0 & \dots & v_{0,N-1} \\ \vdots & \ddots & \vdots \\ v_{N-1,0} & \dots & v_{N-1,N-1} - u_{N-1} \end{pmatrix} . \tag{5.15}$$

But this time, the matrix U is not just a scaled identity. And we really use all it's entries. The part W again serves as the matrix for the calculation of $\mathbf{F}$ required in an adapted version of formula (4.11). We will use W in the algorithm 14 where we explicitly build this matrix $\mathbf{F}$.

5.3.2 The coefficients

We stack the coefficient vectors c^n of the N components Φ_n in the same way as shown in (4.30). Thus we need again a matrix $\mathbf{F}$ analogous to the one shown in (4.31) for the time evolution of C. This time we have to take extra care because of the different Hagedorn parameters of each Φ. Basically this prevents us from reusing the evaluated basis functions for all blocks $F_{r,c}$ of $\mathbf{F}$. And we must not forget the global phase that does not cancel out this time.

It becomes immediately clear that this algorithm 14 is more expensive than algorithm 9, for example we have to evaluate the basis for the two components Φ_i and Φ_j over and over again as they differ by their Hagedorn parameters.

5.3.3 Pseudo code for the time propagation

With the preparations of the last section we can now construct an algorithm for the time propagation of an inhomogeneous wavepacket $|\Psi\rangle$. This algorithm is a generalization of the time propagation given in algorithm 10 to allow for different families of parameters. The core concepts carry over, but the details are a little bit more complex as we saw in the last sections.

Algorithm 14 Build the inhomogeneous block matrix $\mathbf{F} := \left(F_{r,c}\right)_{r,c}$

```
Require: A inhomogeneous wavepacket $\Psi$
Require: $W$ a $N \times N$ matrix of scalar functions
  // Initialize $\mathbf{F}$ as the zero-matrix
  $\mathbf{F} \in \mathbb{R}^{NK \times NK}, \quad \mathbf{F} := \mathbf{0}$
  // Iterate over all row and column blocks of this matrix
  for $r = 0$ to $N-1$ do
    for $c = 0$ to $N-1$ do
      // Retrieve the Hagedorn parameters
      given $\Pi_r$ as $\{P_r, Q_r, S_r, p_r, q_r\}$
      given $\Pi_c$ as $\{P_c, Q_c, S_c, p_c, q_c\}$
      Apply the mixing formula to the parameters according to procedure 13
      // Evaluate the function $W_{r,c}$ for all quadrature nodes $\gamma'$
      $(v_0, \ldots, v_{L-1}) := W_{r,c}\left(\left(\gamma'_0, \ldots, \gamma'_{L-1}\right)\right)$
      // Evaluate the basis functions for all quadrature nodes $\gamma'$
      // Apply algorithm 3 for $\Pi_r$ and $\Pi_c$ individually
      $B_r := \left(\beta^r_0, \ldots, \beta^r_{K-1}\right)$
      $B_c := \left(\beta^c_0, \ldots, \beta^c_{K-1}\right)$
      // Do not forget the non-vanishing phase
      $\pi_{r,c} := \exp\left(\frac{i}{\varepsilon^2}\left(S_c - \overline{S_r}\right)\right)$
      // Set up a zero matrix
      $F \in \mathbb{R}^{K \times K}, \quad F := \mathbf{0}$
      // Iterate over all $L$ quadrature pairs $(\gamma'_l, \omega_l)$
      for $l = 0$ to $L-1$ do
        $F := F + v_l \varepsilon M_Q \cdot {B_r}^{\mathrm{H}} B_c \cdot \omega_l$
      end for
      // Insert the block $F$ into the block matrix $\mathbf{F}$
      $\mathbf{F}_{r,c} := \pi_{r,c} \cdot F$
    end for
  end for
  return $\mathbf{F}$
```

Algorithm 15 Time propagation of a inhomogeneous wavepacket $|\Psi\rangle$

```
Require: A semiclassical wavepacket $|\Psi(t)\rangle$
Require: The sets $\Pi_0, \dots \Pi_{N-1}$ of Hagedorn parameters of $\Psi$
  // Propagate with the kinetic operator
  for $n := 0$ to $N-1$ do
    $q_n^{\left(j+\frac{1}{2}\right)} := q_n^{(j)} + \frac{\tau}{2} p_n^{(j)}$
    $Q_n^{\left(j+\frac{1}{2}\right)} := Q_n^{(j)} + \frac{\tau}{2} P_n^{(j)}$
    $S_n^{\left(j+\frac{1}{2},-\right)} := S_n^{(j)} + \frac{\tau}{4} p_n^{(j)\mathrm{T}} p_n^{(j)}$
  end for
  // Propagate with the local quadratic potential
  for $n := 0$ to $N-1$ do
    $p_n^{(j+1)} := p_n^{(j)} - \tau \nabla \lambda_n \left( q_n^{(j+1/2)} \right)$
    $P_n^{(j+1)} := P_n^{(j)} - \tau \nabla^2 \lambda_n \left( q_n^{(j+1/2)} \right) Q_n^{(j+1/2)}$
    $S_n^{(j+1/2,+)} := S_n^{(j+1/2,-)} - \tau \lambda_n \left( q_n^{(j+1/2)} \right)$
  end for
  // Propagate with the non-quadratic remainder
  // Stack the coefficient vectors $c^n$ of all components
  $C^{(j)} := \left( c^0, \dots, c^{N-1} \right)^{\mathrm{T}}$
  // Assemble the matrix $\mathbf{F}$ using algorithm 14
  $\mathbf{F}^{(j+1/2)} := \left( F_{r,c} \right)_{r,c} \quad \forall r, c \in 0, \dots, N-1$
  // Propagate the coefficients
  $C^{(j+1)} := \exp\left( -\tau \frac{i}{\varepsilon^2} \mathbf{F}^{(j+1/2)} \right) C^{(j)}$
  // Split the coefficients
  $\left( c^0, \dots, c^{N-1} \right) := C^{(j+1)}$
  // Propagate with the kinetic operator again
  for $n := 0$ to $N-1$ do
    $q_n^{(j+1)} := q_n^{(j+1/2)} + \frac{\tau}{2} p_n^{(j+1)}$
    $Q_n^{(j+1)} := Q_n^{(j+1/2)} + \frac{\tau}{2} P_n^{(j+1)}$
    $S_n^{(j+1)} := S_n^{(j+1/2,+)} + \frac{\tau}{4} p_n^{(j+1)\mathrm{T}} p_n^{(j+1)}$
  end for
  return $|\Psi(t+\tau)\rangle$
```

5.4 Basis transformation

Basis transformations from and to the eigenbasis work in principle exactly like defined in section 4.6. We can again write this as a matrix multiplication with a big block matrix $\mathbf{F}$:

$$\left(\begin{array}{c} d^0 \\ \hline \vdots \\ \hline d^{N-1} \end{array}\right) = \left(\begin{array}{c|c|c} \langle \varphi_0 \,|\, m_{0,0} \,|\, \varphi_0 \rangle & \cdots & \langle \varphi_0 \,|\, m_{0,N-1} \,|\, \varphi_{N-1} \rangle \\ \hline \vdots & & \vdots \\ \hline \langle \varphi_{N-1} \,|\, m_{N-1,0} \,|\, \varphi_0 \rangle & \cdots & \langle \varphi_{N-1} \,|\, m_{N-1,N-1} \,|\, \varphi_{N-1} \rangle \end{array}\right) \left(\begin{array}{c} c^0 \\ \hline \vdots \\ \hline c^{N-1} \end{array}\right)$$

where each φ_n collects the basis functions ϕ_k of component Φ_n in a vector and d^i denotes the transformed coefficient vectors. This time all φ_n differ because we deal with inhomogeneous wavepackets. Only for the calculation of the submatrices we now switch to algorithm 14 instead of 9 in the homogeneous case.

5.5 Observables

The calculation of observables is roughly the same as with homogeneous wavepackets but we encounter some details which are not present in the less general version. However, we will only outline the changes with respect to the corresponding section in chapter 4.

5.5.1 Norm calculation

The calculation of the norm of our wavepackets is as simple as in the homogeneous case. We only need the inner products for all components. And the nice thing is that all the integrals consist of only bras and kets with an equal family of parameters

$$\|\Psi\|^2 = \langle \Psi \,|\, \Psi \rangle = \sum_{i}^{N-1} \langle \Phi_i \,|\, \Phi_i \rangle = \sum_{i}^{N-1} \|c^i\|^2 \,. \tag{5.16}$$

The only tricky part here may be the transformation to the eigenbasis to get the norms of all components we are interested in. But this change of basis can be done easily as shown in 5.4.

5.5.2 Potential and kinetic energy

We are interested in the energies of the wavepacket and its components in the eigenbasis. Hence we need the generalized basis transformation defined above here too. Besides this, everything remains valid and especially the formulae (4.46) and (4.50) for the potential and the kinetic energy still hold.

Chapter 6

Simulation results

In this chapter we present some selected simulations in more detail and show some of the results[1].

6.1 The harmonic oscillator

The harmonic oscillator is probably the most basic non-trivial potential that fulfils the necessary smoothness assumptions. Of course we can solve this model entirely by analytical calculations. But because of this property it's an excellent starting point for testing and calibrating new simulation codes.

$$V(x) := \begin{pmatrix} \frac{1}{2}\sigma x^2 & 0 \\ 0 & \frac{1}{2}\sigma x^2 \end{pmatrix} \qquad \sigma = 0.05 \tag{6.1}$$

This potential is already diagonal, thus we can not expect any interaction of the two components of $|\Psi\rangle$. Figure 6.1 shows the time evolution of the energies of a wavepacket on each level. The image looks like we expected it.

Let's now look at more interesting potentials in the next sections.

6.2 A simple avoided crossing

In this section we present some results for a potential that has a simple single avoided crossing. We used several different values for the energy gap δ. The

[1] Many more simulation results can be found at:
http://n.ethz.ch/~raoulb/research/bachelor_thesis/simulations/

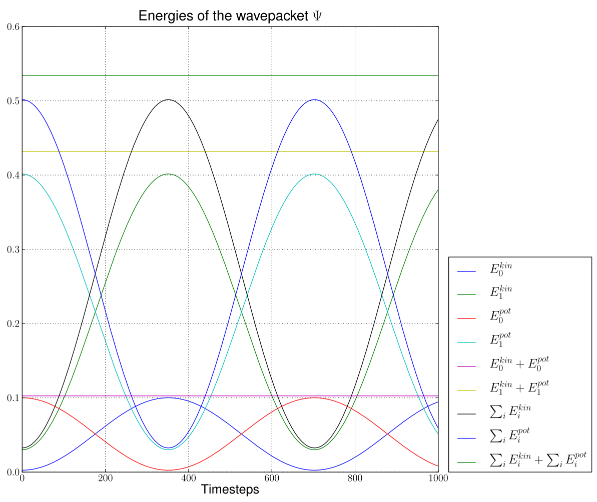


Figure 6.1: The time evolution of the energies of an initial wavepacket on each level.

potential is given by the following matrix:

$$V(x) := \begin{pmatrix} \frac{1}{2}\tanh(x) & \frac{\delta}{2} \\ \frac{\delta}{2} & -\frac{1}{2}\tanh(x) \end{pmatrix} . \tag{6.2}$$

The two energy levels of this potential are

$$\lambda_0 = \frac{\sqrt{\tanh(x)^2 + \delta^2}}{2} \qquad \lambda_1 = -\frac{\sqrt{\tanh(x)^2 + \delta^2}}{2} \tag{6.3}$$

Figure 6.2 shows these two energy levels for the parameter δ set to 0.05.

This potential is a standard example for avoided crossings and consists of nothing but the essential properties a crossing has.

The simulation starts with an incoming Gaussian wavepacket on the upper level. An initial momentum pointing to the right (positive x axis) is used in the following.

We did several simulations within a wide range of the parameters ε and δ. Some of these simulations are shown here. Let's compare the operator splitting based method with the wavepacket based one. Here we used homogeneous wavepackets with the leading components χ set to the upper energy level.

Figures 6.3 and 6.4 show the energies and the norms of Ψ and it's components Φ_0 on the upper level and Φ_1 on the lower one for several simulation runs based on operator splitting. The figures 6.5 and 6.6 show the simulation results

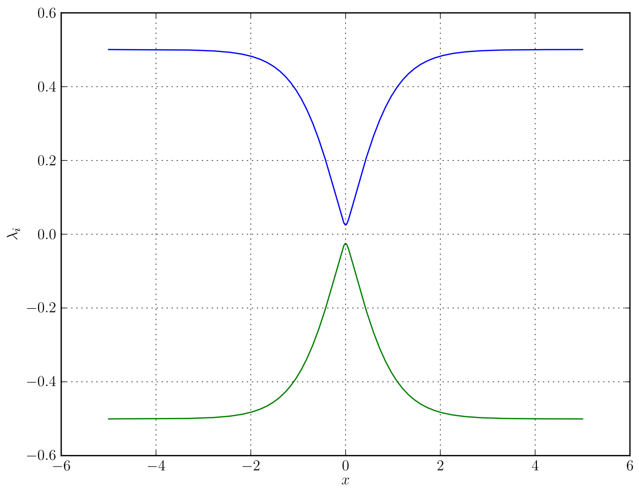


Figure 6.2: Plot of the energy levels of the potential given by equation (6.2). The parameter δ equals 0.05.

obtained by using semiclassical wavepackets and identical initial conditions. Both algorithms yield the same energy curves within the limits of optical comparison.

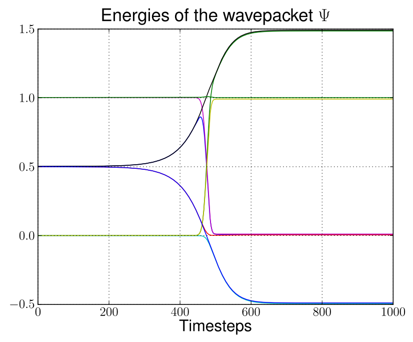
From the plots of the norms we can estimate the part of the wavepacket that remains on the respective energy level after the packet has crossed the narrow part in the middle. While for the smallest δ most of the packet jumps over to the lower energy level we see that for a bigger energy gap δ almost no transition takes place.

Finally we can say that the wavepacket based algorithm works very well in this case.

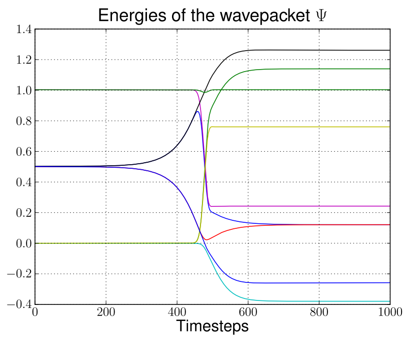
It may be interesting to see the time evolution of the Hagedorn parameters Π and the coefficients c^i of a wavepacket $|\Psi\rangle$. In the figure 6.7 we see the evolution of the parameters. The figures 6.8 and 6.9 show the first and last four coefficients of both components Φ_i. The general setup of the simulation corresponds to the example in the above figures 6.5c and 6.6c

6.3 Two avoided crossings in series

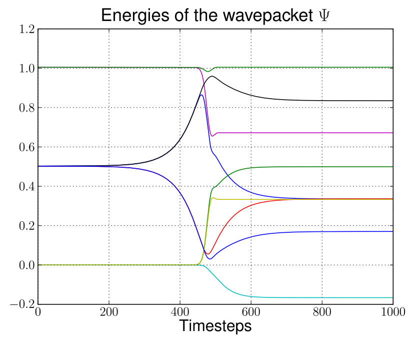
After we have seen the simulation results for a single avoided crossing let's look at another interesting question. That is, what happens if we have multiple of these avoided crossings in series. When entering the second one, the wavepacket



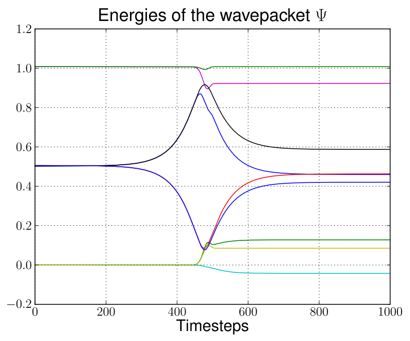
(a)



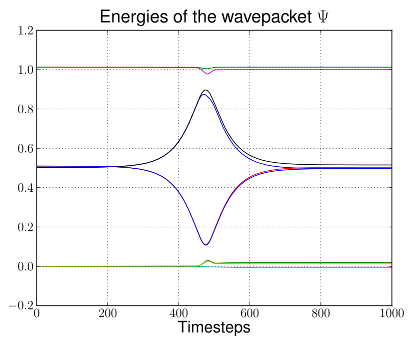
(b)



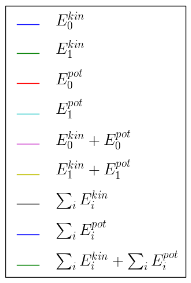
(c)



(d)

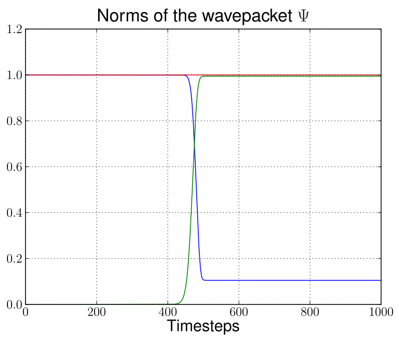


(e)

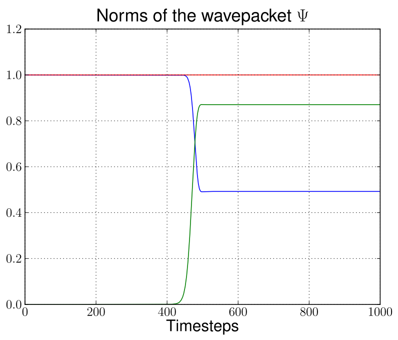


(f)

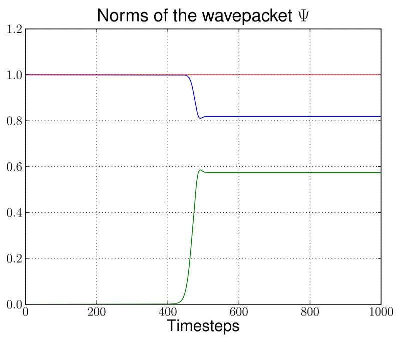
Figure 6.3: Plots of the energies of the wavepacket's individual components Φ_i. These results were obtained by the operator splitting method. (a) $\varepsilon = 0.1$ and $\delta = 0.1\varepsilon$ (b) $\varepsilon = 0.1$ and $\delta = 0.5\varepsilon$ (c) $\varepsilon = 0.1$ and $\delta = 1.0\varepsilon$ (d) $\varepsilon = 0.1$ and $\delta = 1.5\varepsilon$ (e) $\varepsilon = 0.1$ and $\delta = 2.0\varepsilon$



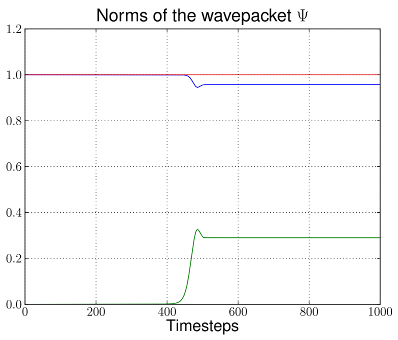
(a)



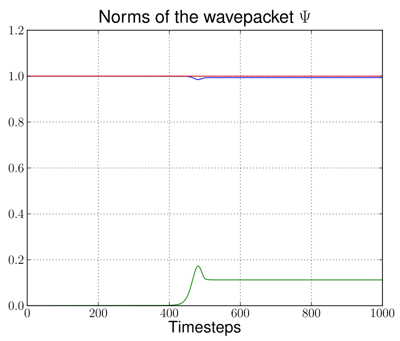
(b)



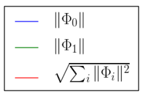
(c)



(d)

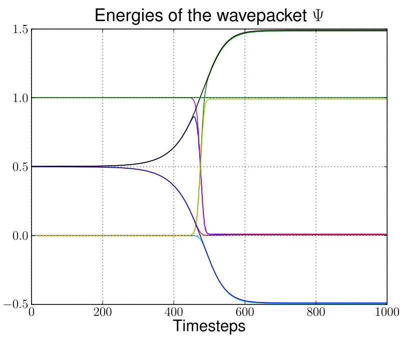


(e)

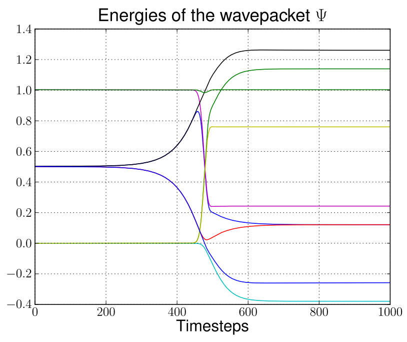


(f)

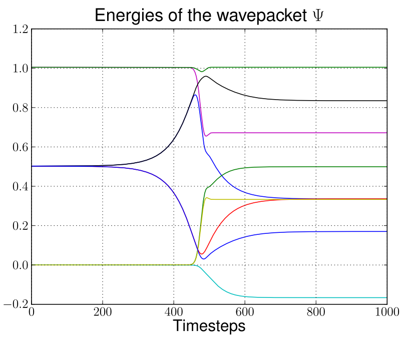
Figure 6.4: Plots of the norms of the wavepacket's individual components Φ_i. These results were obtained by the operator splitting method. (a) $\varepsilon = 0.1$ and $\delta = 0.1\varepsilon$ (b) $\varepsilon = 0.1$ and $\delta = 0.5\varepsilon$ (c) $\varepsilon = 0.1$ and $\delta = 1.0\varepsilon$ (d) $\varepsilon = 0.1$ and $\delta = 1.5\varepsilon$ (e) $\varepsilon = 0.1$ and $\delta = 2.0\varepsilon$



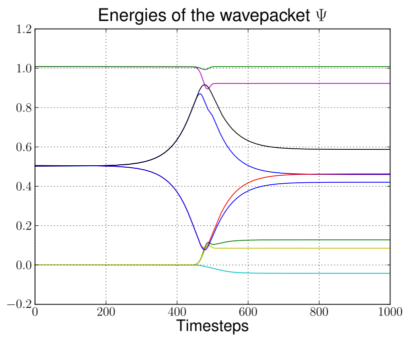
(a)



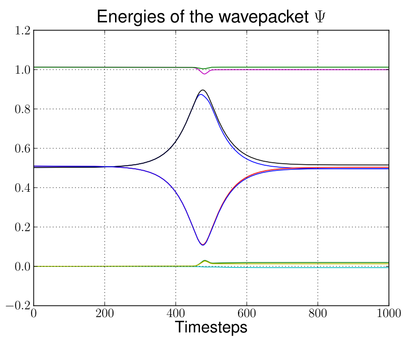
(b)



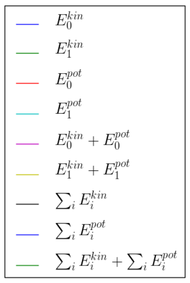
(c)



(d)

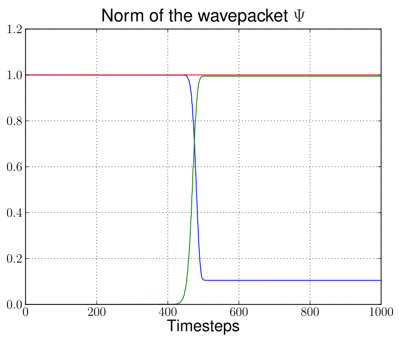


(e)

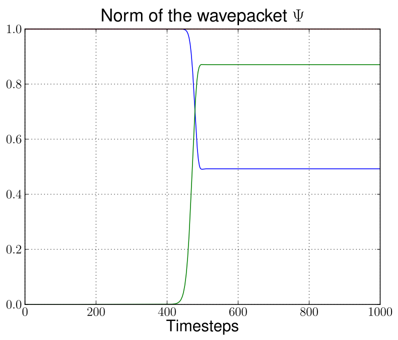


(f)

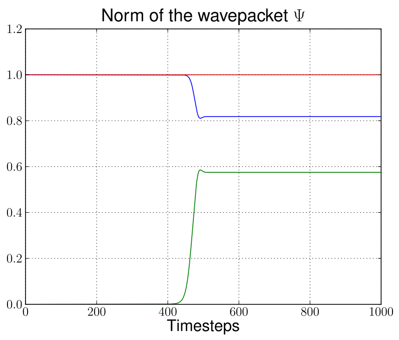
Figure 6.5: Plots of the energies of the wavepacket's individual components Φ_i. These results were obtained by propagating wavepackets. (a) $\varepsilon = 0.1$ and $\delta = 0.1\varepsilon$ (b) $\varepsilon = 0.1$ and $\delta = 0.5\varepsilon$ (c) $\varepsilon = 0.1$ and $\delta = 1.0\varepsilon$ (d) $\varepsilon = 0.1$ and $\delta = 1.5\varepsilon$ (e) $\varepsilon = 0.1$ and $\delta = 2.0\varepsilon$



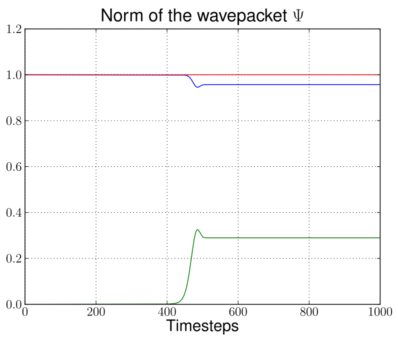
(a)



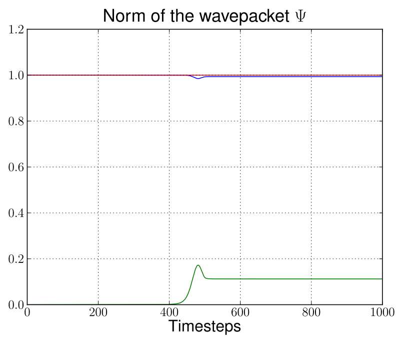
(b)



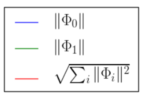
(c)



(d)



(e)



(f)

Figure 6.6: Plots of the norms of the wavepacket's individual components Φ_i. These results were obtained by propagating wavepackets. (a) $\varepsilon = 0.1$ and $\delta = 0.1\varepsilon$ (b) $\varepsilon = 0.1$ and $\delta = 0.5\varepsilon$ (c) $\varepsilon = 0.1$ and $\delta = 1.0\varepsilon$ (d) $\varepsilon = 0.1$ and $\delta = 1.5\varepsilon$ (e) $\varepsilon = 0.1$ and $\delta = 2.0\varepsilon$

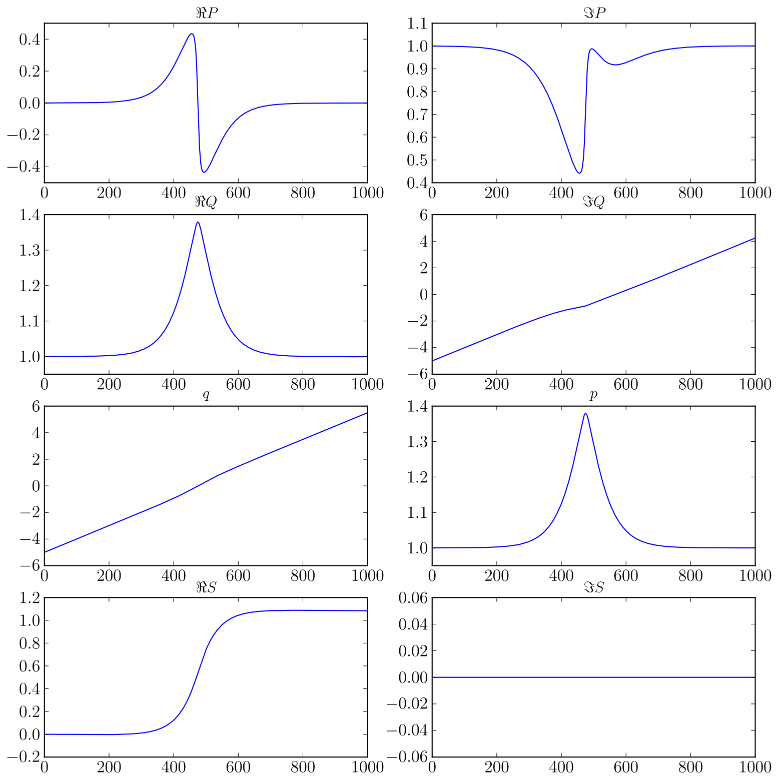


Figure 6.7: Plot of the time evolution of the Hagedorn parameters P, Q, S, p and q. Mind the scales!

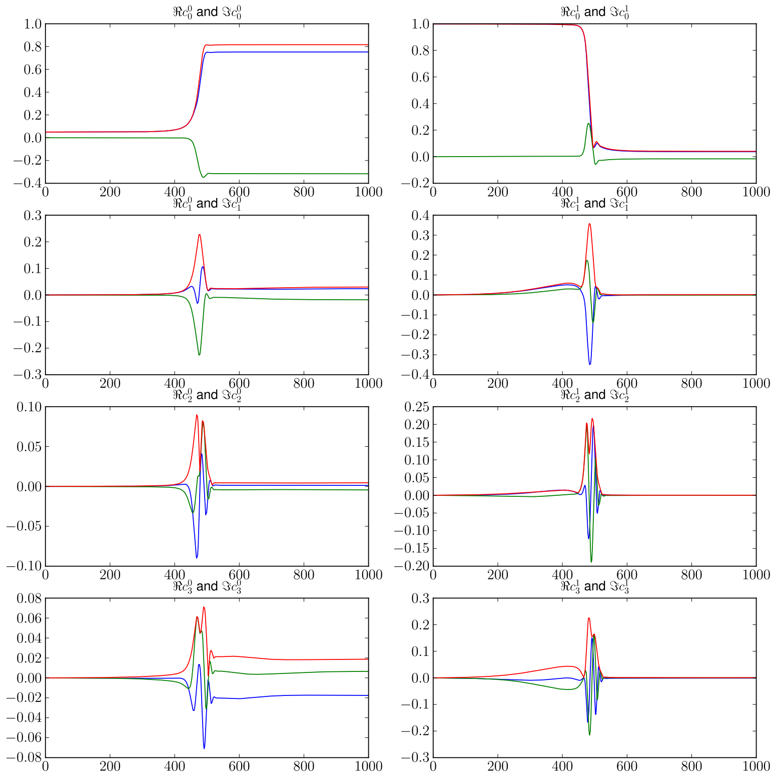


Figure 6.8: Plot of the time evolution of the first 4 coefficients c_0, c_1, c_2, c_3 for both components. The blue and the green line are the real and the imaginary part, and the red one is the absolute value. Mind the scales!

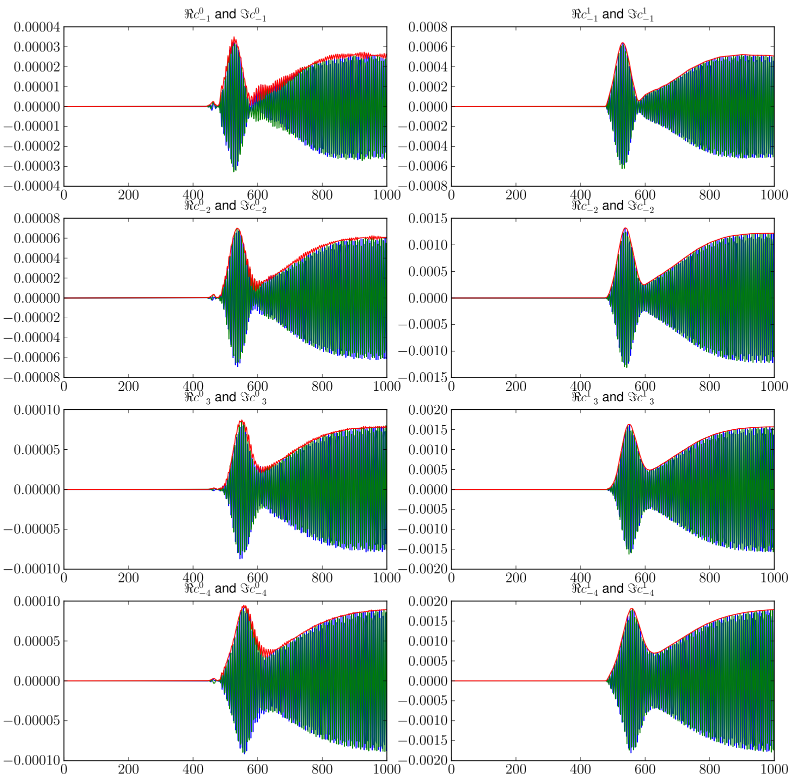


Figure 6.9: Plot of the time evolution of the last 4 coefficients c_{-4}, c_{-3}, c_{-2}, c_{-1} for both components. The blue and the green line are the real and the imaginary part, and the red one is the absolute value. Mind the scales!

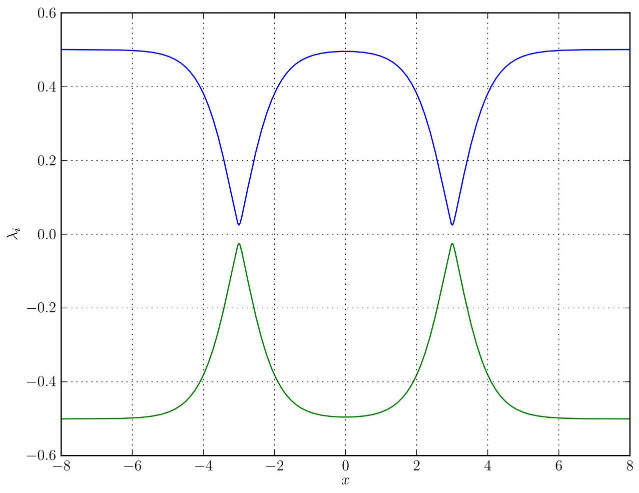


Figure 6.10: Plot of the energy levels of the potential given by equation (6.4). The parameter δ equals 0.05.

is already scattered to both energy levels. But first we define the potential:

$$V(x) := \begin{pmatrix} \frac{1}{2}\tanh(x-\sigma)\tanh(x+\sigma) & \frac{\delta}{2} \\ \frac{\delta}{2} & -\frac{1}{2}\tanh(x-\sigma)\tanh(x+\sigma) \end{pmatrix} \quad \sigma = 3\,. \tag{6.4}$$

The parameter σ determines the location of the avoided crossings. For an even longer series we could use a product of more factors:

$$V_{0,0}(x) := \prod_i \tanh(x-\sigma_i) \tag{6.5}$$

for an arbitrary set $\{\sigma_i\}_i$ and $V_{1,1} := -V_{0,0}$. However let's return to the simplest case of only two narrow parts. The two energy levels of this potential are given by:

$$\lambda = \pm \frac{\sqrt{\tanh(x-\sigma)^2 \tanh(x+\sigma)^2 + \delta^2}}{2}\,. \tag{6.6}$$

Figure 6.10 shows these two energy levels for the parameter δ set to 0.05. The effect of the parameter δ is shown in the plot 6.11 for multiple values ranging from 0.5ε up to 10ε. For bigger δ we get an increasing energy gap and also much smoother insections.

The figures in 6.12 show the energy evolution of a wavepacket traversing the potential shown in 6.10. The norms of the individual components are shown

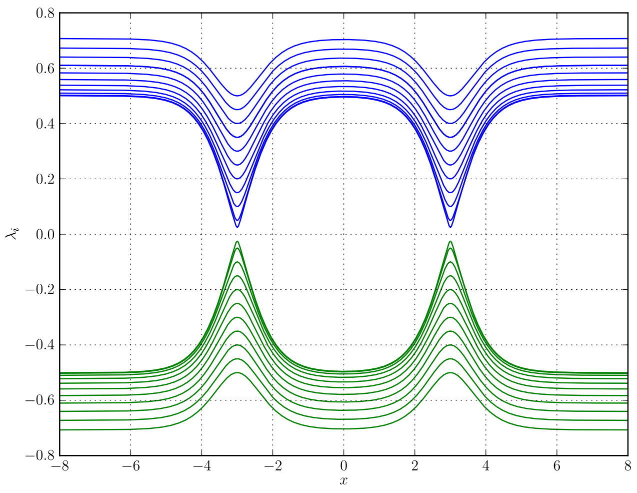


Figure 6.11: The effect of the parameter δ on the energy levels.

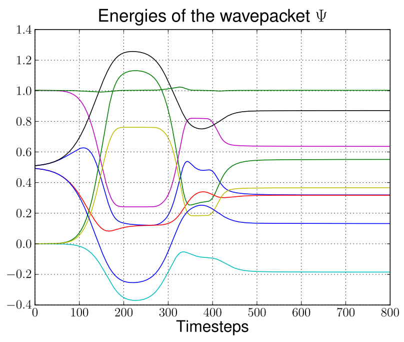
on figure 6.13 and we can see how most of the packet jumps over to the lower level at the first crossing and back to the upper at the second one. For big δ the packet does almost not react to the lower level's bumps. These results were obtained by the operator splitting ansatz which works very well here.

For this potential the wavepacket based algorithm breaks down as soon as the packet arrives at the second crossing for yet unknown reasons. This is true at least for small values of δ. For big δ the time evolution gets better and better. But we can not resolve the interesting details for small ε and δ. We used a basis of size $K = 64$. One might think that the basis is just too small but other tests with $K = 128$ and even $K = 256$ showed the same issues. The algorithm broke down just a few timesteps later. Hence we can conclude that the algorithm does not work in this configuration for yet unknown reasons.

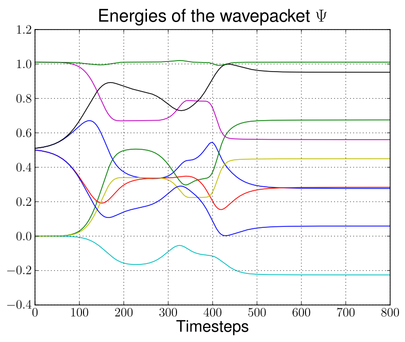
6.4 A potential with three energy levels

Finally we want to look at a potential with three energy levels. This also tests the numerical abilities of our code as we can not do analytical calculations for this matrix and the implementation falls back to pure numerical algorithms. First of all, this is the matrix we use:

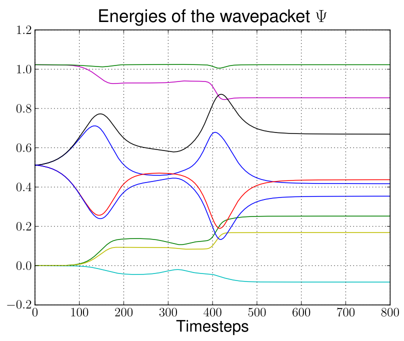
$$V(x) := \begin{pmatrix} \tanh(x+\sigma) + \tanh(x-\sigma) & \delta_1 & \delta_2 \\ \delta_1 & -\tanh(x+\sigma) & 0 \\ \delta_2 & 0 & 1 - \tanh(x-\sigma) \end{pmatrix} . \tag{6.7}$$



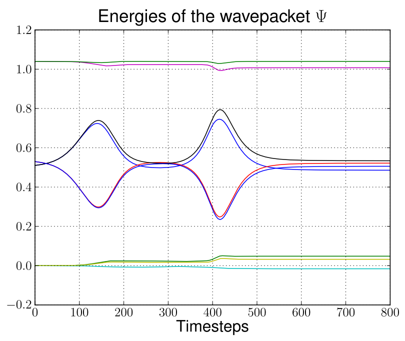
(a)



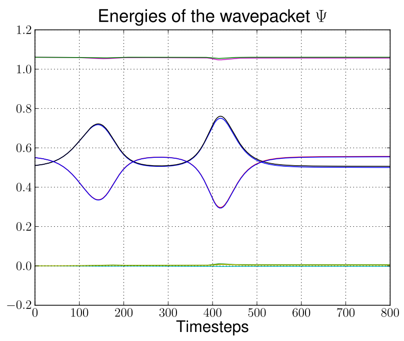
(b)



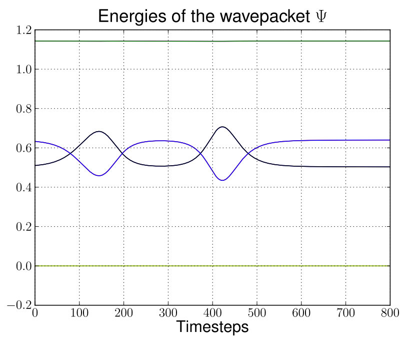
(c)



(d)

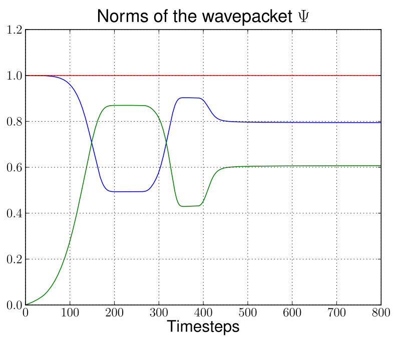


(e)

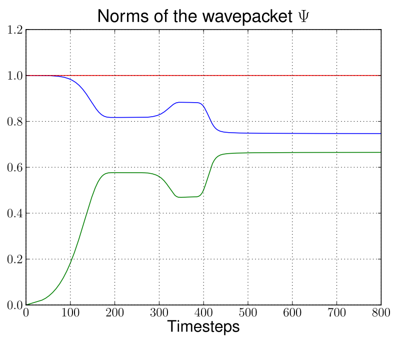


(f)

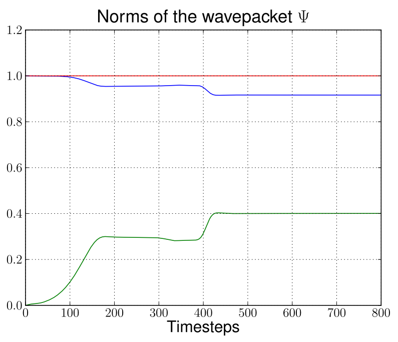
Figure 6.12: Plots of the energies of the wavepacket's individual components Φ_i. These results were obtained by the operator splitting method. (The legend for these figures is shown in 6.3f) (a) $\varepsilon = 0.2$ and $\delta = 0.5\varepsilon$ (b) $\varepsilon = 0.2$ and $\delta = 1.0\varepsilon$ (c) $\varepsilon = 0.2$ and $\delta = 1.5\varepsilon$ (d) $\varepsilon = 0.2$ and $\delta = 2.0\varepsilon$ (e) $\varepsilon = 0.2$ and $\delta = 2.5\varepsilon$ (f) $\varepsilon = 0.2$ and $\delta = 4.0\varepsilon$



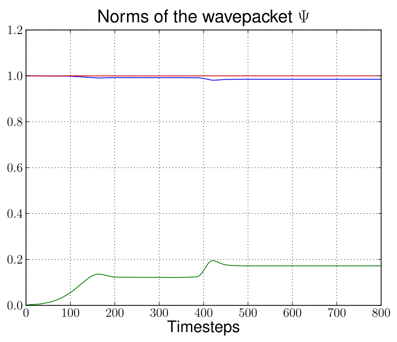
(a)



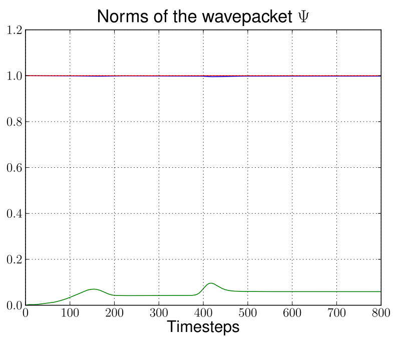
(b)



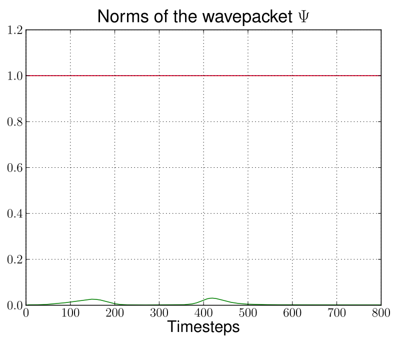
(c)



(d)

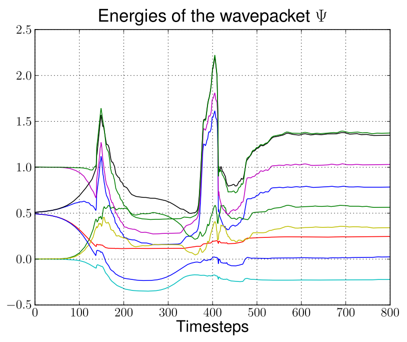


(e)

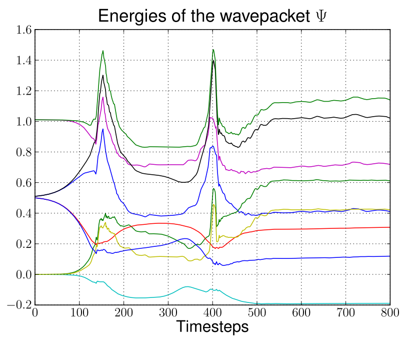


(f)

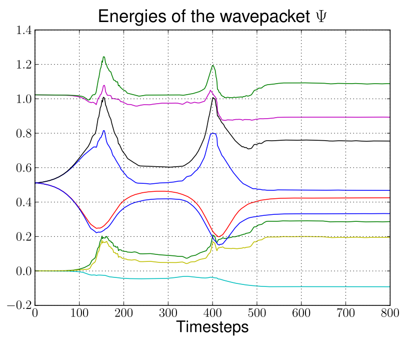
Figure 6.13: Plots of the norms of the wavepacket's individual components Φ_i. These results were obtained by the operator splitting method. (The legend for these figures is shown in 6.4f) (a) $\varepsilon = 0.2$ and $\delta = 0.5\varepsilon$ (b) $\varepsilon = 0.2$ and $\delta = 1.0\varepsilon$ (c) $\varepsilon = 0.2$ and $\delta = 1.5\varepsilon$ (d) $\varepsilon = 0.2$ and $\delta = 2.0\varepsilon$ (e) $\varepsilon = 0.2$ and $\delta = 2.5\varepsilon$ (f) $\varepsilon = 0.2$ and $\delta = 4.0\varepsilon$



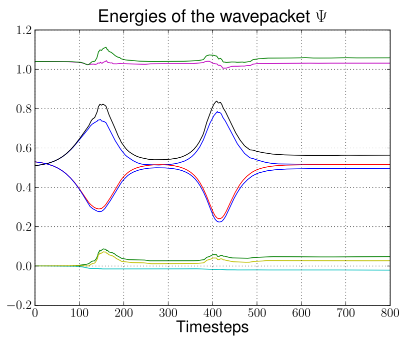
(a)



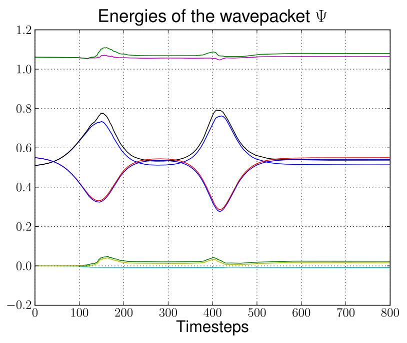
(b)



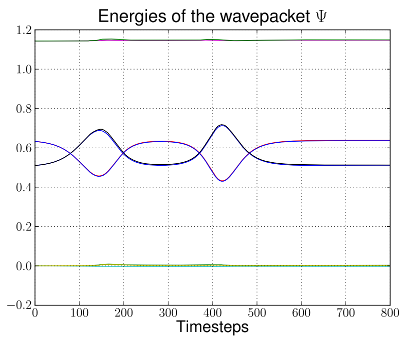
(c)



(d)



(e)



(f)

Figure 6.14: Plots of the energies of the wavepacket's individual components Φ_i. These results were obtained by propagating wavepackets. (The legend for these figures is shown in 6.3f) (a) $\varepsilon = 0.2$ and $\delta = 0.5\varepsilon$ (b) $\varepsilon = 0.2$ and $\delta = 1.0\varepsilon$ (c) $\varepsilon = 0.2$ and $\delta = 1.5\varepsilon$ (d) $\varepsilon = 0.2$ and $\delta = 2.0\varepsilon$ (e) $\varepsilon = 0.2$ and $\delta = 2.5\varepsilon$ (f) $\varepsilon = 0.2$ and $\delta = 4.0\varepsilon$

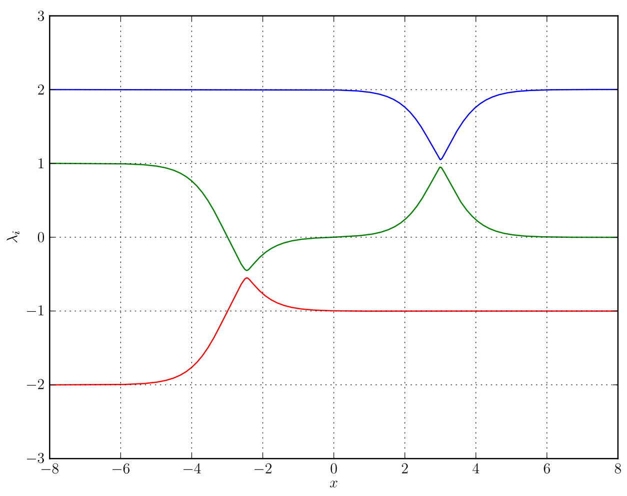


Figure 6.15: Plot of the energy levels of the potential given by equation (6.7). The parameters δ_i equal 0.05.

For this potential we can no longer give an analytical closed form expression for the eigenvalues. Hence we use numerical eigenvalue calculation. The procedure is very similar to what we sketched in figure 1.2. Figure 6.15 shows all three energy levels $\lambda_0(x)$, $\lambda_1(x)$, $\lambda_2(x)$ for the parameters $\delta_1 = \delta_2 = \delta$ set to 0.05.

In this asymmetric energy landscape we expect a rich set of different transitions between all levels. Figure 6.17 shows the energy curves for a Gaussian wavepacket in different initial situations entering the potential from the left or the right. The simulations are all done with the operator splitting method. A reason for this is that the wavepackets break down already for the much simpler case presented in 6.14.

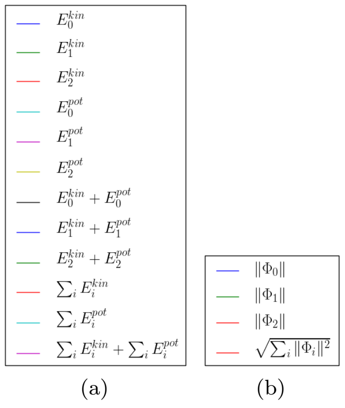
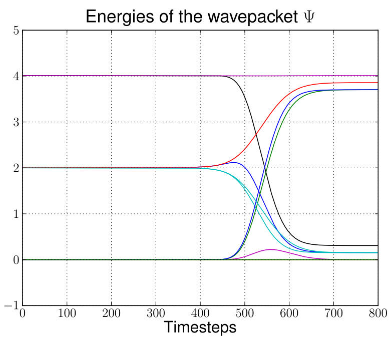
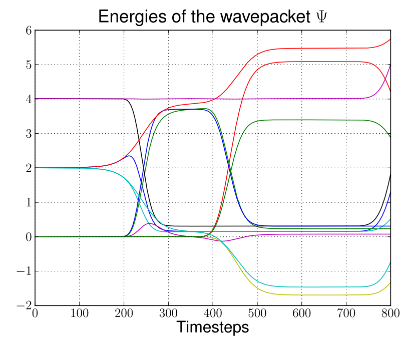


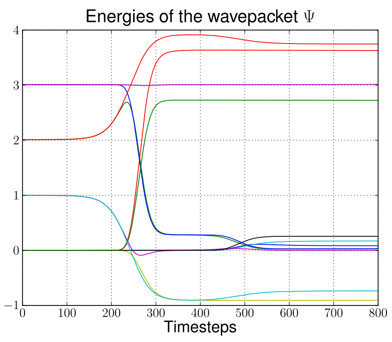
Figure 6.16: Legends of the figures 6.17 and 6.18.



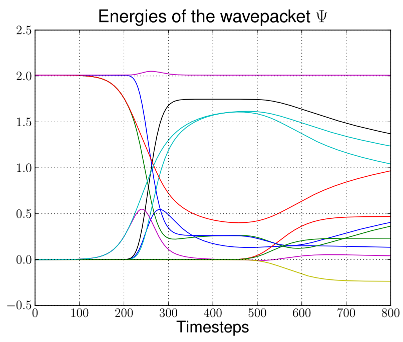
(a)



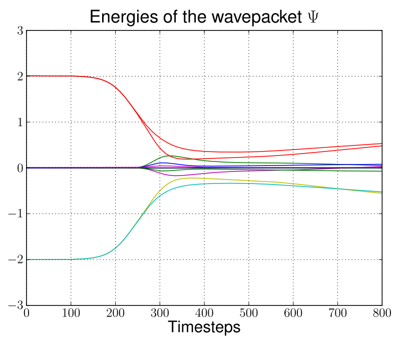
(b)



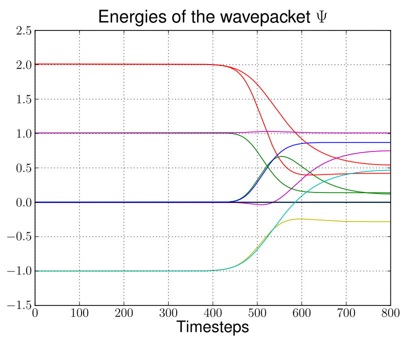
(c)



(d)



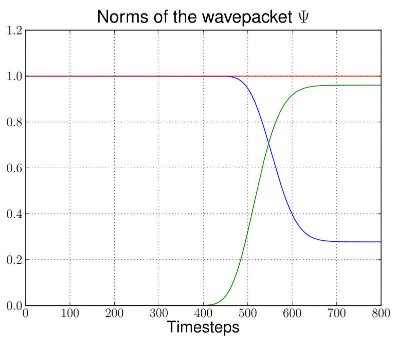
(e)



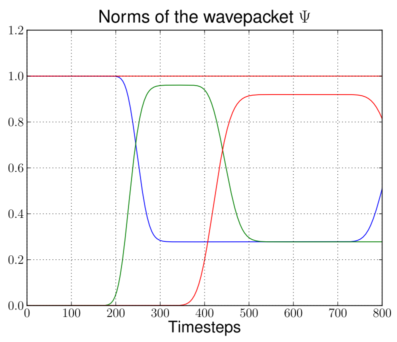
(f)

Figure 6.17: Plots of the energies of the wavepacket's individual components Φ_i. (The legend for these figure is shown in 6.16a) (a) A wavepacket $|\Psi\rangle$ on the upper most level coming from the left. (b) A wavepacket $|\Psi\rangle$ on the upper most level coming from the right. (c) A wavepacket $|\Psi\rangle$ on the middle level coming from the left. (d) A wavepacket $|\Psi\rangle$ on the middle level coming from the right. (e) A wavepacket $|\Psi\rangle$ on the lowest level coming from the left. (f) A wavepacket $|\Psi\rangle$ on the lowest level coming from the right.

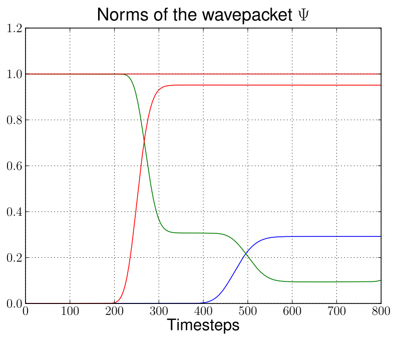
The figures in table 6.18 show the evolution of the norms of all components Φ_i of $|\Psi\rangle$. From these plots we can estimate the transitions that take place. The plots correspond to the ones in figure 6.17.



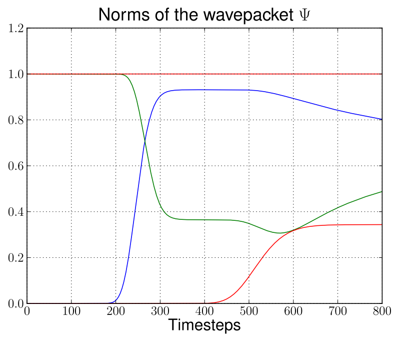
(a)



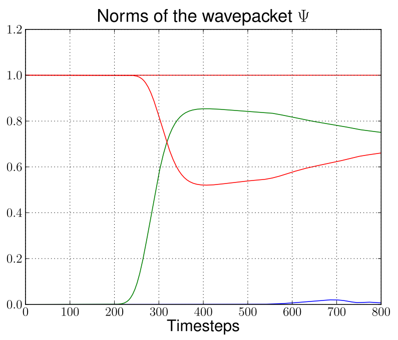
(b)



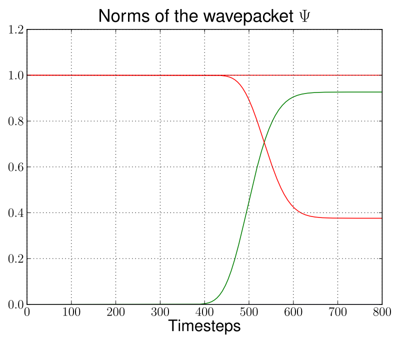
(c)



(d)



(e)



(f)

Figure 6.18: Plots of the norms of the wavepacket's individual components Φ_i. (The legend for these figure is shown in 6.16b) (a) A wavepacket $|\Psi\rangle$ on the upper most level coming from the left. (b) A wavepacket $|\Psi\rangle$ on the upper most level coming from the right. (c) A wavepacket $|\Psi\rangle$ on the middle level coming from the left. (d) A wavepacket $|\Psi\rangle$ on the middle level coming from the right. (e) A wavepacket $|\Psi\rangle$ on the lowest level coming from the left. (f) A wavepacket $|\Psi\rangle$ on the lowest level coming from the right.

Appendix A

Code documentation

In this chapter we describe structure of the source code[1].

The remaining sections contain a description of all classes and their member methods and instance variables. For the implementation details we refer to the source code.

[1] A recent version of the source code can be found at:
`http://n.ethz.ch/~raoulb/research/bachelor_thesis/src/`

A.1 Class MatrixPotential

Subclasses: MatrixPotential1S, MatrixPotential2S, MatrixPotentialMS

This class represents a potential $V(x)$. The potential is given as an analytical expression. Some calculations with the potential are supported. For example calculation of eigenvalues and exponentials and numerical evaluation. Further, there are methods for splitting the potential into a Taylor expansion and for basis transformations between canonical and eigenbasis.

A.1.1 Methods

__init__(*self*)

Create a new *MatrixPotential* instance for a given potential matrix $V(x)$.

Raises

`NotImplementedError` This is an abstract base class.

__str__(*self*)

Put the number of components and the analytical expression (the matrix) into a printable string.

Raises

`NotImplementedError` This is an abstract base class.

get_number_components(*self*)

Return Value

The number N of components the potential supports.

Raises

`NotImplementedError` This is an abstract base class.

evaluate_at(*self*, *nodes*, *component*=`None`)

Evaluate the potential matrix elementwise at some given grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the potential at.
`component:`	The component $V_{i,j}$ that gets evaluated or *None* to evaluate all.

Raises

`NotImplementedError` This is an abstract base class.

calculate_eigenvalues(*self*)

Calculate the eigenvalues $\lambda_i(x)$ of the potential $V(x)$.

Raises

`NotImplementedError`	This is an abstract base class.

evaluate_eigenvalues_at(*self*, *nodes*, *diagonal_component*=`None`)

Evaluate the eigenvalues $\lambda_i(x)$ at some grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the eigenvalues at.
`diagonal_component:`	The index i of the eigenvalue λ_i that gets evaluated or *None* to evaluate all.

Raises

`NotImplementedError`	This is an abstract base class.

calculate_eigenvectors(*self*)

Calculate the eigenvectors $\nu_i(x)$ of the potential $V(x)$.

Raises

`NotImplementedError`	This is an abstract base class.

evaluate_eigenvectors_at(*self*, *nodes*)

Evaluate the eigenvectors $\nu_i(x)$ at some grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the eigenvectors at.

Raises

`NotImplementedError`	This is an abstract base class.

project_to_eigen(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the canonical basis to the eigenbasis of the potential.

Parameters

`nodes:`	The grid nodes γ for the pointwise transformation.
`values:`	The list of vectors φ_i containing the values we want to transform.
`basis:`	A list of basisvectors ν_i. Allows to use this function for external data, similar to a static function.

Raises

`NotImplementedError`	This is an abstract base class.

project_to_canonical(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the potential's eigenbasis to the canonical basis.

Parameters

`nodes:`	The grid nodes γ for the pointwise transformation.
`values:`	The list of vectors φ_i containing the values we want to transform.
`basis:`	A list of basis vectors ν_i. Allows to use this function for external data, similar to a static function.

Raises

`NotImplementedError` This is an abstract base class.

calculate_exponential(*self*, *factor*=`1`)

Calculate the matrix exponential $E = \exp(\alpha M)$.

Parameters

`factor:` A prefactor α in the exponential.

Raises

`NotImplementedError` This is an abstract base class.

evaluate_exponential_at(*self*, *nodes*)

Evaluate the exponential of the potential matrix V at some grid nodes γ.

Parameters

`nodes:` The grid nodes γ we want to evaluate the exponential at.

Raises

`NotImplementedError` This is an abstract base class.

calculate_jacobian(*self*)

Calculate the Jacobian matrix for each component $V_{i,j}$ of the potential. For potentials which depend only one variable x, this equals the first derivative.

Raises

`NotImplementedError` This is an abstract base class.

evaluate_jacobian_at(*self*, *nodes*, *component*=`None`)

Evaluate the Jacobian at some grid nodes γ for each component $V_{i,j}$ of the potential.

Parameters

`nodes:`	The grid nodes γ the Jacobian gets evaluated at.
`component:`	The index tuple (i, j) that specifies the potential's entry of which the Jacobian is evaluated. (Defaults to *None* to evaluate all)

Raises

`NotImplementedError` This is an abstract base class.

calculate_hessian(*self*)

Calculate the Hessian matrix for each component $V_{i,j}$ of the potential. For potentials which depend only one variable x, this equals the second derivative.

Raises

`NotImplementedError` This is an abstract base class.

evaluate_hessian_at(*self*, *nodes*, *component*=`None`)

Evaluate the Hessian at some grid nodes γ for each component $V_{i,j}$ of the potential.

Parameters

`nodes:` The grid nodes γ the Hessian gets evaluated at.

`component:` The index tuple (i, j) that specifies the potential's entry of which the Hessian is evaluated. (Or *None* to evaluate all)

Raises

`NotImplementedError` This is an abstract base class.

calculate_local_quadratic(*self*, *diagonal_component*=`None`)

Calculate the local quadratic approximation matrix U of the potential's eigenvalues in Λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component:` Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i.

Raises

`NotImplementedError` This is an abstract base class.

evaluate_local_quadratic_at(*self*, *nodes*)

Numerically evaluate the local quadratic approximation matrix U of the potential's eigenvalues in Λ at the given grid nodes γ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`nodes:` The grid nodes γ we want to evaluate the quadratic approximation at.

Raises

`NotImplementedError` This is an abstract base class.

calculate_local_remainder(*self*, *diagonal_component*=`0`)

Calculate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalue matrix Λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component`:	Specifies the index χ of the leading component λ_χ.

Raises

`NotImplementedError`	This is an abstract base class.

evaluate_local_remainder_at(*self*, *position*, *nodes*, *component*=`None`)

Numerically evaluate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalues in Λ at the given nodes γ. This function is used for the homogeneous and the inhomogeneous case and just evaluates the remainder matrix W.

Parameters

`position`:	The point q where the Taylor series is computed.
`nodes`:	The grid nodes γ we want to evaluate the potential at.
`component`:	The component (i, j) of the remainder matrix W that is evaluated.

Raises

`NotImplementedError`	This is an abstract base class.

calculate_local_quadratic_multi(*self*)

Calculate the local quadratic approximation matrix U of all the potential's eigenvalues in Λ. This function is used for the inhomogeneous case.

Raises

`NotImplementedError`	This is an abstract base class.

evaluate_local_quadratic_multi_at(*self*, *nodes*, *component*=`None`)

Numerically evaluate the local quadratic approximation matrix U of the potential's eigenvalues in Λ at the given grid nodes γ. This function is used for the inhomogeneous case.

Parameters

`nodes`:	The grid nodes γ we want to evaluate the quadratic approximation at.
`component`:	The component (i, j) of the quadratic approximation matrix U that is evaluated.

Raises

`NotImplementedError`	This is an abstract base class.

calculate_local_remainder_multi(*self*)

Calculate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalue matrix Λ. This function is used for the inhomogeneous case.

Raises

`NotImplementedError` This is an abstract base class.

A.2 Class MatrixPotential1S

MatrixPotential ─┐
  MatrixPotential1S

This class represents a scalar potential $V(x)$. The potential is given as an analytical 1×1 matrix expression. Some symbolic calculations with the potential are supported. For example calculation of eigenvalues and exponentials and numerical evaluation. Further, there are methods for splitting the potential into a Taylor expansion and for basis transformations between canonical and eigenbasis.

A.2.1 Methods

__init__(*self*, *expression*)

Create a new *MatrixPotential1S* instance for a given potential matrix $V(x)$.

Parameters

`expression:` An expression representing the potential.

Overrides: MatrixPotential.__init__

__str__(*self*)

Put the number of components and the analytical expression (the matrix) into a printable string.

Overrides: MatrixPotential.__str__

get_number_components(*self*)

Return Value

The number N of components the potential supports. In the one dimensional case, it's just 1.

Overrides: MatrixPotential.get_number_components

evaluate_at(*self*, *nodes*, *component*=`0`)

Evaluate the potential matrix elementwise at some given grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the potential at.
`component:`	The component $V_{i,j}$ that gets evaluated or *None* to evaluate all.

Return Value

A list with the single entry evaluated at the nodes.

Overrides: MatrixPotential.evaluate_at

calculate_eigenvalues(*self*)

Calculate the eigenvalue $\lambda_0(x)$ of the potential $V(x)$. In the scalar case this is just the matrix entry $V_{0,0}$.

Note: Note: the eigenvalues are memoized for later reuse.

Overrides: MatrixPotential.calculate_eigenvalues

evaluate_eigenvalues_at(*self*, *nodes*, *diagonal_component*=`None`)

Evaluate the eigenvalue $\lambda_0(x)$ at some grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the eigenvalue at.
`diagonal_component:`	Dummy parameter that has no effect here.

Return Value

A list with the single eigenvalue evaluated at the nodes.

Overrides: MatrixPotential.evaluate_eigenvalues_at

calculate_eigenvectors(*self*)

Calculate the eigenvector $\nu_0(x)$ of the potential $V(x)$. In the scalar case this is just the value 1.

Note: The eigenvectors are memoized for later reuse.

Overrides: MatrixPotential.calculate_eigenvectors

evaluate_eigenvectors_at(*self*, *nodes*)

Evaluate the eigenvector $\nu_0(x)$ at some grid nodes γ.

Parameters

`nodes:` The grid nodes γ we want to evaluate the eigenvector at.

Return Value

A list with the eigenvector evaluated at the given nodes.

Overrides: MatrixPotential.evaluate_eigenvectors_at

project_to_eigen(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the canonical basis to the eigenbasis of the potential.

Parameters

`nodes:` The grid nodes γ for the pointwise transformation.

`values:` The list of vectors φ_i containing the values we want to transform.

`basis:` A list of basisvectors ν_i. Allows to use this function for external data, similar to a static function.

Return Value

This method does nothing and returns the values.

Overrides: MatrixPotential.project_to_eigen

project_to_canonical(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the potential's eigenbasis to the canonical basis.

Parameters

`nodes:` The grid nodes γ for the pointwise transformation.

`values:` The list of vectors φ_i containing the values we want to transform.

`basis:` A list of basis vectors ν_i. Allows to use this function for external data, similar to a static function.

Return Value

This method does nothing and returns the values.

Overrides: MatrixPotential.project_to_canonical

calculate_exponential(*self*, *factor*=`1`)

Calculate the matrix exponential $E = \exp(\alpha M)$. In this case the matrix is of size 1×1 thus the exponential simplifies to the scalar exponential function.

Parameters

`factor:` A prefactor α in the exponential.

Overrides: MatrixPotential.calculate_exponential

evaluate_exponential_at(*self*, *nodes*)

Evaluate the exponential of the potential matrix V at some grid nodes γ.

Parameters

`nodes:` The grid nodes γ we want to evaluate the exponential at.

Return Value

The numerical approximation of the matrix exponential at the given grid nodes.

Overrides: MatrixPotential.evaluate_exponential_at

calculate_jacobian(*self*)

Calculate the Jacobian matrix for the component $V_{0,0}$ of the potential. For potentials which depend only on one variable x, this equals the first derivative.

Overrides: MatrixPotential.calculate_jacobian

evaluate_jacobian_at(*self*, *nodes*, *component*=`None`)

Evaluate the potential's Jacobian at some grid nodes γ.

Parameters

`nodes:` The grid nodes γ the Jacobian gets evaluated at.

`component:` Dummy parameter that has no effect here.

Return Value

The value of the potential's Jacobian at the given nodes.

Overrides: MatrixPotential.evaluate_jacobian_at

calculate_hessian(*self*)

Calculate the Hessian matrix for component $V_{0,0}$ of the potential. For potentials which depend only on one variable x, this equals the second derivative.

Overrides: MatrixPotential.calculate_hessian

evaluate_hessian_at(*self*, *nodes*, *component*=`None`)

Evaluate the potential's Hessian at some grid nodes γ.

Parameters

`nodes:` The grid nodes γ the Hessian gets evaluated at.

`component:` Dummy parameter that has no effect here.

Return Value

The value of the potential's Hessian at the given nodes.

Overrides: MatrixPotential.evaluate_hessian_at

calculate_local_quadratic(*self*, *diagonal_component*=`0`)

Calculate the local quadratic approximation U of the potential's eigenvalue λ.

Parameters

`diagonal_component:` Dummy parameter that has no effect here.

Overrides: MatrixPotential.calculate_local_quadratic

evaluate_local_quadratic_at(*self*, *nodes*)

Numerically evaluate the local quadratic approximation U of the potential's eigenvalue λ at the given grid nodes γ. This function is used for the homogeneous case.

Parameters

`nodes`: The grid nodes γ we want to evaluate the quadratic approximation at.

Return Value

An array containing the values of U at the nodes γ.

Overrides: MatrixPotential.evaluate_local_quadratic_at

calculate_local_remainder(*self*, *diagonal_component*=`0`)

Calculate the non-quadratic remainder W of the quadratic approximation U of the potential's eigenvalue λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component`: Dummy parameter that has no effect here.

Overrides: MatrixPotential.calculate_local_remainder

evaluate_local_remainder_at(*self*, *position*, *nodes*, *component*=`None`)

Numerically evaluate the non-quadratic remainder W of the quadratic approximation U of the potential's eigenvalue λ at the given nodes γ. This function is used for the homogeneous and the inhomogeneous case and just evaluates the remainder W.

Parameters

`position`: The point q where the Taylor series is computed.

`nodes`: The grid nodes γ we want to evaluate the potential at.

`component`: Dummy parameter that has no effect here.

Return Value

A list with a single entry consisting of an array containing the values of W at the nodes γ.

Overrides: MatrixPotential.evaluate_local_remainder_at

calculate_local_quadratic_multi(*self*)

Calculate the local quadratic approximation U of the potential's eigenvalue λ. This function is used for the inhomogeneous case.

Raises

`ValueError` There are no inhomogeneous wavepackets with a single component.

Overrides: MatrixPotential.calculate_local_quadratic_multi

evaluate_local_quadratic_multi_at(*self*, *nodes*, *component*=`None`)

Numerically evaluate the local quadratic approximation U of the potential's eigenvalue λ at the given grid nodes γ. This function is used for the inhomogeneous case.

Parameters

`nodes:` The grid nodes γ we want to evaluate the quadratic approximation at.

`component:` Dummy parameter that has no effect here.

Raises

`ValueError` There are no inhomogeneous wavepackets with a single component.

Overrides: MatrixPotential.evaluate_local_quadratic_multi_at

calculate_local_remainder_multi(*self*)

Calculate the non-quadratic remainder W of the quadratic approximation U of the potential's eigenvalue λ. This function is used for the inhomogeneous case.

Raises

`ValueError` There are no inhomogeneous wavepackets with a single component.

Overrides: MatrixPotential.calculate_local_remainder_multi

A.2.2 Instance Variables

Name	Description
x	The variable x that represents position space.
potential	The matrix of the potential $V(x)$.

A.3 Class MatrixPotential2S

MatrixPotential ─┐
└ **MatrixPotential2S**

This class represents a matrix potential $V(x)$. The potential is given as an analytical 2×2 matrix expression. Some symbolical calculations with the potential are supported. For example calculation of eigenvalues and exponentials and numerical evaluation. Further, there are methods for splitting the potential into a Taylor expansion and for basis transformations between canonical and eigenbasis.

A.3.1 Methods

__init__(*self*, *expression*)

Create a new *MatrixPotential2S* instance for a given potential matrix $V(x)$.

Parameters

`expression`: An expression representing the potential.

Overrides: MatrixPotential.__init__

__str__(*self*)

Put the number of components and the analytical expression (the matrix) into a printable string.

Overrides: MatrixPotential.__str__

---

get_number_components(*self*)

Return Value

The number N of components the potential supports. This is also the size of the matrix. In the current case it's 2.

Overrides: MatrixPotential.get_number_components

evaluate_at(*self*, *nodes*, *component*=`None`, *as_matrix*=`False`)

Evaluate the potential matrix elementwise at some given grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the potential at.
`component:`	The component $V_{i,j}$ that gets evaluated or *None* to evaluate all.
`as_matrix:`	Dummy parameter which has no effect here.

Return Value

A list with the 4 entries evaluated at the nodes.

Overrides: MatrixPotential.evaluate_at

calculate_eigenvalues(*self*)

Calculate the two eigenvalues $\lambda_i\left(x\right)$ of the potential $V\left(x\right)$. We can do this by symbolical calculations. The multiplicities are taken into account.

Note: Note: the eigenvalues are memoized for later reuse.

Overrides: MatrixPotential.calculate_eigenvalues

evaluate_eigenvalues_at(*self*, *nodes*, *component*=`None`, *as_matrix*=`False`)

Evaluate the eigenvalues $\lambda_i\left(x\right)$ at some grid nodes γ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the eigenvalues at.
`component:`	The index i of the eigenvalue λ_i that gets evaluated.
`as_matrix:`	Returns the whole matrix Λ instead of only a list with the eigenvalues λ_i.

Return Value

A sorted list with 2 entries for the two eigenvalues evaluated at the nodes. Or a single value if a component was specified.

Overrides: MatrixPotential.evaluate_eigenvalues_at

calculate_eigenvectors(*self*)

Calculate the two eigenvectors $\nu_i\left(x\right)$ of the potential $V\left(x\right)$. We can do this by symbolical calculations.

Note: The eigenvectors are memoized for later reuse.

Overrides: MatrixPotential.calculate_eigenvectors

evaluate_eigenvectors_at(*self*, *nodes*)

Evaluate the eigenvectors $\nu_i(x)$ at some grid nodes γ.

Parameters

`nodes`: The grid nodes γ we want to evaluate the eigenvectors at.

Return Value

A list with the two eigenvectors evaluated at the given nodes.

Overrides: MatrixPotential.evaluate_eigenvectors_at

project_to_eigen(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the canonical basis to the eigenbasis of the potential.

Parameters

`nodes`: The grid nodes γ for the pointwise transformation.

`values`: The list of vectors φ_i containing the values we want to transform.

`basis`: A list of basisvectors ν_i. Allows to use this function for external data, similar to a static function.

Return Value

Returned is another list containing the projection of the values into the eigenbasis.

Overrides: MatrixPotential.project_to_eigen

project_to_canonical(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the potential's eigenbasis to the canonical basis.

Parameters

`nodes`: The grid nodes γ for the pointwise transformation.

`values`: The list of vectors φ_i containing the values we want to transform.

`basis`: A list of basis vectors ν_i. Allows to use this function for external data, similar to a static function.

Return Value

Returned is another list containing the projection of the values into the eigenbasis.

Overrides: MatrixPotential.project_to_canonical

calculate_exponential(*self*, *factor*=`1`)

Calculate the matrix exponential $E = \exp(\alpha M)$. In this case the matrix is of size 2×2 thus the general exponential can be calculated analytically.

Parameters

`factor`: A prefactor α in the exponential.

Overrides: MatrixPotential.calculate_exponential

evaluate_exponential_at(*self*, *nodes*)

Evaluate the exponential of the potential matrix V at some grid nodes γ.

Parameters

`nodes`: The grid nodes γ we want to evaluate the exponential at.

Return Value

The numerical approximation of the matrix exponential at the given grid nodes.

Overrides: MatrixPotential.evaluate_exponential_at

calculate_jacobian(*self*)

Calculate the Jacobian matrix for each component $V_{i,j}$ of the potential. For potentials which depend only on one variable x, this equals the first derivative.

Overrides: MatrixPotential.calculate_jacobian

evaluate_jacobian_at(*self*, *nodes*, *component*=`None`)

Evaluate the Jacobian at some grid nodes γ for each component $V_{i,j}$ of the potential.

Parameters

`nodes`:	The grid nodes γ the Jacobian gets evaluated at.
`component`:	The index tuple (i, j) that specifies the potential's entry of which the Jacobian is evaluated. (Defaults to *None* to evaluate all)

Return Value

Either a list or a single value depending on the optional parameters.

Overrides: MatrixPotential.evaluate_jacobian_at

calculate_hessian(*self*)

Calculate the Hessian matrix for each component $V_{i,j}$ of the potential. For potentials which depend only on one variable x, this equals the second derivative.

Overrides: MatrixPotential.calculate_hessian

evaluate_hessian_at(*self*, *nodes*, *component*=`None`)

Evaluate the Hessian at some grid nodes γ for each component $V_{i,j}$ of the potential.

Parameters

`nodes:`	The grid nodes γ the Hessian gets evaluated at.
`component:`	The index tuple (i, j) that specifies the potential's entry of which the Hessian is evaluated. (Or *None* to evaluate all)

Return Value

Either a list or a single value depending on the optional parameters.

Overrides: MatrixPotential.evaluate_hessian_at

calculate_local_quadratic(*self*, *diagonal_component*)

Calculate the local quadratic approximation matrix U of the potential's eigenvalues in Λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component:`	Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i.

Overrides: MatrixPotential.calculate_local_quadratic

evaluate_local_quadratic_at(*self*, *nodes*)

Numerically evaluate the local quadratic approximation matrix U of the potential's eigenvalues in Λ at the given grid nodes γ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the quadratic approximation at.

Return Value

A list of arrays containing the values of $U_{i,j}$ at the nodes γ.

Overrides: MatrixPotential.evaluate_local_quadratic_at

calculate_local_remainder(*self*, *diagonal_component*)

Calculate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalue matrix Λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component:`	Specifies the index χ of the leading component λ_χ.

Overrides: MatrixPotential.calculate_local_remainder

evaluate_local_remainder_at(*self*, *position*, *nodes*, *component*=`None`)

Numerically evaluate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalues in Λ at the given nodes γ. This function is used for the homogeneous and the inhomogeneous case and just evaluates the remainder matrix W.

Parameters

`position:`	The point q where the Taylor series is computed.
`nodes:`	The grid nodes γ we want to evaluate the potential at.
`component:`	The component (i, j) of the remainder matrix W that is evaluated.

Return Value

A list with a single entry consisting of an array containing the values of W at the nodes γ.

Overrides: MatrixPotential.evaluate_local_remainder_at

calculate_local_quadratic_multi(*self*)

Calculate the local quadratic approximation matrix U of all the potential's eigenvalues in Λ. This function is used for the inhomogeneous case.

Overrides: MatrixPotential.calculate_local_quadratic_multi

evaluate_local_quadratic_multi_at(*self*, *nodes*, *component*=`None`)

Numerically evaluate the local quadratic approximation matrix U of the potential's eigenvalues in Λ at the given grid nodes γ. This function is used for the inhomogeneous case.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the quadratic approximation at.
`component:`	The component (i, j) of the quadratic approximation matrix U that is evaluated.

Return Value

A list of arrays or a single array containing the values of $U_{i,j}$ at the nodes γ.

Overrides: MatrixPotential.evaluate_local_quadratic_multi_at

calculate_local_remainder_multi(*self*)

Calculate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalue matrix Λ. This function is used for the inhomogeneous case.

Overrides: MatrixPotential.calculate_local_remainder_multi

A.3.2 Instance Variables

Name	**Description**
x	The variable x that represents position space.
potential	The matrix of the potential $V(x)$.

A.4 Class MatrixPotentialMS

MatrixPotential ─┐
　　　　　　　　MatrixPotentialMS

This class represents a matrix potential $V(x)$. The potential is given as an analytical expression with a matrix of size bigger than 2×2. Some calculations with the potential are supported. For example calculation of eigenvalues and exponentials and numerical evaluation. Further, there are methods for splitting the potential into a Taylor expansion and for basis transformations between canonical and eigenbasis. All methods use numerical techniques because symbolical calculations are unfeasible.

A.4.1 Methods

__init__(*self*, *expression*)

Create a new *MatrixPotentialMS* instance for a given potential matrix $V(x)$.

Parameters

`expression`: An expression representing the potential.

Overrides: MatrixPotential.__init__

__str__(*self*)

Put the number of components and the analytical expression (the matrix) into a printable string.

Overrides: MatrixPotential.__str__

get_number_components(*self*)

Return Value

The number N of components the potential supports. This is also the size of the matrix.

Overrides: MatrixPotential.get_number_components

evaluate_at(*self*, *nodes*, *component*=`None`)

Evaluate the potential matrix elementwise at some given grid nodes γ.

Parameters

`nodes`: The grid nodes γ we want to evaluate the potential at.

`component`: The component $V_{i,j}$ that gets evaluated or *None* to evaluate all.

Return Value

A list with the N^2 entries evaluated at the nodes.

Overrides: MatrixPotential.evaluate_at

calculate_eigenvalues(*self*)

Calculate the eigenvalues $\lambda_i(x)$ of the potential $V(x)$. We do the calculations with numerical tools. The multiplicities are taken into account.

Note: Note: the eigenvalues are memoized for later reuse.

Overrides: MatrixPotential.calculate_eigenvalues

evaluate_eigenvalues_at(*self*, *nodes*, *component*=`None`)

Evaluate the eigenvalues $\lambda_i(x)$ at some grid nodes γ.

Parameters

`nodes`: The grid nodes γ we want to evaluate the eigenvalues at.

`component`: The index i of the eigenvalue λ_i that gets evaluated.

Return Value

A sorted list with N entries for all the eigenvalues evaluated at the nodes. Or a single value if a component was specified.

Overrides: MatrixPotential.evaluate_eigenvalues_at

calculate_eigenvectors(*self*)

Calculate the two eigenvectors $\nu_i(x)$ of the potential $V(x)$. We do the calculations with numerical tools.

Note: The eigenvectors are memoized for later reuse.

Overrides: MatrixPotential.calculate_eigenvectors

evaluate_eigenvectors_at(*self*, *nodes*)

Evaluate the eigenvectors $\nu_i(x)$ at some grid nodes γ.

Parameters

`nodes`: The grid nodes γ we want to evaluate the eigenvectors at.

Return Value

A list with the N eigenvectors evaluated at the given nodes.

Overrides: MatrixPotential.evaluate_eigenvectors_at

project_to_eigen(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the canonical basis to the eigenbasis of the potential.

Parameters

`nodes:`	The grid nodes γ for the pointwise transformation.
`values:`	The list of vectors φ_i containing the values we want to transform.
`basis:`	A list of basisvectors ν_i. Allows to use this function for external data, similar to a static function.

Return Value

Returned is another list containing the projection of the values into the eigenbasis.

Overrides: MatrixPotential.project_to_eigen

project_to_canonical(*self*, *nodes*, *values*, *basis*=`None`)

Project a given vector from the potential's eigenbasis to the canonical basis.

Parameters

`nodes:`	The grid nodes γ for the pointwise transformation.
`values:`	The list of vectors φ_i containing the values we want to transform.
`basis:`	A list of basis vectors ν_i. Allows to use this function for external data, similar to a static function.

Return Value

Returned is another list containing the projection of the values into the eigenbasis.

Overrides: MatrixPotential.project_to_canonical

calculate_exponential(*self*, *factor*=`1`)

Calculate the matrix exponential $E = \exp(\alpha M)$. In the case where the matrix is of size bigger than 2×2 symbolical calculations become unfeasible. We use numerical approximations to determine the matrix exponential.

Parameters

`factor:`	A prefactor α in the exponential.

Overrides: MatrixPotential.calculate_exponential

evaluate_exponential_at(*self*, *nodes*)

Evaluate the exponential of the potential matrix V at some grid nodes γ. For matrices of size > 2 we do completely numerical exponentiation.

Parameters

`nodes`: The grid nodes γ we want to evaluate the exponential at.

Return Value

The numerical approximation of the matrix exponential at the given grid nodes.

Overrides: MatrixPotential.evaluate_exponential_at

calculate_jacobian(*self*)

Calculate the Jacobian matrix for each component $V_{i,j}$ of the potential. For potentials which depend only one variable x, this equals the first derivative.

Overrides: MatrixPotential.calculate_jacobian

evaluate_jacobian_at(*self*, *nodes*, *component*=`None`)

Evaluate the Jacobian at some grid nodes γ for each component $V_{i,j}$ of the potential.

Parameters

`nodes`: The grid nodes γ the Jacobian gets evaluated at.

`component`: The index tuple (i,j) that specifies the potential's entry of which the Jacobian is evaluated. (Defaults to *None* to evaluate all)

Return Value

Either a list or a single value depending on the optional parameters.

Overrides: MatrixPotential.evaluate_jacobian_at

calculate_hessian(*self*)

Calculate the Hessian matrix for each component $V_{i,j}$ of the potential. For potentials which depend only one variable x, this equals the second derivative.

Overrides: MatrixPotential.calculate_hessian

evaluate_hessian_at(*self*, *nodes*, *component*=`None`)

Evaluate the Hessian at some grid nodes γ for each component $V_{i,j}$ of the potential.

Parameters

`nodes:`	The grid nodes γ the Hessian gets evaluated at.
`component:`	The index tuple (i, j) that specifies the potential's entry of which the Hessian is evaluated. (Or *None* to evaluate all)

Return Value

Either a list or a single value depending on the optional parameters.

Overrides: MatrixPotential.evaluate_hessian_at

calculate_local_quadratic(*self*, *diagonal_component*)

Calculate the local quadratic approximation matrix U of the potential's eigenvalues in Λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component:`	Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i.

Overrides: MatrixPotential.calculate_local_quadratic

evaluate_local_quadratic_at(*self*, *nodes*)

Numerically evaluate the local quadratic approximation matrix U of the potential's eigenvalues in Λ at the given grid nodes γ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the quadratic approximation at.

Return Value

A list of arrays containing the values of $U_{i,j}$ at the nodes γ.

Overrides: MatrixPotential.evaluate_local_quadratic_at

calculate_local_remainder(*self*, *diagonal_component*)

Calculate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalue matrix Λ. This function is used for the homogeneous case and takes into account the leading component χ.

Parameters

`diagonal_component:`	Specifies the index χ of the leading component λ_χ.

Overrides: MatrixPotential.calculate_local_remainder

evaluate_local_remainder_at(*self*, *position*, *nodes*, *component*=`None`)

Numerically evaluate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalues in Λ at the given nodes γ. This function is used for the homogeneous and the inhomogeneous case and just evaluates the remainder matrix W.

Parameters

`position:`	The point q where the Taylor series is computed.
`nodes:`	The grid nodes γ we want to evaluate the potential at.
`component:`	The component (i, j) of the remainder matrix W that is evaluated.

Return Value

A list with a single entry consisting of an array containing the values of W at the nodes γ.

Overrides: MatrixPotential.evaluate_local_remainder_at

calculate_local_quadratic_multi(*self*)

Calculate the local quadratic approximation matrix U of all the potential's eigenvalues in Λ. This function is used for the inhomogeneous case.

Overrides: MatrixPotential.calculate_local_quadratic_multi

evaluate_local_quadratic_multi_at(*self*, *nodes*, *component*=`None`)

Numerically evaluate the local quadratic approximation matrix U of the potential's eigenvalues in Λ at the given grid nodes γ. This function is used for the inhomogeneous case.

Parameters

`nodes:`	The grid nodes γ we want to evaluate the quadratic approximation at.
`component:`	The component (i, j) of the quadratic approximation matrix U that is evaluated.

Return Value

A list of arrays or a single array containing the values of $U_{i,j}$ at the nodes γ.

Overrides: MatrixPotential.evaluate_local_quadratic_multi_at

calculate_local_remainder_multi(*self*)

Calculate the non-quadratic remainder matrix W of the quadratic approximation matrix U of the potential's eigenvalue matrix Λ. This function is used for the inhomogeneous case.

Overrides: MatrixPotential.calculate_local_remainder_multi

A.4.2 Instance Variables

Name	**Description**
x	The variable x that represents position space.
potential	The matrix of the potential $V(x)$.

A.5 Class PotentialFactory

A factory for *MatrixPotential* instances. We decide which subclass of the abstract base class *MatrixPotential* to instantiate according to the size of the potential's matrix. For a 1×1 matrix we can use the class *MatrixPotential1S* which implements simplified scalar symbolic calculations. In the case of a 2×2 matrix we use the class *MatrixPotential2S* that implements the full symbolic calculations for matrices. And for matrices of size bigger than 2×2 symbolic calculations are unfeasible and we have to fall back to pure numerical methods implemented in *MatrixPotentialMS*.

A.5.1 Methods

create_potential(*potential_expression*)

Static method that creates a *MatrixPotential* instance and decides which subclass to instantiate depending on the given potential expression.

Parameters

`potential_expression`: The symbolic potential matrix given.

Return Value

An adequate *MatrixPotential* instance.

Raises

`ValueError` If the potential matrix is not square.

A.6 Class HagedornWavepacket

This class represents homogeneous vector valued wavepackets $|\Psi\rangle$.

A.6.1 Methods

__init__(*self*, *number_components*, *basis_size*)

Initialize the *HagedornWavepacket* object that represents $|\Psi\rangle$.

Parameters

`number_components:`	The number N of components $\Phi_0, \ldots, \Phi_{N-1}$ the vector Ψ has got.
`basis_size:`	The number K of basis functions $\phi_0, \ldots, \phi_{K-1}$.

Raises

`ValueError` For $N < 1$ or $K < 2$.

__str__(*self*)

Return Value

A string describing the Hagedorn wavepacket.

get_number_components(*self*)

Return Value

The number N of components the wavepacket Ψ has.

set_coefficients(*self*, *values*, *component*=`None`)

Update the coefficients c of Ψ.

Parameters

`values:`	The new values of the coefficients c^i of Φ_i.
`component:`	The index i of the component we want to update with new coefficients.

Raises

`ValueError` For invalid indices i.

Note: This function can either set new coefficients for a single component Φ_i only if the *component* attribute is set or for all components simultaneously if *values* is a list of arrays.

set_coefficient(*self*, *component*, *index*, *value*)

Set a single coefficient c^i_k of the specified component Φ_i of $|\Psi\rangle$.

Parameters

`component:`	The index i of the component Φ_i we want to update.
`index:`	The index k of the coefficient c^i_k we want to update.
`value:`	The new value of the coefficient c^i_k.

Raises

`ValueError`	For invalid indices i or k.

get_coefficients(*self*, *component*=`None`)

Returns the coefficients c^i for some components Φ_i of $|\Psi\rangle$.

Parameters

`component:`	The index i of the coefficients c^i we want to get.

Return Value

The coefficients c^i either for all components Φ_i or for a specified one.

get_coefficient_vector(*self*)

Return Value

The coefficients c^i of all components Φ_i as a single long column vector.

set_coefficient_vector(*self*, *vector*)

Set the coefficients for all components Φ_i simultaneously.

Parameters

`vector:`	The coefficients of all components as a single long column vector.

get_parameters(*self*)

Get the Hagedorn parameters Π of the wavepacket Ψ.

Return Value

The Hagedorn parameters P, Q, S, p, q of Ψ in this order.

set_parameters(*self*, *parameters*)

Set the Hagedorn parameters Π of the wavepacket Ψ.

Parameters

`parameters:`	The Hagedorn parameters P, Q, S, p, q of Ψ in this order.

set_quadrator(*self*, *quadrator*)

Set the *Quadrator* instance used for quadrature.

Parameters

`quadrator:` The new *Quadrator* instance. May be *None* to use a dafault one of order $K + 4$.

evaluate_base_at(*self*, *nodes*)

Evaluate the Hagedorn functions ϕ_k recursively at the given nodes γ.

Parameters

`nodes:` The nodes γ at which the Hagedorn functions are evaluated.

Return Value

Returns a two dimensional array H where the entry $H[k, i]$ is the value of the k-th Hagedorn function evaluated at the node i.

evaluate_at(*self*, *nodes*, *component*=`None`, *prefactor*=`False`)

Evaluate the Hagedorn wavepacket Ψ at the given nodes γ.

Parameters

`nodes:` The nodes γ at which the Hagedorn wavepacket gets evaluated.

`component:` The index i of a single component Φ_i to evaluate. (Defaults to for evaluating all components.)

`prefactor:` Whether to include a factor of $\det(Q)^{-\frac{1}{2}}$.

Return Value

A list of arrays or a single array containing the values of the Φ_i at the nodes γ.

quadrate(*self*, *function*, *summed*=`False`)

Performs the quadrature of $\langle \Psi \mid f \mid \Psi \rangle$ for a general f.

Parameters

`function:` A real-valued function $f(x) : R \to R^{N \times N}$

`summed:` Whether to sum up the individual integrals $\langle \Phi_i \mid f_{i,j} \mid \Phi_j \rangle$.

Return Value

The value of $\langle \Psi \mid f \mid \Psi \rangle$. This is either a scalar value or a list of N^2 scalar elements.

matrix(*self*, *function*)

Calculate the matrix representation of $\langle \Psi \mid f \mid \Psi \rangle$.

Parameters

`function:` A function with two arguments $f : (q, x) \to \mathbb{R}$.

Return Value

A square matrix of size $NK \times NK$.

get_norm(*self*, *component*=`None`, *summed*=`False`)

Calculate the L^2 norm of the wavepacket $|\Psi\rangle$.

Parameters

`component:` The component Φ_i of which the norm is calculated.

`summed:` Whether to sum up the norms of the individual components Φ_i.

Return Value

A list containing the norms of all components Φ_i or the overall norm of Ψ.

potential_energy(*self*, *potential*, *summed*=`False`)

Calculate the potential energy $\langle \Psi \mid V \mid \Psi \rangle$ of the wavepacket componentwise.

Parameters

`potential:` The potential energy operator V as function.

`summed:` Whether to sum up the individual integrals $\langle \Phi_i \mid V_{i,j} \mid \Phi_j \rangle$.

Return Value

The potential energy of the wavepacket's components Φ_i or the overall potential energy of Ψ.

kinetic_energy(*self*, *summed*=`False`)

Calculate the kinetic energy $\langle \Psi \mid T \mid \Psi \rangle$ of the wavepacket componentwise.

Parameters

`summed:` Whether to sum up the individual integrals $\langle \Phi_i \mid T_{i,j} \mid \Phi_j \rangle$.

Return Value

The kinetic energy of the wavepacket's components Φ_i or the overall kinetic energy of Ψ.

grady(*self*, *component*)

Calculate the effect of $-i\varepsilon^2 \frac{\partial}{\partial x}$ on a component Φ_i of the Hagedorn wavepacket Ψ.

Parameters

`component`: The index i of the component Φ_i on which we apply the above operator.

Return Value

The modified coefficients.

project_to_canonical(*self*, *potential*)

Project the Hagedorn wavepacket into the canonical basis.

Parameters

`potential`: The potential V whose eigenvectors ν_l are used for the transformation.

Note: This function is expensive and destructive! It modifies the coefficients of the *self* instance.

project_to_eigen(*self*, *potential*)

Project the Hagedorn wavepacket into the eigenbasis of a given potential V.

Parameters

`potential`: The potential V whose eigenvectors ν_l are used for the transformation.

Note: This function is expensive and destructive! It modifies the coefficients of the *self* instance.

A.6.2 Instance Variables

Name	Description
number_components	Number of components Φ_i the wavepacket $\|\Psi\rangle$ has got.
basis_size	Size of the basis from which we construct the wavepacket.
coefficients	The coefficients c^i of the linear combination for each component Φ_k.
quadrator	An object that provides nodes γ and weights ω for Gauss-Hermite quadrature.

A.7 Class HagedornMultiWavepacket

This class represents inhomogeneous vector valued wavepackets $|\Psi\rangle$.

A.7.1 Methods

__init__(*self*, *number_components*, *basis_size*)

Initialize the *HagedornMultiWavepacket* object that represents $|\Psi\rangle$.

Parameters

`number_components:`	The number N of components $\Phi_0, \ldots, \Phi_{N-1}$ the vector Ψ has got.
`basis_size:`	The number K of basis functions $\Phi_0, \ldots, \Phi_{K-1}$.

Raises

`ValueError` For $N < 1$ or $K < 2$.

__str__(*self*)

Return Value

A string describing the Hagedorn wavepacket.

get_number_components(*self*)

Return Value

The number N of components the wavepacket Ψ has.

set_coefficients(*self*, *values*, *component*=`None`)

Update the coefficients c of Ψ.

Parameters

`values:`	The new values of the coefficients c^i of Φ_i.
`component:`	The index i of the component we want to update with new coefficients.

Raises

`ValueError` For invalid indices i.

Note: This function can either set new coefficients for a single component Φ_i only if the *component* attribute is set or for all components simultaneously if *values* is a list of arrays.

set_coefficient(*self*, *component*, *index*, *value*)

Set a single coefficient c^i_k of the specified component Φ_i of $|\Psi\rangle$.

Parameters

`component:`	The index i of the component Φ_i we want to update.
`index:`	The index k of the coefficient c^i_k we want to update.
`value:`	The new value of the coefficient c^i_k.

Raises

`ValueError` For invalid indices i or k.

get_coefficients(*self*, *component*=`None`)

Returns the coefficients c^i for some components Φ_i of $|\Psi\rangle$.

Parameters

`component:` The index i of the coefficients c^i we want to get.

Return Value

The coefficients c^i either for all components Φ_i or for a specified one.

get_coefficient_vector(*self*)

Return Value

The coefficients c^i of all components Φ_i as a single long column vector.

set_coefficient_vector(*self*, *vector*)

Set the coefficients for all components Φ_i simultaneously.

Parameters

`vector:` The coefficients of all components as a single long column vector.

get_parameters(*self*, *component*=`None`)

Get the Hagedorn parameters Π_i of each component Φ_i of the wavepacket Ψ.

Parameters

`component:` The index i of the component whose parameters Π_i we want to get.

Return Value

A list with all the sets Π_i or a single set.

set_parameters(*self*, *parameters*, *component*=`None`)

Set the Hagedorn parameters Π_i of each component Φ_i of the wavepacket Ψ.

Parameters

`parameters:` A list or a single set of Hagedorn parameters.

`component:` The index i of the component whose parameters Π_i we want to update.

set_quadrator(*self*, *quadrator*)

Set the *Quadrator* instance used for quadrature.

Parameters

`quadrator:` The new *Quadrator* instance. May be *None* to use a dafault one of order $K + 4$.

evaluate_base_at(*self*, *nodes*, *component*, *prefactor*=`False`)

Evaluate the Hagedorn functions Φ_k recursively at the given nodes γ.

Parameters

`nodes:` The nodes γ at which the Hagedorn functions are evaluated.

`component:` The index i of the component whose basis functions ϕ_k^i we want to evaluate.

`prefactor:` Whether to include a factor of $\det(Q_i)^{-\frac{1}{2}}$.

Return Value

Returns a two dimensional array H where the entry $H[k, i]$ is the value of the k-th Hagedorn function evaluated at the node i.

evaluate_at(*self*, *nodes*, *component*=`None`, *prefactor*=`False`)

Evaluate the Hagedorn wavepacket Ψ at the given nodes γ.

Parameters

`nodes:` The nodes γ at which the Hagedorn wavepacket gets evaluated.

`component:` The index i of a single component Φ_i to evaluate. (Defaults to *None* for evaluating all components.)

`prefactor:` Whether to include a factor of $\det(Q_i)^{-\frac{1}{2}}$.

Return Value

A list of arrays or a single array containing the values of the Φ_i at the nodes γ.

quadrate(*self*, *function*, *summed*=`False`)

Performs the quadrature of $\langle \Psi \,|\, f \,|\, \Psi \rangle$ for a general f.

Parameters

`function:` A real-valued function $f(x) : R \to R^{N \times N}$

`summed:` Whether to sum up the individual integrals $\langle \Phi_i \,|\, f_{i,j} \,|\, \Phi_j \rangle$.

Return Value

The value of $\langle \Psi \,|\, f \,|\, \Psi \rangle$. This is either a scalar value or a list of N^2 scalar elements.

matrix(*self*, *function*)

Calculate the matrix representation of $\langle \Psi \,|\, f \,|\, \Psi \rangle$.

Parameters

`function:` A function with two arguments $f : (q, x) \to \mathbb{R}$.

Return Value

A square matrix of size $NK \times NK$.

get_norm(*self*, *component*=`None`, *summed*=`False`)

Calculate the L^2 norm of the wavepacket $|\Psi\rangle$.

Parameters

`component:` The component Φ_i of which the norm is calculated.

`summed:` Whether to sum up the norms of the individual components Φ_i.

Return Value

A list containing the norms of all components Φ_i or the overall norm of Ψ.

potential_energy(*self*, *potential*, *summed*=`False`)

Calculate the potential energy $\langle \Psi \,|\, V \,|\, \Psi \rangle$ of the wavepacket componentwise.

Parameters

`potential:` The potential energy operator V as function.

`summed:` Whether to sum up the individual integrals $\langle \Phi_i \,|\, V_{i,j} \,|\, \Phi_j \rangle$.

Return Value

The potential energy of the wavepacket's components Φ_i or the overall potential energy of Ψ.

kinetic_energy(*self*, *summed*=`False`)

Calculate the kinetic energy $\langle \Psi \,|\, T \,|\, \Psi \rangle$ of the wavepacket componentwise.

Parameters

`summed`: Whether to sum up the individual integrals $\langle \Phi_i \,|\, T_{i,j} \,|\, \Phi_j \rangle$.

Return Value

The kinetic energy of the wavepacket's components Φ_i or the overall kinetic energy of Ψ.

grady(*self*, *component*)

Calculate the effect of $-i\varepsilon^2 \frac{\partial}{\partial x}$ on a component Φ_i of the Hagedorn wavepacket Ψ.

Parameters

`component`: The index i of the component Φ_i on which we apply the above operator.

Return Value

The modified coefficients.

project_to_canonical(*self*, *potential*)

Project the Hagedorn wavepacket into the canonical basis.

Parameters

`potential`: The potential V whose eigenvectors ν_l are used for the transformation.

Note: This function is expensive and destructive! It modifies the coefficients of the *self* instance.

project_to_eigen(*self*, *potential*)

Project the Hagedorn wavepacket into the eigenbasis of a given potential V.

Parameters

`potential`: The potential V whose eigenvectors ν_l are used for the transformation.

Note: This function is expensive and destructive! It modifies the coefficients of the *self* instance.

A.7.2 Instance Variables

Name	Description
number_components	Number of components Φ_i the wavepacket $\lvert\Psi\rangle$ has got.
basis_size	Size of the basis from which we construct the wavepacket.
parameters	Data structure that contains the Hagedorn parameters Π_i of each component Φ_i.

continued on next page

Name	**Description**
coefficients	The coefficients c^i of the linear combination for each component Φ_i.
quadrator	An object that provides nodes γ and weights ω for Gauss-Hermite quadrature.

A.8 Class WaveFunction

This class represents a vector valued quantum state $|\Psi\rangle$ as used in the vector valued time-dependent Schroedinger equation. The state $|\Psi\rangle$ is composed of $\psi_0, \ldots, \psi_{N-1}$ where ψ_i is a single wavefunction component.

A.8.1 Methods

__init__(*self*, *nodes*, *values*)

Initialize the *WaveFunction* object that represents the vector of states $|\Psi\rangle$.

Parameters

`nodes`: The grid nodes to which the numerical values of ψ_i belong to.

`values`: A list with the numerical values of each component ψ_i sampled at the given nodes.

__str__(*self*)

Return Value

A string that describes the wavefunction $|\Psi\rangle$.

get_number_components(*self*)

Return Value

The number of components ψ_i the vector $|\Psi\rangle$ consists of.

get_nodes(*self*)

Return Value

The grid nodes γ the wave function values belong to.

get_values(*self*)

Return the wave function values for each component of $|\Psi\rangle$.

Return Value

A list with the values of all components ψ_i evaluated on the grid nodes γ.

set_values(*self*, *values*)

Assign new function values for each component of $|\Psi\rangle$.

Parameters

`values`: A list with the new values of all the ψ_i.

Raises

`ValueError` If the list *values* has the wrong number of entries.

get_norm(*self*, *values*=`None`, *summed*=`False`, *component*=`None`)

Calculate the L^2 norm of the whole vector $|\Psi\rangle$ or some individual components ψ_i. The calculation is done in momentum space.

Parameters

`values:`	Allows to use this function for external data, similar to a static function.
`summed:`	Whether to sum up the norms of the individual components.
`component:`	The component ψ_i of which the norm is calculated.

Return Value

The L^2 norm of $|\Psi\rangle$ or a list of the L^2 norms of all components ψ_i. (Depending on the optional arguments.)

kinetic_energy(*self*, *kinetic*, *summed*=`False`)

Calculate the kinetic energy $E_{\text{kin}} := \langle \Psi \,|\, T \,|\, \Psi \rangle$ of the different components.

Parameters

`kinetic:`	The kinetic energy operator T.
`summed:`	Whether to sum up the kinetic energies of the individual components.

Return Value

A list with the kinetic energies of the individual components or the overall kinetic energy of the wavefunction. (Depending on the optional arguments.)

potential_energy(*self*, *potential*, *summed*=`False`)

Calculate the potential energy $E_{\text{pot}} := \langle \Psi \,|\, V \,|\, \Psi \rangle$ of the different components.

Parameters

`potential:`	The potential energy operator V.
`summed:`	Whether to sum up the potential energies of the individual components.

Return Value

A list with the potential energies of the individual components or the overall potential energy of the wavefunction. (Depending on the optional arguments.)

A.9 Class Propagator

Subclasses: FourierPropagator, HagedornPropagator, HagedornMultiPropagator

Propagators can numerically simulate the time evolution of quantum states as described by the time-dependent Schrödinger equation.

A.9.1 Methods

__init__(*self*)

Initialize a new *Propagator* instance.

Raises

`NotImplementedError` This is an abstract base class.

__str__(*self*)

Prepare a printable string representing the *Propagator* instance.

Raises

`NotImplementedError` This is an abstract base class.

get_number_components(*self*)

Return Value

The number of components of $|\Psi\rangle$.

Raises

`NotImplementedError` This is an abstract base class.

get_potential(*self*)

Return Value

The embedded *MatrixPotential* instance.

Raises

`NotImplementedError` This is an abstract base class.

get_wavefunction(*self*)

Return Value

Create a *WaveFunction* instance representing the wave function evaluated on a given grid.

Raises

`NotImplementedError` This is an abstract base class.

Note: This function can have an additional parameter for providing the grid.

propagate(*self*)

Given the wave function Ψ at time t, calculate the new Ψ at time $t + \tau$. We do exactly one timestep τ here.

Raises

`NotImplementedError` This is an abstract base class.

A.10 Class FourierPropagator

Propagator ─┐
FourierPropagator

This class can numerically propagate given initial values $|\Psi\rangle$ in a potential surface $V(x)$. The propagation is done with a Strang splitting of the time propagation operator.

A.10.1 Methods

__init__(*self*, *potential*, *initial_values*)

Initialize a new *FourierPropagator* instance. Precalculate also the grid and the propagation operators.

Parameters

`potential:`	The potential the state $|\Psi\rangle$ feels during the time propagation.
`initial_values:`	The initial values $|\Psi(t=0)\rangle$ given in the canonical basis.

Raises

`ValueError` If the number of components of $|\Psi\rangle$ does not match the number of energy levels λ_i of the potential.

Overrides: Propagator.__init__

__str__(*self*)

Prepare a printable string representing the *Propagator* instance.

Overrides: Propagator.__str__

get_number_components(*self*)

Return Value

The number of components of $|\Psi\rangle$.

Overrides: Propagator.get_number_components

get_potential(*self*)

Return Value

The *MatrixPotential* instance used for time propagation.

Overrides: Propagator.get_potential

get_wavefunction(*self*)

Return Value

The *WaveFunction* instance that stores the current wave function data.

Overrides: Propagator.get_wavefunction

get_operators(*self*)

Return Value

Return the numerical expressions of the propagation operators T and V.

propagate(*self*)

Given the wave function values Ψ at time t, calculate new values at time $t + \tau$. We perform exactly one timestep τ here.

Overrides: Propagator.propagate

kinetic_energy(*self*, *summed*=`False`)

This method just delegates the calculation of kinetic energies to the embedded *WaveFunction* object.

Parameters

`summed`: Whether to sum up the kinetic energies of the individual components.

Return Value

The kinetic energies.

Overrides: Propagator.kinetic_energy

potential_energy(*self*, *summed*=`False`)

This method just delegates the calculation of potential energies to the embedded *WaveFunction* object.

Parameters

`summed`: Whether to sum up the potential energies of the individual components.

Return Value

The potential energies.

Overrides: Propagator.potential_energy

A.10.2 Instance Variables

Name	Description
potential	The embedded *MatrixPotential* instance representing the potential V.

continued on next page

Name	**Description**
Psi	The initial values of the components ψ_i sampled at the given nodes.
nodes	The position space nodes γ.
V	The potential operator V defined in position space.
omega	The momentum space nodes ω.
T	The kinetic operator T defined in momentum space.
TE	Exponential $\exp(T)$ of T used in the Strang splitting.
VE	Exponential $\exp(V)$ of V used in the Strang splitting.

A.11 Class HagedornPropagator

Propagator ─┐
HagedornPropagator

This class can numerically propagate given initial values $|\Psi\rangle$ in a potential $V(x)$. The propagation is done for a given homogeneous Hagedorn wavepacket.

A.11.1 Methods

__init__(*self*, *potential*, *packet*, *leading_component*)

Initialize a new *HagedornPropagator* instance.

Parameters

`potential:`	The potential the wavepacket $|\Psi\rangle$ feels during the time propagation.
`packet:`	The initial homogeneous Hagedorn wavepacket we propagate in time.
`leading_component:`	The leading component index χ.

Raises

`ValueError` If the number of components of $|\Psi\rangle$ does not match the number of energy levels λ_i of the potential.

Overrides: Propagator.__init__

__str__(*self*)

Prepare a printable string representing the *HagedornPropagator* instance.

Overrides: Propagator.__str__

get_number_components(*self*)

Return Value

The number N of components Φ_i of $|\Psi\rangle$.

Overrides: Propagator.get_number_components

get_potential(*self*)

Return Value

The *MatrixPotential* instance used for time propagation.

Overrides: Propagator.get_potential

get_wavepacket(*self*)

Return Value

The *HagedornWavepacket* instance that represents the current wavepacket $|\Psi\rangle$.

get_wavefunction(*self*, *nodes*)

Construct a *WaveFunction* object which contains the components Φ_i of the Hagedorn wavepacket evaluated at the given nodes γ.

Parameters

`nodes`: The nodes γ on which the Hagedorn wavepacket is evaluated.

Return Value

A *WaveFunction* instance representing the values of the current $|\Psi\rangle$.

Note: This method is quite expensive.

Overrides: Propagator.get_wavefunction

propagate(*self*)

Given the wavepacket Ψ at time t, calculate a new wavepacket at time $t+\tau$. We perform exactly one timestep τ here.

Overrides: Propagator.propagate

A.11.2 Instance Variables

Name	Description
potential	The potential $V(x)$ the packet feels.
number_components	Number N of components the wavepacket $\|\Psi\rangle$ has got.
leading	The leading component χ is the index of the eigenvalue of the potential that is responsible for propagating the Hagedorn parameters.
packet	The Hagedorn wavepacket.

A.12 Class HagedornMultiPropagator

Propagator ─┐
HagedornMultiPropagator

This class can numerically propagate given initial values $|\Psi\rangle$ in a potential $V(x)$. The propagation is done for a given inhomogeneous Hagedorn wavepacket.

A.12.1 Methods

__init__(*self*, *potential*, *packet*)

Initialize a new *HagedornMultiPropagator* instance.

Parameters

`potential:` The potential the wavepacket $|\Psi\rangle$ feels during the time propagation.

`packet:` The initial inhomogeneous Hagedorn wavepacket we propagate in time.

Raises

`ValueError` If the number of components of $|\Psi\rangle$ does not match the number of energy levels λ_i of the potential.

Overrides: Propagator.__init__

__str__(*self*)

Prepare a printable string representing the *HagedornMultiPropagator* instance.

Overrides: Propagator.__str__

get_number_components(*self*)

Return Value

The number N of components Φ_i of $|\Psi\rangle$.

Overrides: Propagator.get_number_components

get_potential(*self*)

Return Value

The *MatrixPotential* instance used for time propagation.

Overrides: Propagator.get_potential

get_wavepacket(*self*)

Return Value

The *HagedornMultiWavepacket* instance that represents the current wavepacket $|\Psi\rangle$.

get_wavefunction(*self*, *nodes*)

Construct a *WaveFunction* object which contains the components Φ_i of the Hagedorn wavepacket evaluated at the given nodes γ.

Parameters

`nodes`: The nodes γ on which the Hagedorn wavepacket is evaluated.

Return Value

A *WaveFunction* instance representing the values of the current $|\Psi\rangle$.

Note: This method is quite expensive.

Overrides: Propagator.get_wavefunction

propagate(*self*)

Given the wavepacket Ψ at time t, calculate a new wavepacket at time $t + \tau$. We perform exactly one timestep τ here.

Overrides: Propagator.propagate

A.12.2 Instance Variables

Name	Description
potential	The potential $V(x)$ the packet feels.
number_components	Number N of components the wavepacket $|\Psi\rangle$ has got.
packet	The Hagedorn wavepacket.

A.13 Class Quadrator

This class is an abstract interface to a Gauss-Hermite quadrature rule tailored at the needs of Hagedorn wavepackets.

A.13.1 Methods

__init__(*self*, *order*)

Initialize a new quadrature rule.

Parameters

`order`: The order R of the Gauss-Hermite quadrature.

Raises

`ValueError` If the order is less then 2.

__str__(*self*)

get_order(*self*)

Return Value

The order R of the quadrature.

get_number_nodes(*self*)

Return Value

The number of quadrature nodes.

get_nodes(*self*)

Return Value

An array containing the quadrature nodes γ_i.

get_weights(*self*)

Return Value

An array containing the quadrature weights ω_i.

hermite_recursion(*self*, *nodes*)

Evaluate the Hermite functions recursively up to the order R on the given nodes.

Parameters

`nodes`: The points at which the Hermite functions are evaluated.

Return Value

Returns a twodimensional array H where the entry $H[k, i]$ is the value of the k-th Hermite function evaluated at the node i.

A.13.2 Instance Variables

Name	**Description**
order	The order R of the Gauss-Hermite quadrature.
nodes	The quadrature nodes γ_i.
weights	The quadrature weights ω_i.

A.14 Module Parameters

This is the configuration file for the multistate simulation code. All available configuration parameters can be set here. This file is imported from the simulation code. For configuring the code, just modifiy the values here.

A.14.1 Variables

Name	Description
algorithm	The algorithm used for time propagation, can be one of: `fourier` | `hagedorn` | `multihagedorn`.
potential	The potential $V(x)$ used in the simulation. See the *PotentialLibrary* for available potentials.
T	Perform a simulation in the time interval $[0, T]$.
dt	Duration of a single time step τ.
eps	The parameter ε in the semiclassical scaling.
delta	A variable that is used in the definition of some potentials.
coefficients	A list of N lists of (k, c_k) tuples that set the coefficient c_k of the basis function ϕ_k. The i-th list contains the coefficients c^i of the component Φ_i of the initial wavepacket Ψ.
parameters	A list of the Hagedorn parameter sets Π_i of component Φ_i of the initial wavepacket Ψ.
ngn	The Number of grid nodes γ_i in position space. **Value:** `4096`
f	Scaling factor f for the computational domain Ω in position space. The interval in the position space is given by $[-f\pi, f\pi]$. **Value:** `5.0`
basis_size	The number K of basis functions ϕ_k used for Hagedorn wavepackets Φ. **Value:** `64`
leading_component	The leading component index χ of the eigenvalue λ_χ that governs the propagation of the Hagedorn parameters Π for homogeneous wavepackets. **Value:** `0`
outfile_nodes	Filename of the output file that contains the grid nodes γ_i. **Value:** `'nodes.dat'`

continued on next page

Name	Description
outfile_wavefunction	Filename of the output file that contains the wavefunction ψ_i. **Value:** `'wavefunction.dat'`
outfile_energies	Filename of the output file that contains the energies E. **Value:** `'energies.dat'`
outfile_operators	Filename of the output file that contains the operators T and V. **Value:** `'operators.dat'`
outfile_parameters	Filename of the output file that contains the Hagedorn parameters Π_i. **Value:** `'parameters.dat'`
outfile_coefficients	Filename of the output file that contains the Hagedorn coefficients c^i. **Value:** `'coefficients.dat'`
write_nth	Write data to disk only each n-th timestep. **Value:** `1`

A.15 Module PotentialLibrary

This file contains some ready made potentials. They are stored as tupels of variables and expressions. The expression need to be a sympy 'Matrix' object, even if it's only a 1×1 matrix.

A.15.1 Variables

Name	Description
x	Position space variables. Currently only one space dimension is supported. **Value:** x
quadratic	Simple harmonic potential. **Potential:** $$\left(\frac{1}{2}\sigma x^2\right) \tag{A.1}$$ where $\sigma = 0.05$.
pert_quadratic	Perturbed harmonic potential. **Potential:** $$\left(\frac{1}{2}\sigma x^2 + \frac{1}{2}\varepsilon^2 x^2\right) \tag{A.2}$$ where $\sigma = 0.05$.
quartic	A simple fourth order anharmonic potential. **Potential:** $$\left(\frac{1}{4}\sigma x^4\right) \tag{A.3}$$ where $\sigma = 0.05$.
cos_waves	A potential consisting of a cosine wave. **Potential:** $$\left(\alpha(1 - \cos(\gamma x))\right) \tag{A.4}$$ where $\alpha = 1.0$ and $\gamma = 1.0$.
double_well	A double well potential. **Potential:** $$\left(\sigma(x^2 - 1)^2\right) \tag{A.5}$$ where $\sigma = 1.0$.
eckart	The Eckart potential. **Potential:** $$\left(\sigma \frac{1}{\cosh\left(\frac{x}{a}\right)^2}\right) \tag{A.6}$$ where $\sigma = 100 \cdot 3.8088e^{-4}$ and $a = 1.0/(2.0 \cdot 0.52917721018)$.
wall	A smooth unitstep like wall. **Potential:** $$\left(\arctan(\sigma x) + \frac{\pi}{2}\right) \tag{A.7}$$ where $\sigma = 10.0$.

continued on next page

Name	Description
v_shape	A narrow 'V'-like potential. **Potential:** $$\left(\frac{1}{2}\sqrt{\frac{9}{16}\varepsilon^2+\tanh(x)^2}\right) . \qquad (A.8)$$
two_quadratic	Double harmonic potential for two components. **Potential:** $$\begin{pmatrix} \frac{1}{2}\sigma x^2 & 0 \\ 0 & \frac{1}{2}\sigma x^2 \end{pmatrix} \qquad (A.9)$$ where $\sigma = 0.05$.
two_quartic	Double quartic anharmonic potential for two components. **Potential:** $$\begin{pmatrix} \frac{1}{4}\sigma x^4 & 0 \\ 0 & \frac{1}{8}\sigma x^4 \end{pmatrix} \qquad (A.10)$$ where $\sigma = 1.0$.
matrix1_diag	Diagonalized single avoided crossing. **Potential:** $$\begin{pmatrix} \sqrt{\frac{9}{16}\varepsilon^2+\tanh(x)^2} & 0 \\ 0 & -\sqrt{\frac{9}{16}\varepsilon^2+\tanh(x)^2} \end{pmatrix} . \qquad (A.11)$$
delta_gap	A potential with a single avoided crossing. **Potential:** $$\frac{1}{2}\begin{pmatrix} \tanh(x) & \delta \\ \delta & -\tanh(x) \end{pmatrix} . \qquad (A.12)$$
two_crossings	A potential with two avoided crossings in series. **Potential:** $$\frac{1}{2}\begin{pmatrix} \Theta & \delta \\ \delta & -\Theta \end{pmatrix} \qquad (A.13)$$ where $\Theta := \tanh(x-\rho)\tanh(x+\rho)$ and $\rho = 3.0$.
three_quadratic	Decoupled harmonic potentials for three components. **Potential:** $$\begin{pmatrix} \frac{1}{2}\sigma x^2 & 0 & 0 \\ 0 & \frac{1}{2}\sigma x^2 & 0 \\ 0 & 0 & \frac{1}{2}\sigma x^2 \end{pmatrix} \qquad (A.14)$$ where $\sigma = 0.05$.

continued on next page

Name	**Description**
three_states	A potential with three energy levels and multiple crossings. **Potential:** $$\begin{pmatrix} \Theta + \Xi & \delta_1 & \delta_2 \\ \delta_1 & -\Theta & 0 \\ \delta_2 & 0 & 1 - \Xi \end{pmatrix} \tag{A.15}$$ where $$\Theta := \tanh(x + \rho) \quad \text{and} \quad \Xi := \tanh(x - \rho)$$ and $\rho = 3.0$.
four_powers	Harmonic and higher order anharmonic potentials for four components. **Potential:** $$\begin{pmatrix} \frac{1}{2}\sigma x^2 & 0 & 0 & 0 \\ 0 & \frac{1}{4}\sigma x^4 & 0 & 0 \\ 0 & 0 & \frac{1}{6}\sigma x^6 & 0 \\ 0 & 0 & 0 & \frac{1}{8}\sigma x^8 \end{pmatrix} \tag{A.16}$$ where $\sigma = 0.05$.
five_quadratic	Decoupled harmonic potentials for five components. **Potential:** $$\begin{pmatrix} \frac{1}{2}\sigma x^2 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2}\sigma x^2 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2}\sigma x^2 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}\sigma x^2 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}\sigma x^2 \end{pmatrix} \tag{A.17}$$ where $\sigma = 0.05$.

A.16 Class SimulationLoop

This class acts as the main simulation loop. It owns a propagator that propagates a set of initial values during a time evolution. All values are read from the *Parameters.py* file.

A.16.1 Methods

__init__(*self*)

Create a new simulation loop instance.

add_fourier_propagator(*self*)

Set up a Fourier propagator for the simulation loop. Set the potential and initial values according to the configuration.

Raises

`ValueError` For invalid or missing input data.

add_hagedorn_propagator(*self*)

Set up a Hagedorn propagator for the simulation loop. Set the potential and initial values according to the configuration.

Raises

`ValueError` For invalid or missing input data.

add_multi_hagedorn_propagator(*self*)

Set up a multi Hagedorn propagator for the simulation loop. Set the potential and initial values according to the configuration.

Raises

`ValueError` For invalid or missing input data.

run_fourier_propagator(*self*)

Run the simulation loop for a number of time steps. The number of steps is calculated in the *initialize* function.

run_hagedorn_propagator(*self*)

Run the simulation loop for a number of time steps. The number of steps is calculated in the *initialize* function.

run_multi_hagedorn_propagator(*self*)

Run the simulation loop for a number of time steps. The number of steps is calculated in the *initialize* function.

end_simulation(*self*)

Do the necessary cleanup after a simulation. For example request the serializer to write the data and close the output files.

A.16.2 Instance Variables

Name	Description
propagator	The time propagator instance driving the simulation.
serializer	A *Serializer* instance for saving simulation results.
nsteps	The number of time steps we will perform.

A.17 Class Serializer

A serializer class that can save various simulation results into data files. The output files can be processed further for producing e.g. plots.

A.17.1 Methods

__init__(*self*)

Set up a new *Serializer* instance. The output files are created and opened.

set_interval(*self*, *value*)

Set the interval (time steps) at which the data gets written to the disk.

Parameters

`value`: Skip that number of time steps before writing data again.

finalize(*self*)

Close the open output files.

save_nodes(*self*, *nodes*)

Save the grid nodes to a file.

save_wavefunction(*self*, *wavefunction*)

Save a *WaveFunction* instance. The output is suitable for the plotting routines.

Parameters

`wavefunction`: The *WaveFunction* instance to save.

save_energies(*self*, *energies*)

Save the kinetic and potential energies to a file.

Parameters

`energies`: A tuple `(ekin, epot)` containing the energies.

save_operators(*self*, *operators*)

Save the kinetic and potential operator to a file.

Parameters

`operators`: The operators to save.

save_coefficients(*self*, *coefficients*)

Save the coefficients of the Hagedorn wavepacket to a file.

Parameters

`coefficients`: The coefficients of the Hagedorn wavepacket.

save_parameters(*self*, *parameters*)

Save the parameters of the Hagedorn wavepacket to a file.

Parameters

`parameters`: The parameters of the Hagedorn wavepacket.

Bibliography

[1] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, 10th printing with corrections edition, 1972. (Cited on page 41.)

[2] Dennis Bernstein and Wasin So. Some explicit formulas for the matrix exponential. *IEEE Transactions on Automatic Control*, 38(8):1228–1232, Aug 1993. (Cited on page 12.)

[3] Erwan Faou, Vasile Gradinaru, and Christian Lubich. Computing semiclassical quantum dynamics with Hagedorn wavepackets. *SIAM Journal on Scientific Computing*, 31(4):3027–3041, 2009. (Cited on pages 35, 36, 37 and 43.)

[4] Gene H. Golub and John H. Welsch. Calculation of Gauss quadrature rules. *Mathematics of Computation*, 23(106):221–230, 04 1967. (Cited on page 41.)

[5] V. Gradinaru. Strang splitting for the time dependent Schrödinger equation on sparse grids. *SIAM Journal on Numerical Analysis*, 46(1):103–123, 2007/08. (Cited on page 16.)

[6] George A. Hagedorn. Semiclassical quantum mechanics. I: The $\hbar \to 0$ limit for coherent states. *Communications in Mathematical Physics*, 71(1):77–93, 1980. (Cited on page 27.)

[7] George A. Hagedorn. Semiclassical quantum mechanics III: The large order asymptotics and more general states. *Annals of Physics*, 135(1):58–70, 1981. (Cited on page 27.)

[8] George A. Hagedorn. Classification and normal forms for avoided crossings of quantum-mechanical energy levels. *Journal of Physics A: Mathematical and General*, 31:369, 1998. (Cited on page 9.)

[9] George A. Hagedorn. Raising and lowering operators for semiclassical wave packets. *Annals of Physics*, 269(1):77–104, 1998. (Cited on page 27.)

[10] George A. Hagedorn and Sam L. Robinson. Bohr-Sommerfeld quantization rules in the semiclassical limit. *Journal of Physics. A. Mathematical and General*, 31(50):10113–10130, 1998. (Cited on page 57.)

[11] George A. Hagedorn and Julio H. Toloza. Exponentially accurate semi-

classical asymptotics of low-lying eigenvalues for 2×2 matrix Schrödinger operators. *Journal of Mathematical Analysis and Applications*, 312(1):300–329, 2005. (Cited on page 11.)

[12] Tobias Jahnke and Christian Lubich. Error bounds for exponential operator splittings. *BIT Numerical Mathematics*, 40:735–744, 2000. (Cited on page 16.)

[13] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. (Cited on page 13.)

[14] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. 20(4):801–836, 1978. (Cited on page 13.)

[15] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. 45(1):3–49, 2003. (Cited on page 13.)

[16] S. Teufel. *Adiabatic perturbation theory in quantum dynamics.* Springer Verlag, 2003. (Cited on page 8.)