Efficient implementation of Hagedorn wavepackets in C++

Bachelor Thesis

written by
Michaja Bösch

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

Seminar for Applied Mathematics
ETH Zurich

Spring semester 2015

Contents

1 Introduction

The topic of this thesis is the C++ reimplementation of the library WaveBlock-sND [2], which is written in Python. This report describes the inner working of my C++ implementation and explains the reasoning behind important software designs choices. This report doesn't explain the theory behind Hagedorn wavepackets. For an introduction to Hagedorn wavepackets, I refer to some earlier work of George A. Hagedorn [4]. The technical details of WaveBlock-sND are explained in the master thesis of Raoul Bourquin [1].
WaveBlocksND implements the time propagation of Hagedorn wavepackets. The extent is too large for a bachelor thesis, therefore I focused on the evaluation and gradient computation of Hagedorn wavepackets. I made extensive use of Eigen [3], a C++ template library for linear algebra.

2 Basis shape

A D-dimensional basis shape $\mathfrak{K}$ is a set of *unordered* D-dimensional integer-tuples, also referred to as *nodes*. A shape is suitable for our needs, if it satisfies the fundamental property

$$\underline{k} \in \mathfrak{K} \Rightarrow \forall \underline{k} - \underline{e}^d \in \mathfrak{K} \; \forall d \in \{d \mid k_d \geq 1\} \tag{1}$$

where $\underline{e}^d$ is the unit vector in direction d. That means, if an arbitrary node is part of the basis shape, then all nodes in the backward cone are part of the shape too.

2.1 Basis shape description

In WaveBlocksND, implementing a shape is quite involved since you not only have to describe which nodes lie in a shape (which is easy), as well you have to provide node iterators in two flavours (which is hard). In my implementation, the iterator part is a service provided by the *shape enumerator*. Thus, if you want to use a new shape type in your simulation, all you have to do, is to provide an algebraic description, that describes which nodes are part of the shape. The shape enumerator takes care of the rest.
This algebraic description consists of a set of surface functions and a minimum bounding box.

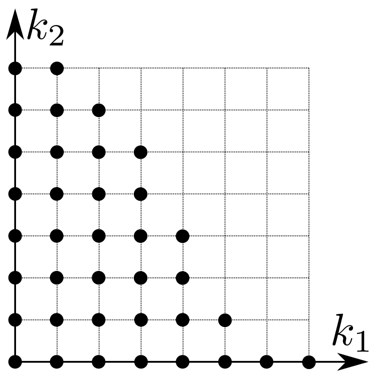
Definition 1 (Surface functions of basis shapes)**.** *The D surface functions of an arbitrary D-dimensional shape are defined by*

$$s_\alpha(\underline{n}) = \max\{k_\alpha \mid \underline{k} \in \mathfrak{K} \wedge k_d = n_d \; \forall d \neq \alpha\} \tag{2}$$

These functions are well-defined, because of the fundamental shape property (1). This definition looks intricate, but once understood, defining shapes with it is straightforward.

Definition 2 (Minimum bounding box of basis shapes). *The minimum bounding box describes the smallest box, within which all shape nodes lie.*

$$L_\alpha = \max\{k_\alpha \mid \underline{k} \in \mathfrak{K}\} \tag{3}$$



$\underline{n}$	$s_1(\underline{n})$	$\underline{n}$	$s_2(\underline{n})$
(?, 0)	7	(0, ?)	7
(?, 1)	5	(1, ?)	7
(?, 2)	4	(2, ?)	6
(?, 3)	4	(3, ?)	5
(?, 4)	3	(4, ?)	3
(?, 5)	3	(5, ?)	1
(?, 6)	2	(6, ?)	0
(?, 7)	1	(7, ?)	0
(?, 8)	$-\infty$	(8, ?)	$-\infty$

Figure 1: Surface description of an 2-dimensional basis shape.

2.2 Common basis shapes

Definition 3 (Hypercubic basis shape). *Given the limits $\underline{K} \in \mathbb{N}^D$, a hypercubic shape is defined by*

$$\mathfrak{K}(D, \underline{K}) := \left\{(k_1, \dots, k_D) \in \mathbb{N}_0^D \mid k_d < K_d \forall d\right\} \tag{4}$$

We derive the surface functions by inserting (4) into equation (2):

$$s_\alpha(\underline{n}) = \max\{k_\alpha \mid (k_\alpha < K_\alpha) \land (n_d < K_d \ \forall d \neq \alpha)\}$$

$$s_\alpha(\underline{n}) = \begin{cases} K_\alpha - 1 & n_d < K_d \ \forall d \neq \alpha \\ -\infty & otherwise \end{cases} \tag{5}$$

We retrieve the minimum bounding box by inserting (4) into equation (3):

$$L_\alpha = \max\{k_\alpha \mid k_\alpha < K_\alpha\}$$

$$L_\alpha = K_\alpha - 1 \tag{6}$$

Definition 4 (Hyperbolic cut basis shape). *Given the sparsity parameter $S \geq 1$, a hyperbolic cut shape is defined by*

$$\mathfrak{K}(D, S) := \left\{(k_1, \dots, k_D) \in \mathbb{N}_0^D \mid \prod_{d=1}^{D} (k_d + 1) \leq S\right\} \tag{7}$$

We derive the surface functions by inserting (7) into equation (2):

$$s_\alpha(\underline{n}) = \max\left\{k_\alpha \mid (k_\alpha + 1)\prod_{d\neq\alpha}(n_d + 1) \leq S\right\} = \max\left\{k_\alpha \mid k_\alpha + 1 \leq \frac{S}{\prod_{d\neq\alpha}(n_d+1)}\right\}$$

$$s_\alpha(\underline{n}) = \frac{1}{\prod_{d\neq\alpha}(n_d+1)} - 1 \tag{8}$$

We retrieve the minimum bounding box by inserting (7) into equation (3):

$$L_\alpha = \max\left\{k_\alpha \mid \prod_{d=1}^{D}(k_d + 1) \leq S\right\} = \max\left\{k_\alpha \mid k_\alpha + 1 \leq \frac{S}{\prod_{d\neq\alpha}(k_d+1)}\right\}$$

$$L_\alpha = S - 1 \tag{9}$$

Definition 5 (Shape intersection)**.** *The intersection of the shapes $\mathfrak{K}^A$ and $\mathfrak{K}^B$ is defined by*

$$\mathfrak{K}^{A\cap B} = \left\{\underline{k} \mid \underline{k} \in \mathfrak{K}^A \wedge \underline{k} \in \mathfrak{K}^B\right\} \tag{10}$$

Using equation (2) and (3) we obtain the surface functions

$$s_\alpha^{A\cap B}(\underline{n}) = \min\{s_\alpha^A(\underline{n}), s_\alpha^B(\underline{n})\} \tag{11}$$

and the minimum bounding box

$$L_\alpha^{A\cap B} = \min\{L_\alpha^A, L_\alpha^B\} \tag{12}$$

Remark: The *limited hyperbolic cut shape* is an intersection of an *hypercubic shape* and a *hyperbolic cut shape*.

Definition 6 (Shape union)**.** *The union of the shapes $\mathfrak{K}^A$ and $\mathfrak{K}^B$ is defined by*

$$\mathfrak{K}^{A\cup B} = \left\{\underline{k} \mid \underline{k} \in \mathfrak{K}^A \vee \underline{k} \in \mathfrak{K}^B\right\} \tag{13}$$

Using equation (2) and (3) we obtain the surface functions

$$s_\alpha^{A\cup B}(\underline{n}) = \max\{s_\alpha^A(\underline{n}), s_\alpha^B(\underline{n})\} \tag{14}$$

and the minimum bounding box

$$L_\alpha^{A\cup B} = \max\{L_\alpha^A, L_\alpha^B\} \tag{15}$$

2.3 Basis shape slicing

Many algorithms, notably the evaluation of a Hagedorn wavepacket, use recursive formulas of the form $\phi_{\underline{k}} = f(\phi_{\underline{k}-\underline{e}^1}, \ldots, \phi_{\underline{k}-\underline{e}^D})$ where $\phi_{\underline{k}}$ is a value associated with the node $\underline{k}$ and $\underline{e}^d$ is the unit vector in direction d. Thus, it is beneficial to organize a shape into slices.

Definition 7 (Basis shape slice). *Basis shapes $\mathfrak{K}$ are divided into disjoint slices $\mathfrak{S}$*

$$\mathfrak{K} = \bigcup_{s=0}^{\infty} \mathfrak{S}^s \tag{16}$$

The s-th slice $\mathfrak{S}^s$ of a shape $\mathfrak{K}$ contains all nodes $\underline{k} \in \mathfrak{K}$ that satisfy

$$\sum_{d=1}^{D} k_d = s \tag{17}$$

Apparently, s is the Manhattan distance or l_1 norm to the origin.

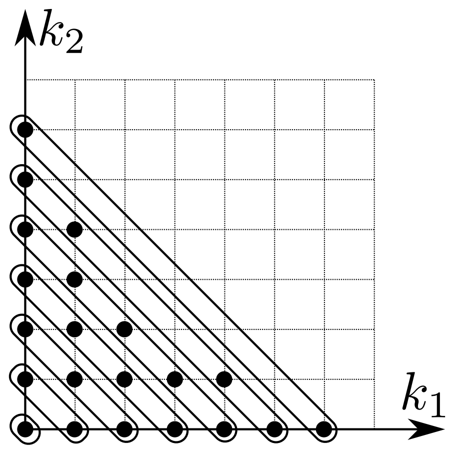


Figure 2: A basis shape and its slicing.

2.4 Basis shape enumeration

A basis shape description just tells, whether it contains a specific node. But we need to associate coefficients $c_{\underline{k}}$ and basis functions $\phi_{\underline{k}}$ with shape nodes $\underline{k}$. We can use a hash table to map $\underline{k}$ to $c_{\underline{k}}$, $\phi_{\underline{k}}$. But it is simpler to enumerate all nodes in a shape. This means, if a multi-index $\underline{k}$ maps to an ordinal i, we find $\phi_{\underline{k}}$ at the position i in the array $\{\phi\}$. This way, we can keep coefficients and basis function values in an array, ordered according to the shape enumeration.

Definition 8. *A shape enumeration is a bijective mapping that orders all nodes of $\mathfrak{K}$ and assigns the i-th node the ordinal i.*

Remark: Due to the close relationship of the enumeration to the shape, the symbol $\mathfrak{K}$ is used for both shape and shape enumeration.

2.4.1 Shape enumeration algorithm

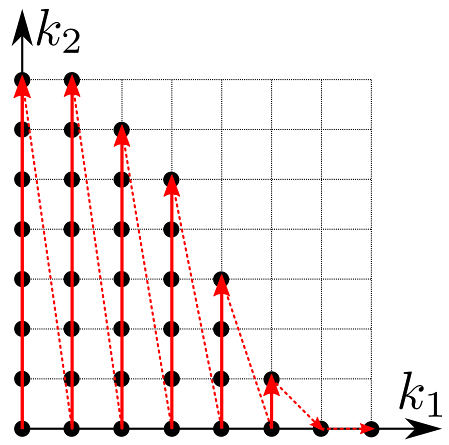


Figure 3: Enumerator passage through a 2-dimensional basis shape

The enumerator takes a description of a basis shape and enumerates it in two passes.
In the first pass, knowing the surface functions s_α and the minimum bounding volume L_α of a shape, the enumerator iterates through all nodes inside the basis shape in lexicographical order. Each node is subsequently appended to the proper slice (see section (2.3)). In the second pass, the enumerator determines the offset of each slice. The offset is the number of all nodes in previous slices. Equivalently, it is the ordinal of the first node of the slice (if assuming zero-based arrays).

Algorithm 1: Enumerator passage through a 3-dimensional basis shape.

```
*Enumerate all nodes in lexicographical order*
for $k_1 \leftarrow 0$ to $s_1((0,0,0))$ do
    for $k_2 \leftarrow 0$ to $s_2((k_1,0,0))$ do
        for $k_3 \leftarrow 0$ to $s_3((k_1,k_2,0))$ do
            *Append lattice node to its slice*
            $\mathfrak{S}_{k_1+k_2+k_3} \leftarrow \mathfrak{S}_{k_1+k_2+k_3} \cup (k_1,k_2,k_3)$
        end
    end
end
```

Remark: This algorithm is recursive (one loop for each dimension). The real implementation however is iterative.

2.4.2 Data structure

The enumerator data structure is as follows

```
template<dim_t D, class MultiIndex>
struct ShapeEnum {
    std::vector< ShapeSlice<D,MultiIndex> > slices;
};

template<dim_t D, class MultiIndex>
struct ShapeSlice {
    std::size_t offset;
    std::vector< MultiIndex > table;
};
```

The class *ShapeEnum* representing $\mathfrak{K}$ contains a list of all non-empty shape slices. The class *ShapeSlice* $\mathfrak{S}^s$ contains an array of all nodes $\underline{k} \in \mathfrak{S}^s$.
All multi-indices of a slice are lexicographically ordered. The lexicographical order begins on the first index, continues to the second index and so forth.

$$a_1 a_2 \dots a_D <_{lex} b_1 b_2 \dots b_D \iff \begin{cases} (a_1 < b_1) \vee (a_1 = b_1 \wedge a_2 \dots a_D <_{lex} b_2 \dots b_D) & D > 1 \\ (a_1 < b_1) & D = 1 \end{cases} \tag{18}$$

2.4.3 Queries

The two main operations of a shape enumeration are:

Retrieve the multi-index $\underline{k}$ associated with a given ordinal i

All multi-indices of a shape are stored in an array. Thus, we simply return the i-th multi-index.

Run-time complexity: $\mathcal{O}(1)$

Find the ordinal i of a given multi-index $\underline{k}$

First we have to determine, to which slice $\mathfrak{S}^s$ the multi-index $\underline{k}$ belongs by computing $s = \sum k_d$.

All nodes inside a slice are ordered lexicographically. Therefore we determine the position of the node $\underline{k}$ inside the slice using **binary search**. The final ordinal is the position of $\underline{k}$ inside its slice plus the number of nodes in all previous slices.

Run-time complexity: $\mathcal{O}(\log S)$, where S is the size of the slice.

2.4.4 Finding backward neighbours

A common task is looking up all backward neighbours $\{\underline{k}-\underline{e}^1,\dots,\underline{k}-\underline{e}^D\}$ of a node $\underline{k}$. This task can be slightly optimized: First, all neighbours live inside the same slice. Secondly, they are ordered i.e. $(\underline{k}-\underline{e}^d)$ is stored before $(\underline{k}-\underline{e}^{d+1})$. Thirdly, due to how lexicographical order works, the distance between $(\underline{k}-\underline{e}^{d-1})$ and $(\underline{k}-\underline{e}^d)$ is usually much larger than the distance between $(\underline{k}-\underline{e}^d)$ and $(\underline{k}-\underline{e}^{d+1})$. The following algorithm takes advantage of this alignment by defining a search range $[a;b]$, which is the whole slice at the beginning. After each determined ordinal, the algorithm subsequently reduces the search range of the remaining nodes.

Algorithm 2: Algorithm to find ordinals of backward neighbours

```
Input : Node $\underline{k}$.
Input : Slice that contains the backward neighbours $\mathfrak{S}$.
Output: Ordinals of backward neighbours $\{i^1,\dots,i^D\}$.
$a \leftarrow 1$
$b \leftarrow size(\mathfrak{S})$
$i^D \leftarrow find(\underline{k}-\underline{e}^D) \in \mathfrak{S}[a;b]$
$b \leftarrow i^D$
for $d \leftarrow 1$ to $D-1$ do
    $i^d \leftarrow find(\underline{k}-\underline{e}^d) \in \mathfrak{S}[a;b]$
    $a \leftarrow i^d$
end
```

2.4.5 Multi-index representation

The size of a multi-index has a big influence to lookup times and overall simulation time. Therefore the user can choose at compile-time an appropriate type to represent multi-indices. I provide the class *TinyMultiIndex* that packs the whole multi-index into 64 bits. Multi-indices of accomplishable simulations should fit into 64 bits. Otherwise you can define a new multi-index type. A custom implementation must possess the same semantics as `std::array<int,D>`. Furthermore it has to specialize `std::less` that performs lexicographical index comparison beginning on the first index. And it has to specialize `std::equal_to` and `std::hash` to enable use of multi-indices as hash table keys.

2.4.6 Alternative data structures

I tried a hash table (`std::unordered_set`) to map multi-indices to ordinals. To represent multi-indices, I used the tiny (64 bits) representation. These 64 bits can be used unmodified as a hash value. But queries proved to be 2-3 times slower compared to binary search, despite of $\mathcal{O}(1)$ compared to $\mathcal{O}(\log n)$ lookup time.

Hash Tables are good when querying random keys. But an algorithm working with basis shapes does *not* access random nodes. Such algorithms typically access neighbours of previously accessed nodes. The problem is, that hash tables map neighbour nodes to seemingly random locations in memory, while an sorted array keeps neighbours together. Thus, a hash table triggers a lot more cache misses, than binary search on an sorted array.

2.5 Basis shape extension

We need basis shape extensions to compute wavepacket gradients.

Definition 9 (Extended basis shape)**.** *Given a basis shape $\mathfrak{K}$, the shape extension $\mathfrak{K}_{ext}$ is defined by*

$$\mathfrak{K}_{ext} := \mathfrak{K} \cup \left\{ \underline{k}' \mid \underline{k}' = \underline{k} + \underline{e}^d \forall \underline{k} \in \mathfrak{K} \forall d \right\} \tag{19}$$

where $\underline{e}^d$ is the unit vector in direction d.

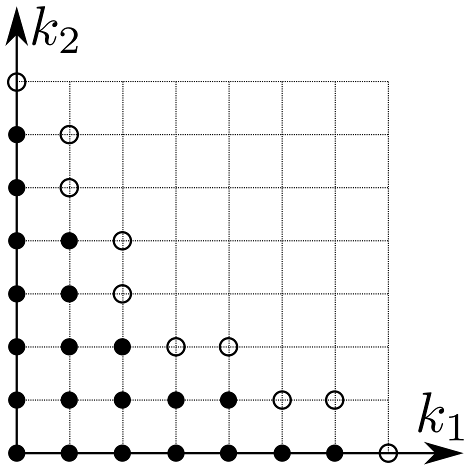


Figure 4: Basis shape (filled bullets) and its extension (empty bullets).

While is possible to define surface functions of the extension, creating an extended shape this way is impractical. The basis shape extension is needed, when we have to compute the gradient of a wavepacket. At this stage, the wavepacket's basis shape is in its enumerated form, basically just an array of multi-indices. The original shape description is not available. Therefore, we somehow have to create the extension out of the enumerated form.
This task turns out rather straight-forward. We take one slice $\mathfrak{S}^s$ of the input basis shape, clone it and shift it one unit to direction d by incrementing the d-th index of every multi-index. We create one clone for each direction. Then we merge these D clones and get the slice $\mathfrak{S}^{s+1}_{ext}$ of the extended shape. Notice

that nodes inside a clone are already lexicographically ordered. Thus, during the merge operation, we just have to interleave the clones in such a way that lexicographical order is kept and duplicates get eliminated. Using a divide and conquer approach, the total algorithmic complexity is $\mathcal{O}(D \log D \cdot N)$, where D is the wavepacket dimensionality and N is the number of basis shape nodes.

Algorithm 3: Create extended shape of an already enumerated basis shape.

```
Input  : Enumerated basis shape (slices) $\mathfrak{K} = \bigcup_s \mathfrak{S}^s$
Output: Basis shape extension (slices) $\mathfrak{K}_{ext} = \bigcup_s \mathfrak{S}^s_{ext}$
$\mathfrak{S}^0_{ext} \leftarrow \underline{0}$
foreach $\mathfrak{S}^s \in \mathfrak{K}$ do
    for $d \leftarrow 1$ to $D$ do
        $\mathfrak{S}^{s+1}_{ext} \leftarrow \mathfrak{S}^{s+1}_{ext} \cup \left\{ \underline{k} + \underline{e}^d \ \forall \underline{k} \in \mathfrak{S}^s \right\}$
    end
end
```

3 Hagedorn wavepackets

3.1 Scalar wavepackets

A D-dimensional scalar Hagedorn wavepacket is a linear combination of basis functions $\phi_{\underline{k}}$ with coefficients $c_{\underline{k}}$

$$|\Phi\rangle := \Phi[\Pi](\vec{x}) = \exp\left(\frac{iS}{\varepsilon^2}\right) \sum_{\underline{k} \in \mathbb{N}^D} c_{\underline{k}} \phi_{\underline{k}}[\Pi](\vec{x}) \tag{20}$$

where ε is the semi-classical scaling parameter.
Notice that $\underline{k}$ is a *multi-index*. It is a tuple that contains D integers called indices. The basis functions $\phi_{\underline{k}}$ depend on the Hagedorn parameter set Π. It is a tuple

$$\Pi := (\vec{q}, \vec{p}, \mathbf{Q}, \mathbf{P}, S)$$

containing the vectors $\vec{q}, \vec{p} \in \mathbb{R}^D$ and complex matrices $\mathbf{Q}, \mathbf{P} \in \mathbb{C}^{D \times D}$. $S \in \mathbb{C}$ is the global phase factor.
Equation (20) contains infinitely many basis functions. To compute the wavepacket we have to truncate the set of all basis functions to a finite set $\mathfrak{K} \subset \mathbb{N}^D$ called *basis shape*. The wavepacket equation becomes

$$\Phi[\Pi](\vec{x}) \approx \exp\left(\frac{iS}{\varepsilon^2}\right) \sum_{\underline{k} \in \mathfrak{K}} c_{\underline{k}} \phi_{\underline{k}}[\Pi](\vec{x})$$

To specify a Hagedorn wavepacket I often use the tuple notation

$$\Phi := (\varepsilon, \Pi, \mathfrak{K}, c)$$

3.2 Vectorial wavepackets

Vectorial Hagedorn wavepackets Ψ consist of multiple components Φ_i

$$\Psi(\vec{x}) := \begin{pmatrix} \Phi_1(\vec{x}) \\ \Phi_2(\vec{x}) \\ \vdots \\ \Phi_N(\vec{x}) \end{pmatrix}$$

The components Φ_i itself are plain scalar wavepackets. They have the scaling parameter ε in common, but the Hagedorn parameter set and the basis shape may differ depending on the concrete type.
Common vectorial wavepacket types are:

Inhomogeneous wavepacket The inhomogeneous wavepacket is the most general case, as its components only share the scaling parameter ε.

$$\Phi_i := (\varepsilon, \Pi_i, \mathfrak{K}_i, c_i)$$

$$\Psi := \left(\varepsilon, \begin{pmatrix} (\Pi_1, \mathfrak{K}_1, c_1) \\ \vdots \\ (\Pi_N, \mathfrak{K}_N, c_N) \end{pmatrix} \right)$$

Homogeneous wavepackets A homogeneous wavepacket is a special case of the inhomogeneous wavepacket. All components share the same Hagedorn parameter set Π.

$$\Phi_i := (\varepsilon, \Pi, \mathfrak{K}_i, c_i)$$

$$\Psi := \left(\varepsilon, \Pi, \begin{pmatrix} (\mathfrak{K}_1, c_1) \\ \vdots \\ (\mathfrak{K}_N, c_N) \end{pmatrix} \right)$$

Gradient The gradient of a scalar Hagedorn wavepacket is a vectorial Hagedorn wavepacket. All components share the same Hagedorn parameter set and the same basis shape, which is an extension of the original wavepacket's basis shape.

$$-i\varepsilon^2 \frac{\partial \Phi}{\partial x_n} := (\varepsilon, \Pi, \mathfrak{K}_{ext}, c'_n)$$

$$-i\varepsilon^2 \nabla \Phi := \left(\varepsilon, \Pi, \mathfrak{K}_{ext}, \begin{pmatrix} (c'_1) \\ \vdots \\ (c'_D) \end{pmatrix} \right)$$

3.3 Class diagram

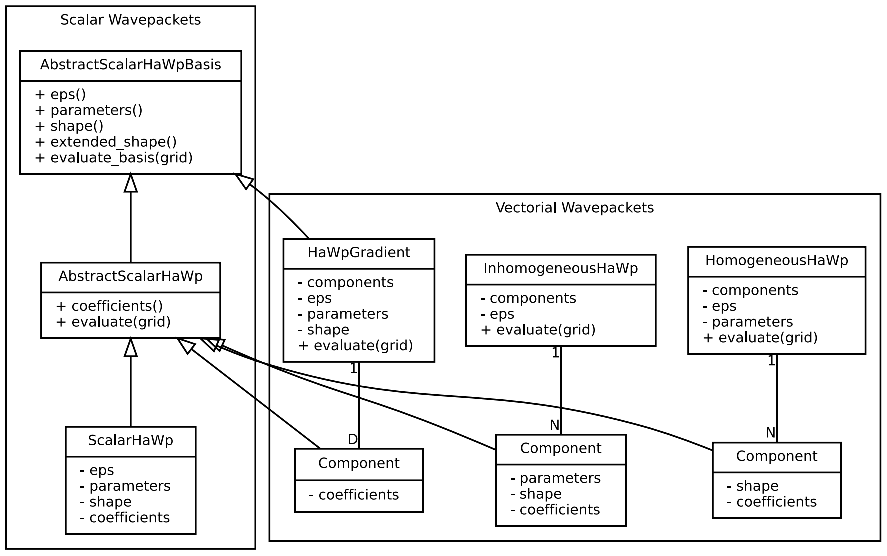


Figure 5: Wavepacket classes and the inheritance hierarchy.

3.3.1 Scalar wavepacket classes

The superclass of all (scalar) Hagedorn wavepackets is **AbstractScalarHaWp-Basis**. It provides read-only access to ε, Π and $\mathfrak{K}$ through its abstract virtual member functions *eps()*, *parameters()* and *shape()* respectively. It does not provide access to the coefficients $\{c_k\}$, because coefficients are not necessary to evaluate the basis functions $\{\phi_k(\vec{x})\}$ on the grid nodes $\vec{x}$, which is implemented by the member function **evaluate_basis(grid)**. This function is not virtual, because it is a service provided by this base class. All classes that inherit AbstractScalarHaWpBasis and thus provide access to ε, Π and $\mathfrak{K}$ automatically gain the ability to evaluate their basis functions.
This superclass further implements the function **extended_shape()**, which returns the basis shape extension $\mathfrak{K}_{ext}$ of the current basis shape $\mathfrak{K}$. It is used for the gradient computation. Creating a basis shape extension is expensive, therefore $\mathfrak{K}_{ext}$ is cached.
The abstract class **AbstractScalarHaWp** extends the *AbstractScalarHaWpBasis*. It provides read-only access to the coefficients $\{c_k\}$ through its virtual member function *coefficients()*. Thus it is able to evaluate itself $\Phi(\vec{x})$ on the grid nodes $\vec{x}$, which is implemented by the member function **evaluate(grid)**.

3.3.2 Vectorial wavepacket classes

Vectorial wavepacket classes are implemented as a composition of scalar wavepackets. Shared aspects are stored in the composition class while different aspects are stored in the component class. Component classes inherit the features of the abstract scalar wavepacket class. Therefore they have to implement access to the parameters ε, Π, $\mathfrak{K}$ and $\{c_k\}$. If a parameter is not stored inside the component class itself, but in the composition class, the component class refers to the parameter stored inside the composition class.
Vectorial wavepacket classes provide a member function to evaluate itself, $\Psi(\vec{x})$. The return type is a vector, instead of a scalar value. This function not only simplifies evaluation, it can considerably improve performance. It exploits that components of homogeneous wavepackets share most basis functions.[1]

3.3.3 Rationale

Abstract superclasses don't provide writable access to parameters.
Editing parameters of a superclass may have side-effects, you have no control over. Some subclasses, like homogeneous wavepackets, share parameters across its components, while others don't, like inhomogeneous wavepacket. It's not known at compile-time, what a parameter change will cause. It may unintendedly change the parameters of many wavepackets.
However, concrete subclasses like ScalarHaWp, HomogeneousHaWp provide writable access to its parameters. This is safe, because you know the concrete wavepacket type at compile-time. Thus the behaviour is specified.

Scalar and vectorial wavepackets don't have a common superclass.
In WaveBlocksND, scalar and vectorial wavepackets actually have a common superclass. It proved to be a bad design choice. The problem is, that the evaluation function yields results with different shape ranks, depending on the actual wavepacket type. It may return a scalar, vector or even a matrix! In python, this behaviour is halfway manageable, thanks to python's highly flexible type system and numpy's broadcasting operations. However, dealing with such behaviour is very hard in C++. Therefore I decided, that scalar and vectorial wavepacket don't have a common superclass.

There is no dedicated class to represent a gradient of a vectorial wavepacket.
Applying the gradient operator to a vectorial wavepacket is trivial: Just apply it to its components.

$$-i\varepsilon^2\nabla\Psi \equiv \begin{pmatrix} -i\varepsilon^2\nabla\Phi_1 \\ \vdots \\ -i\varepsilon^2\nabla\Phi_N \end{pmatrix}$$

[1] See section (3.5).

It is an overkill to introduce a new class, just to replace a loop. Furthermore, storing all coefficients of $-i\varepsilon^2\nabla\Psi$ occupies $\mathcal{O}(D \cdot N \cdot M)$ space. Using a basis shape of a realistic size $M = 1'000'000$, and wavepacket dimensionality $D = 10$, one can easily run out of memory.

3.4 Basis evaluation

All basis function values are evaluated recursively. Once we have the basis function value at an anchor node $\underline{k}$ and the basis functions values on all backward neighbours $\underline{k} - \underline{e}^d$, we can compute all forward neighbours by

$$\begin{pmatrix} \phi_{\underline{k}+\underline{e}^1} \\ \vdots \\ \phi_{\underline{k}+\underline{e}^D} \end{pmatrix} = \left(\sqrt{\frac{2}{\varepsilon^2}} \mathbf{Q}^{-1} (\vec{x} - \vec{q}) \phi_{\underline{k}} - \mathbf{Q}^{\mathrm{H}} \mathbf{Q}^{-\mathrm{T}} \begin{pmatrix} \sqrt{k_1} \phi_{\underline{k}-\underline{e}^1} \\ \vdots \\ \sqrt{k_D} \phi_{\underline{k}-\underline{e}^D} \end{pmatrix} \right) \oslash \begin{pmatrix} \sqrt{k_1 + 1} \\ \vdots \\ \sqrt{k_D + 1} \end{pmatrix} \tag{21}$$

where the operator $\oslash$ denotes a component-wise division.

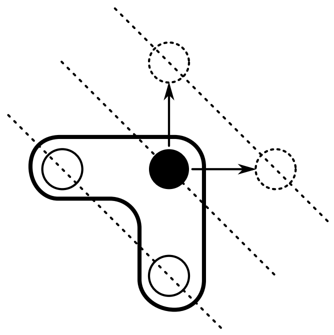


Figure 6: Stencil of a 2-dimensional wavepacket

The root basis function $\phi_{\underline{0}}$ is evaluated by

$$\phi_{\underline{0}}[\Pi](\vec{x}) := (\pi\varepsilon^2)^{-\frac{D}{4}} (\det \mathbf{Q})^{-\frac{1}{2}} \exp\left(\frac{i}{2\varepsilon^2} \langle (\vec{x} - \vec{q}), \mathbf{P}\mathbf{Q}^{-1} (\vec{x} - \vec{q}) \rangle + \frac{i}{\varepsilon^2} \langle \vec{p}, (\vec{x} - \vec{q}) \rangle \right) \tag{22}$$

Basis functions with some negative indices are zero.

3.4.1 Implementation

There are two possibilities to implement this recursion scheme:

Scatter Type Strategy We maintain a queue of anchor nodes whose basis functions already have been evaluated. We subsequently remove one feasible(!) entry $\phi_{\underline{k}}$ of this queue, evaluate the basis functions of all forward neighbours $\phi_{\underline{k}+\underline{e}^d}$ and put these nodes back into the queue. To avoid multiple evaluation of basis functions, we have to check, whether a forward neighbour already has been computed, using a set. Therefore the scatter type strategy is very bad suited for multi-threading since threads need to synchronize access to this set.

Gather Type Strategy We maintain a queue of not yet evaluated forward neighbours $\phi_{\underline{k}+\underline{e}^d}$. Entries of this queue can be evaluated since there exists a suitable[2] anchor node $\phi_{\underline{k}}$ we can apply the stencil on. We subsequently remove one entry of the queue and evaluate its basis function. Then we check, whether *all* backward neighbours of the child node $\phi_{\underline{k}+\underline{e}^d}$ are computed. If this is the case, we put all forward neighbours of $\phi_{\underline{k}+\underline{e}^d}$ into the queue.

A straight-forward implementation has at least one of the following difficulties, that require complex data structures, such as dictionaries or would require excessive thread synchronisation.

A We need to ensure that an already computed node is a valid anchor node (all backward neighbours has been computed too) so that we can compute its forward neighbours.

B We need to avoid multiple evaluation of the same node.

My rather simple solution is to split the basis shape into slices. The s-th slice contains all nodes $\underline{k}$ that fulfil $\sum_{d=1}^{D} k_d = s$.

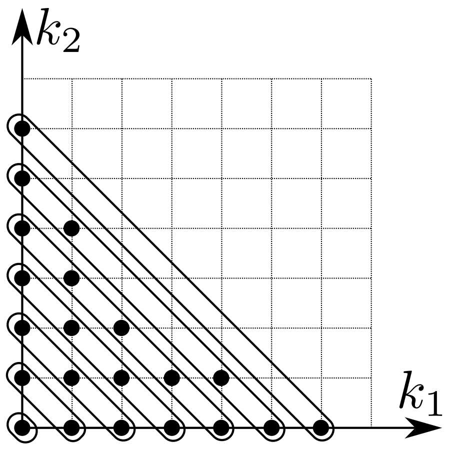


Figure 7: Slicing of a 2-dimensional basis shape.

Using an adequate[3] shape, the recursion scheme implies that if we had computed all basis functions of the parent slice $\mathfrak{S}^{s-1}$ and of the current slice $\mathfrak{S}^{s}$, then we can compute all basis functions of the child slice $\mathfrak{S}^{s+1}$ without having

[2] $\phi_{\underline{k}}$ and all $\phi_{\underline{k}-\underline{e}^d}$'s have been evaluated.

[3] Fundamental property of a basis shape; see equation (1).

difficulty **A**. Furthermore, if we use the gather type strategy where we iterate through the child slice and compute its basis functions, we automatically avoid difficulty **B**.
My solution has two compelling advantages:

Simple data structures We can store the values of basis functions in an array. We need a *shape enumeration* that maps shape nodes to ordinals, used as array offsets.

Easy parallelisation Parallelise the wavepacket evaluation is simple: Put an OpenMP pragma to the loop, that runs over the child slice. All previous slices have been completely computed, therefore the only synchronisation mechanism needed is a barrier at the end of the slice.

Algorithm 4: Recursive basis evaluation of hagedorn wavepacket

```
Input  : Number of slices $S$
Input  : Shape enumeration (slices) $\mathfrak{K} = \left(\mathfrak{S}^0, \dots, \mathfrak{S}^{S-1}\right)$
Output: Evaluated basis functions (slices) $\phi := \left(\phi^0, \phi^1, \dots, \phi^{S-1}\right)$
Compute ground state
$\phi^0 \leftarrow \left\{\phi_{\underline{0}}[\Pi](\vec{x})\right\}$
Loop over all slices
foreach $s \leftarrow 0$ to $S-2$ do
    Loop over all nodes of next slice
    for $\underline{k}^{s+1} \in \mathfrak{S}^{s+1}$ do
        Find suitable anchor node (in anchor slice $\mathfrak{S}^s$)
        $\alpha \leftarrow \emptyset$
        for $d \in \{1, \dots, D\}$ do
            if $k_d^{s+1} > 0$ then
                $\alpha \leftarrow d$
            end
        end
        Here is our anchor node $\underline{k}^s$
        $\underline{k}^s \leftarrow \underline{k}^{s+1} - \underline{e}^{\alpha}$
        Compute contribution of anchor node
        $m \leftarrow \frac{\sqrt{2}}{\varepsilon}\left(\mathbf{Q}^{-1}(\vec{x} - \vec{q})\right)_{\alpha} \phi_{\underline{k}^s}^s$
        Compute contribution of anchor node's backward neighbours
        $b \leftarrow 0$
        for $d \in \{1, \dots, D\}$ do
            if $k_d^s > 0$ then
                $b \leftarrow b + \sqrt{k_d^s}\left(\mathbf{Q}^{\mathrm{H}}\mathbf{Q}^{-\mathrm{T}}\right)_{\alpha d} \phi_{\underline{k}^s - \underline{e}^d}^{s-1}$
            end
        end
        Compute basis function at $\underline{k}^{s+1}$
        $\phi_{\underline{k}^{s+1}}^{s+1} \leftarrow \frac{1}{\sqrt{k_{\alpha}^s + 1}}(m - b)$
    end
end
```

3.4.2 Algorithmic complexity

Evaluating D-dimensional wavepackets involves evaluating N basis functions. To evaluate a basis function we have to lookup the ordinals of D backward neighbours which costs $\mathcal{O}(D \log S)$ because we use binary search. S is the size of the largest slice. A shape has at least $\mathcal{O}(D \log N)$ slices (hypercubic shape). Therefore

$$\mathcal{O}(S) = \mathcal{O}\left(\frac{N}{D \log N}\right) \approx \mathcal{O}(N)$$

Gathering values of previous basis functions $\phi_{\underline{k}-\underline{e}^d}$ and computing $\phi_{\underline{k}}$ has complexity $\mathcal{O}(D)$. Thus evaluating all basis functions has complexity

$$N \cdot (\mathcal{O}(D \log S) + \mathcal{O}(D)) = \mathcal{O}(ND \log S) \approx \mathcal{O}(N \log^2 N)$$

when assuming $D \approx \log N$. Computing the dot-product $\sum c_{\underline{k}} \phi_{\underline{k}}$ costs just $\mathcal{O}(N)$ thus the final *asymptotic* complexity is

$$\mathcal{O}(N \log^2 N) \tag{23}$$

3.4.3 Evaluating on multiple quadrature points at once

To evaluate the basis function $\phi_{\underline{k}}$, we need all backward neighbours $\phi_{\underline{k}-\underline{e}^d}$. Unfortunately the program spends much time on looking up the ordinals of these backward neighbours. To improve the ratio of lookup time versus computation time, I've implemented the possibility to evaluate the basis functions $\phi_{\underline{k}}$ on multiple grid nodes at once. The number of grid nodes is a template parameter.

Figure (8) shows the effect of one evaluation with N grid nodes compared to N evaluations with one grid node each. It shows that the multi-node evaluation of a 10-dimensional wavepacket is on average up to 3 times faster than when using single-node evaluations. The speedup is relatively independent on the dimensionality. Deviations are caused by compiler optimizations and inner working of the Eigen library.

Notice, that you won't evaluate a wavepacket on let's say 100'000 quadrature points at once. You will split these 100'000 quadrature points into chunks, each containing for example 5 nodes. Then you evaluate the wavepacket on every chunk. Compared to Python, in C++, there is no performance advantage in passing large number of nodes to the evaluation function. Far from it, you eventually run out of memory! Remember, that the evaluation function allocates memory for $\mathcal{O}(3SN)$ basis functions, where S is the size of the largest slice and N is the number of grid nodes.

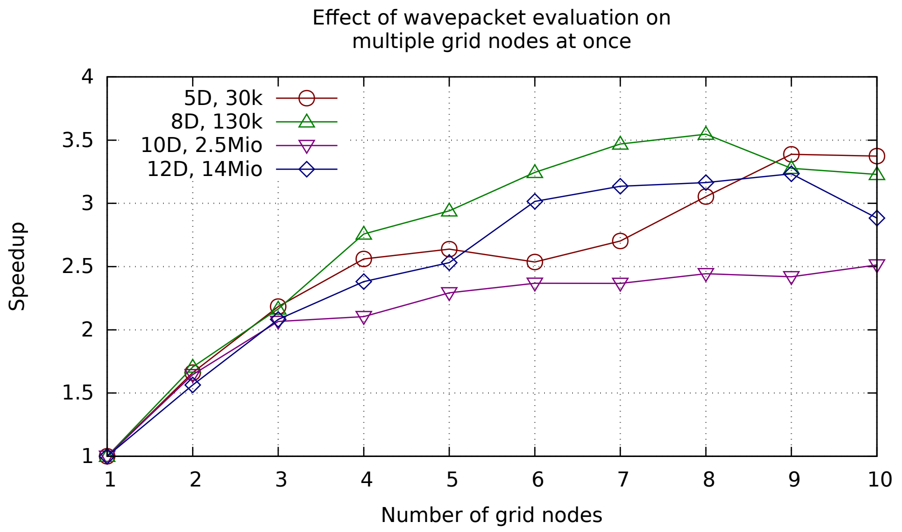


Figure 8: Effect of wavepacket evaluation on multiple quadrature points at once. The speedup is the cumulative run-time of single-node evaluations divided by the run-time of the multi-node-evaluation.

3.5 Optimized evaluation of vectorial wavepackets

The naive approach to evaluate vectorial wavepackets, is to separately evaluate each component. For inhomogeneous wavepackets there is no better way. However components of homogeneous wavepackets share the same Hagedorn parameter set Π and therefore share some basis functions. How many, depends on how large the overlap of the basis shapes is. Basis shapes of components typically are very similar, therefore the benefit of an algorithm, that exploits shape overlaps, can be considerable.
One way to achieve this, is to compute the union of the basis shapes of all components:

$$\mathfrak{K}^{\cup} := \mathfrak{K}^1 \cup \mathfrak{K}^2 \cup \cdots \cup \mathfrak{K}^N$$

Then we compute the basis functions governed by the basis shape union:

$$\left\{\phi_{\underline{k}} \mid \underline{k} \in \mathfrak{K}^{\cup}\right\}$$

Finally, for each wavepacket component Φ_n, we compute the dot product of its coefficients $c_{n,\underline{k}}$ with the subset $\left\{\phi_{\underline{k}} \mid \underline{k} \in \mathfrak{K}^n\right\}$:

$$\Phi_n(\vec{x}) = \sum_{\underline{k} \in \mathfrak{K}^n} c_{n,\underline{k}} \phi_{\underline{k}}(\vec{x})$$

Remark: My implementation does the above computation slice by slice to save memory; the same way as for a scalar wavepacket.

To quantify the improvement, I benchmarked the above algorithm on a homogeneous wavepacket with varying number of components. All components have the same basis shape which is the best case for the algorithm (since in this case, shapes completely overlap). The 10-dimensional basis shape contains one million nodes. The results (see figure 9) show that the optimized algorithm can be six times faster than a naive loop over all wavepacket components.

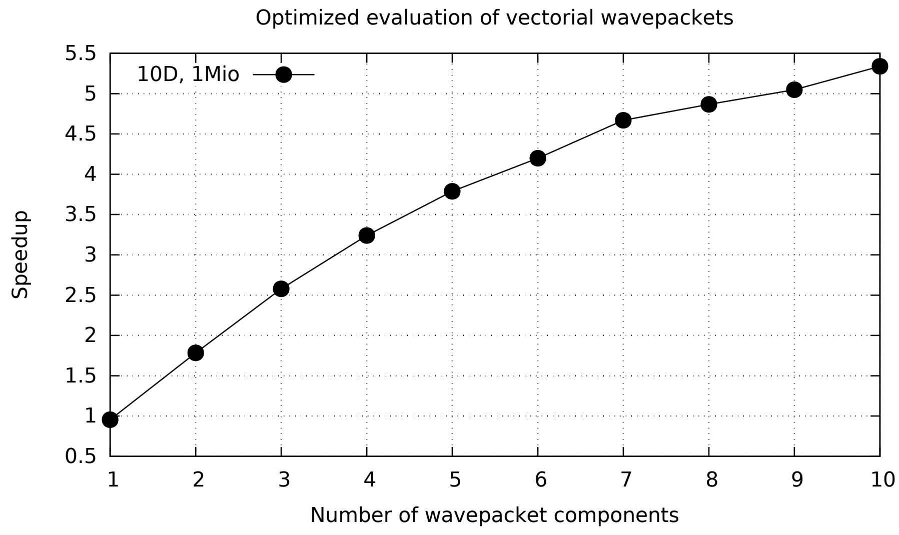


Figure 9: Achievable speedup compared to naive evaluation of a vectorial wavepacket. The speedup is the run-time of the naive evaluation divided by the run-time of the optimized evaluation.

There is an upper bound to the achievable speedup, as though the algorithm avoids multiple evaluation of the same basis functions, it doesn't reduce time spent on computing the dot products for each component. Thus when increasing the number of wavepacket components, computing the dot-product becomes more and more the bottleneck.

3.6 Gradient computation

Applying the gradient operator to a scalar D-dimensional Hagedorn Wavepacket $\Phi = (\varepsilon, \Pi, \mathfrak{K}, c)$ yields a vectorial Hagedorn Wavepacket $\Psi = -i\varepsilon^2 \nabla \Phi$ with D components. The wavepacket gradient has the same parameter set Π as the original wavepacket Φ, but has a new coefficients set $\vec{c'}$ given by

$$\vec{c'}_{\underline{k}} = c_{\underline{k}} \vec{p} + \sqrt{\frac{\varepsilon^2}{2}} \left(\overline{\mathbf{P}} \begin{pmatrix} c_{\underline{k}+\underline{e}^1} \sqrt{k_1 + 1} \\ \vdots \\ c_{\underline{k}+\underline{e}^D} \sqrt{k_D + 1} \end{pmatrix} + \mathbf{P} \begin{pmatrix} c_{\underline{k}-\underline{e}^1} \sqrt{k_1} \\ \vdots \\ c_{\underline{k}-\underline{e}^D} \sqrt{k_D} \end{pmatrix} \right) \tag{24}$$

Some coefficients are non-zero outside of the original basis shape. Thus the wavepacket gradient needs a extended basis shape $\mathfrak{K}_{ext}$ with the following property

$$\underline{k} \in \mathfrak{K} \Rightarrow (\underline{k} \in \mathfrak{K}_{ext}) \wedge \left(\underline{k} + \underline{e}^1 \in \mathfrak{K}_{ext} \wedge \cdots \wedge \underline{k} + \underline{e}^D \in \mathfrak{K}_{ext}\right) \tag{25}$$

3.6.1 How it was implemented in WaveBlocksND

The original python code uses the *scatter-type algorithm*. It loops over all $\underline{k} \in \mathfrak{K}$ and computes the contribution to its neighbours in $\mathfrak{K}_{ext}$. The reason to use this strategy is, how the shape extension $\mathfrak{K}_{ext}$ is implemented in WaveBlocksND. The extension contains far more entries than required by equation (25). Since the scatter-type strategy loops over $\mathfrak{K}$ and not over the oversized $\mathfrak{K}_{ext}$, no computation power is wasted.

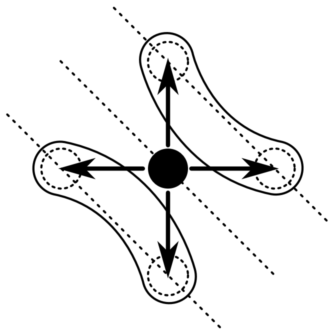


Figure 10: Stencil of the scatter-type algorithm

3.6.2 How I have implemented it

As my implementation of the shape extension $\mathfrak{K}_{ext}$ only contains nodes that are required by equation (25), there is no necessity to use the scatter type strategy. Instead my implementation uses the *gather-type algorithm*. It loops over all $\underline{k} \in \mathfrak{K}_{ext}$, gathers the values of its contributors and finally computes the new coefficients $c'_{\underline{k}}$ according to equation (24). I decided to use this strategy, because it is much easier to parallelise than the scatter-type algorithm. For each node, the program just writes to one instead to $2D$ locations. This greatly simplifies parallelisation.

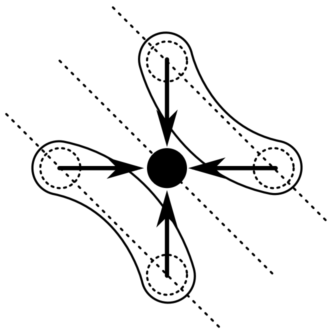


Figure 11: Stencil of the gather-type algorithm.

Algorithm 5: Computes coefficients of the wavepacket gradient $-i\varepsilon^2\nabla\Phi$

```
Input  : Wavepacket dimension $D$
Input  : Scaling parameter $\varepsilon$
Input  : Wavepacket parameters $\Pi = (\vec{q}, \vec{p}, \mathbf{Q}, \mathbf{P}, S)$
Input  : Basis shape $\mathfrak{K}$
Input  : Basis shape extension $\mathfrak{K}_{ext}$
Input  : Wavepacket coefficients $\boldsymbol{c} \in \mathbb{C}^{N\times 1}$ (array of scalars)
Output: Coefficients of wavepacket gradient $\vec{\boldsymbol{c'}} \in \mathbb{C}^{N\times D}$ (array of vectors)
for $\underline{k} \in \mathfrak{K}_{ext}$ do
    Compute contribution of centre node
    $m \leftarrow 0 \in \mathbb{C}$
    if $\underline{k} \in \mathfrak{K}$ then
        $m \leftarrow c_{\underline{k}}$
    end
    Compute contribution of backward neighbours
    $\vec{b} \leftarrow 0 \in \mathbb{C}^D$
    for $d \in \{1, \dots, D\}$ do
        if $\underline{k} - \underline{e}^d \in \mathfrak{K}$ then
            $b_d \leftarrow \sqrt{k_d}\, c_{\underline{k}-\underline{e}^d}$
        end
    end
    Compute contribution of forward neighbours
    $\vec{f} \leftarrow 0 \in \mathbb{C}^D$
    for $d \in \{1, \dots, D\}$ do
        if $\underline{k} + \underline{e}^d \in \mathfrak{K}$ then
            $f_d \leftarrow \sqrt{k_d + 1}\, c_{\underline{k}+\underline{e}^d}$
        end
    end
    Merge contributions of every node to get the coefficients of the gradient
    $\vec{c'}_{\underline{k}} \leftarrow m\vec{p} + \frac{\varepsilon}{\sqrt{2}}(\overline{\mathbf{P}}\vec{f} + \mathbf{P}\vec{b})$
end
```

4 Performance

4.1 Wavepacket evaluation

I benchmarked the wavepacket evaluation and gradient construction on the Euler cluster[4] [5] using one thread. I used a hyperbolic cut shape with limits 10, and varied the sparsity between 2^D and 3^D to obtain different numbers of basis functions for a given dimensionality D.

[4]Measurements done in year 2015, the CPU is Intel Xeon E5-2697v2 (2.7 GHz).

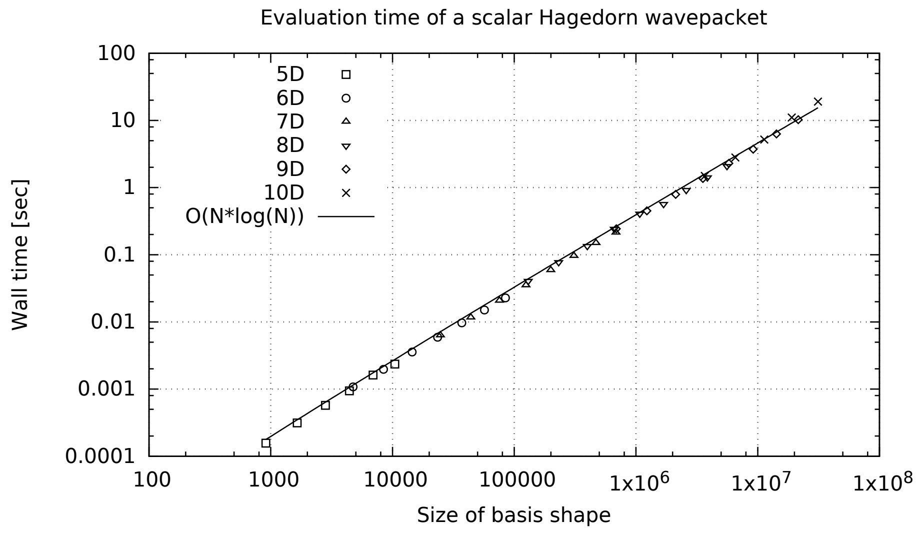


Figure 12: Time to evaluate a scalar wavepacket on one node $\vec{x}$.

The plot (12) shows that evaluating a Hagedorn wavepacket with one million basis functions on one grid node $\vec{x}$ took 150 milliseconds.
Evaluating a Hagedorn wavepacket has an asymptotic run-time complexity[5] $\mathcal{O}(N \log^2 N)$ where N is number of basis shape nodes. The measurements of runtime suggest $\Theta(N \log N)$, which is better because asymptotic limit has not been reached yet.

[5]Assuming $\mathcal{O}(D) \approx \mathcal{O}(\log N)$. See section (3.4.2) for details.

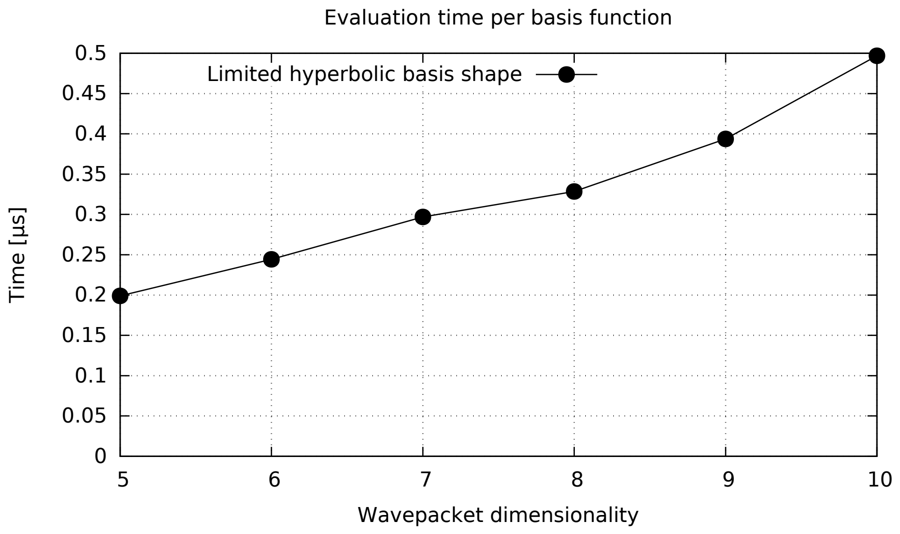


Figure 13: This chart shows the effect of dimensionality to evaluation time. It shows the time needed to evaluate *one* basis function on one node $\vec{x}$ depending on dimensionality.

4.2 Gradient Computation

Measurements reveal that gradient construction, i.e. computing the coefficients of the gradient wavepacket, takes approximately six times longer than wavepacket evaluation. Evaluating the gradient on one grid node $\vec{x}$ takes 2.5 times longer. That means, if the evaluation of some wavepacket Φ_0 takes $1ms$ time, than computing the coefficients of the gradient wavepacket $-i\varepsilon^2\nabla\Phi_0$ takes $6ms$ time. Evaluating the gradient $(-i\varepsilon^2\nabla\Phi_0)(\vec{x})$ on one node requires $2.5ms$ time.

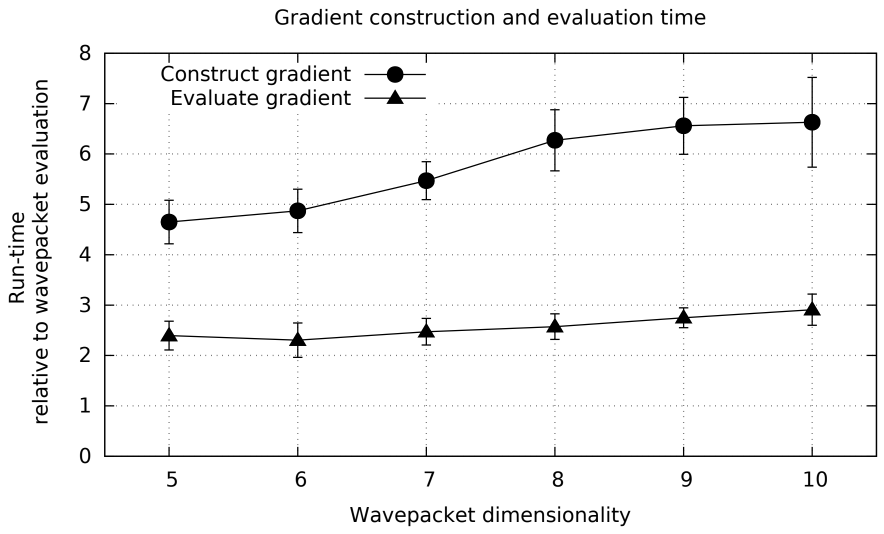


Figure 14: This plot shows two series: One depicts the time to construct a wavepacket gradient. The other depicts the time to evaluate a wavepacket gradient. Time is shown *relative* to wavepacket evaluating. Error bars depict the standard deviation.

4.3 Comparison to WaveBlocksND

4.3.1 Wavepacket evaluation

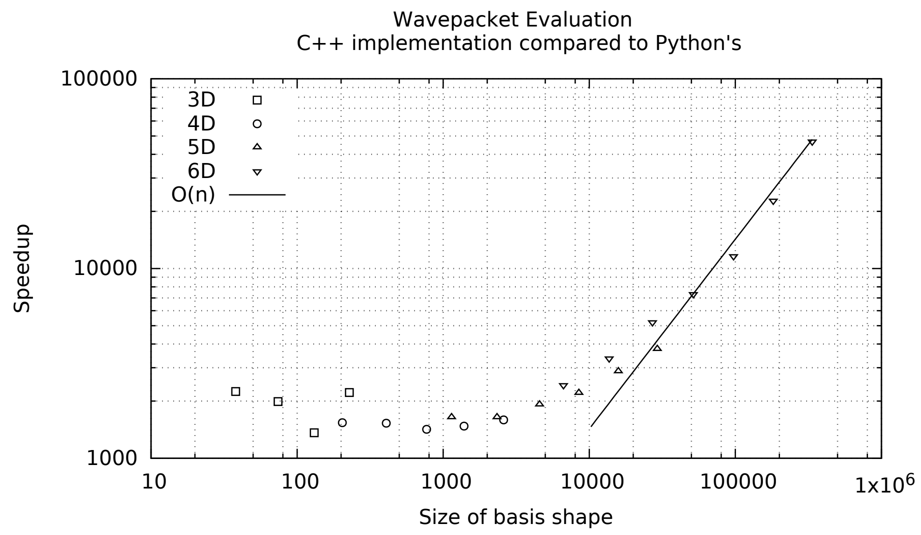


Figure 15: Wavepacket evaluation time in C++ compared to WaveBlocksND. WaveBlocksND uses the **slim recursion** routine.

Apparently, the C++ implementation is asymptotically better than WaveBlocksND's slim recursion by a factor of $\mathcal{O}(N)$. My implementation could have slightly better run-time complexity due to slicing, but by no means $\mathcal{O}(N)$. The run-time complexity of the slim recursion is quadratic to the basis shape size. This behaviour is caused by a flaw, somewhere inside the slim recursion. Due to this discovery, I've done all further comparisons using the fat recursion routine, which behaves well, as seen in the figure below. On average, my implementation evaluates a Hagedorn wavepacket 2000 times faster than WaveBlocksND.

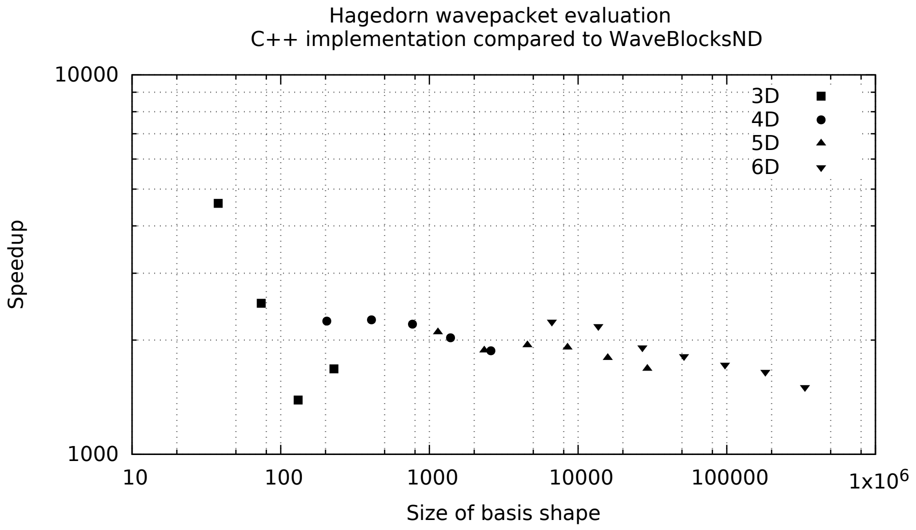


Figure 16: Wavepacket evaluation time in C++ compared to WaveBlocksND. WaveBlocksND uses the **fat recursion** routine.

4.3.2 Gradient computation

Below is a benchmark of the gradient computation. The employed wavepacket has a limited hyperbolic cut basis shape[6]. The results reveal run-time complexity improvements of my implementation.

[6] The cut-off limit is 100, sparsity varies between 2^D and 3^D.

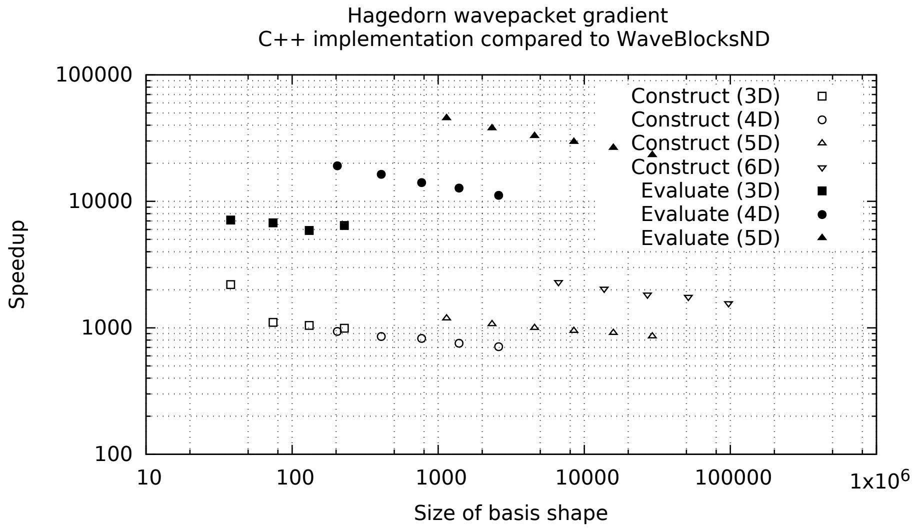


Figure 17: Wavepacket gradient construction and evaluation time in C++ compared to WaveBlocksND. Speedup means the run-time of WaveBlocksND divided by the run-time of my implementation. This chart contains two series: Empty bullets show the speedup when constructing the wavepacket gradient. Filled bullets show the speedup when evaluating the wavepacket gradient.

I've changed the way how shape extensions are generated. WaveBlocksND generates shape extensions that contain many superfluous nodes. The ratio of superfluous to essential nodes grows exponentially with the shape dimensionality.[7] This results in a wavepacket gradient that has $\mathcal{O}(\exp(D)N)$ basis functions instead of $\mathcal{O}(N)$.
WaveBlocksND mitigates this issue by constructing gradients using the scatter-type algorithm, since the scatter-type algorithm simply ignores superfluous nodes. This works well for low dimension. Nevertheless, the cost of allocating and zero-initialize oversized arrays becomes apparent from dimensionality six. My implementation generates shape extensions that only contain essential nodes. The *evaluate*-series reflect the advantage of such tight shape extensions when evaluating wavepacket gradients.

4.4 First steps towards parallelisation using OpenMP

I paid attention to possible parallelisation right from the start. The evaluation of Hagedorn wavepackets is of quite serial nature, due to its recursive scheme. But basis functions on the same slice can be evaluated independently. This was one of my main reasons to split basis shapes into slices. I added a simple *pragma omp parallel for* to the loop, that runs over all nodes of the same slice.

[7] This effect is very apparent when using hyperbolic cut shapes.

I've done some measurements on the Euler cluster, using up to 12 threads. The plot below (18) shows, that the evaluation of large and high dimensional wavepackets scales reasonably well.

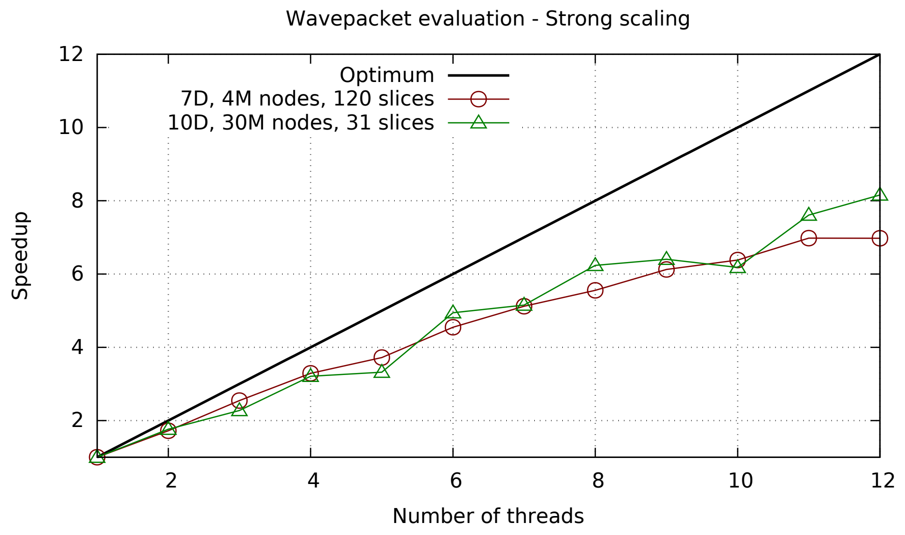


Figure 18: Strong scaling plot of the Hagedorn wavepacket evaluation. It shows two measurements: One of a 7-dimensional wavepacket with four million basis functions, the other of a 10-dimensional wavepacket with 30 million basis functions.

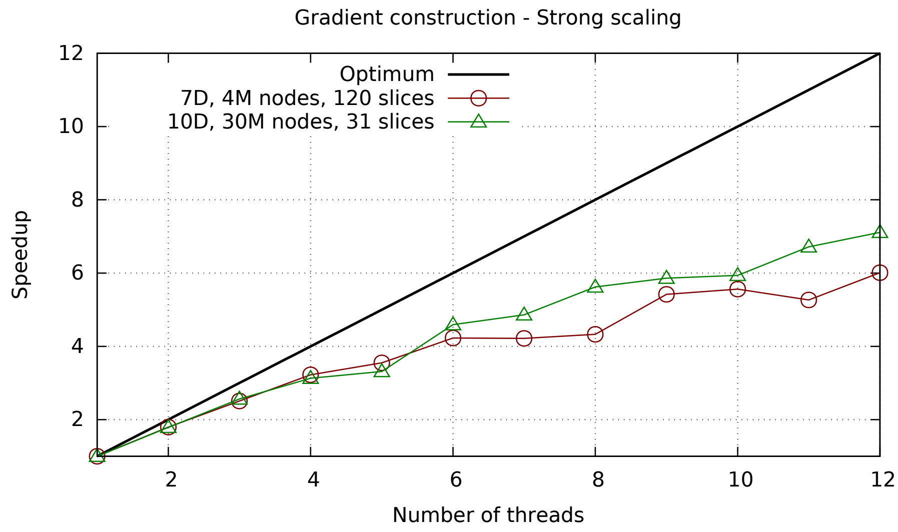


Figure 19: Strong scaling plot of the wavepacket gradient construction. It shows two measurements: One of a 7-dimensional wavepacket with four million basis functions, the other of a 10-dimensional wavepacket with 30 million basis functions.

References

[1] R. Bourquin. Wavepacket propagation in D-dimensional non-adiabatic crossings. mathesis, 2012. http://www.sam.math.ethz.ch/~raoulb/research/master_thesis/tex/main.pdf. 2

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

[3] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. 2

[4] George A. Hagedorn. Raising and lowering operators for semiclassical wave packets. *Annals of Physics*, 269(1):77–104, 1998. 2

[5] Swiss Federal Institute of Technology Zurich. Euler cluster. http://clusterwiki.ethz.ch/brutus/Getting_started_with_Euler, 2015. 23