ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Serialization of Hagedorn Wavepackets in C++ with HDF5 Interface

Bachelor Thesis

Florian Frei

August 25, 2016

Advisors: Prof. Dr. Ralf Hiptmair, Dr. Vasile Grădinaru
Seminar for Applied Mathematics, ETH Zürich

Abstract

It is of general interest in physics and chemistry to find viable algorithms to solve the time-dependent Schrödinger equation. To solve this equation directly is in most cases not feasible so different approximations have to be made. Such an approximation is the semiclassical approach with *Hagedorn* wavepackets. To test this approach for reliability a python implementation was created. With this implementation it is possible to easily test many starting configurations which results in a lot of data. To handle this huge amount of data a binary format is required because only this format can be efficiently compressed. Hence it is a good idea to use a well-known and commonly used binary data format such as *HDF5*. With *h5py* there exists also a reasonable python interface. Since not all problems are solvable in a short period of time, and also due to the fact that Python is an interpreted language, the code was ported to C++ to reduce execution time. Therefore also the IO-operations have to be ported to C++. The *HDF5* library supports also a C++ interface. This however is not as easy comprehensible as the Python interface and thus this thesis explains its core functionality with respect to this application. Furthermore since the same data hierarchy is implemented as in Python the generated data is directly comparable. This data test was done in C++ with the well-known *GoogleTest* framework, which can compare two data files generated from Python and C++ respectively.

Contents

Contents **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 Background 1

2 HDF5 C++ Interface **5**
2.1 Overview 5
2.2 Internal types and states 6
2.3 H5File 7
2.4 Group 7
2.5 DataSet 8
2.6 DataType 9
2.7 DataSpace 10
2.8 PropList 10
2.8.1 DSetCreatePropList 11
2.9 Attribute 11

3 Implementation **13**
3.1 Link to Eigen library 13
3.2 DataType Declaration 13
3.3 Constructor 14
3.4 Write options 15
3.5 DataSet paths 15
3.6 Prestructure 15
3.7 Selection 16
3.8 Transformation 17
3.9 Writing 18
3.10 Extension 18
3.11 Update 18

3.12 Poststructure 19
3.13 Simulation in a picture 19
3.14 Usage in Simulation 20

4 Data Test **21**
4.1 Introduction to GoogleTest 21
4.2 The Main File 21

5 Conclusion **23**

A Appendix **25**
A.1 Code of 2D harmonic oscillator 25

Bibliography **29**

Chapter 1

Introduction

1.1 Motivation

To test the reliability of a numerical method or an approximation, e.g. such as in [2], an implementation in Python is sufficient. Python is easy to write and understand and the code can also be used as a base e.g. for a C++ version. A Python framework to solve the time-dependent Schrödinger equation with a semiclassical approach with *Hagedorn* wavepackets is already available on github [5]. This can be used for example to test different propagation methods, quadrature functions or integration schemes. In case longer simulations or solving higher dimensional problems is desired the execution time in Python gets unfeasible. Therefore it is necessary to port the Python implementation to C or C++ which are highly efficient. The core implementation is already available in C++ on github [4] but further improvement is desired. This implementation currently uses an external project [10] for writing data in *HDF5* format. This project is sufficient if the user supervises the generated data. The goal of this project is to use the *HDF5* interface [13] directly to write data in a comparable way to Python. This further allows to implement a data test which can compare data files between Python and C++. This was done with the well-known *GoogleTest* [7] framework.

1.2 Background

In quantum physics the most prominent problems are governed by the time-dependent Schrödinger equation 1.1

$$i\hbar \frac{\partial}{\partial t} |\psi\rangle = H |\psi\rangle \tag{1.1}$$

where H is the Hamiltonian, $\psi(\underline{x}, t)$ represents the complex valued wave function dependent on position $\underline{x} \in \mathbb{R}^d$ and time t and $\langle\psi|\psi\rangle$ is the proba-

bility density of electrons. This can be found in the most basic courses and books about quantum physics. This equation can be rescaled in a semiclassical setting to obtain the nuclear Schrödinger equation:

$$i\hbar\partial_t\psi = \left(-\frac{\hbar^2}{2}\Delta_x + V(\underline{x})\right)\psi\,. \tag{1.2}$$

The goal is to find for $\psi = \psi(\underline{x}, t)$ depending on the spatial variables $\underline{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ and the time $t \in \mathbb{R}$. In this context Δ_x is the Laplace operator and V is a smooth real-valued potential, e.g. an electronic energy surface in the time-dependent Born-Oppenheimer approximation. Nevertheless there are still many challenges involved in solving this equation 1.2. One of these is the high dimensionality of this equation. A molecule with N nuclei where each of them has three degrees of freedom results in $3N$ coordinates. For example the simple molecule CO_2 has already $d = 9$ degrees of freedom. Another challenge is the multiple scales governed by the small parameter $\hbar = \varepsilon^2$ in case of CO_2 it results in $\hbar \approx 0.0058$. Also the actual solution has frequencies of order $1/\hbar$ which are hard to reproduce for small $\hbar$ on a finite uniform grid as required the widely used Fourier based approach. Further there is the problem of long time evolution.

The semiclassical method makes use of the raising and lowering operators for semiclassical wavepackets [16]. The raising and lowering operators $\mathcal{R}$ and $\mathcal{L}$ produce a complete L^2-orthonormal set of basis functions $\varphi_{\underline{k}}(\underline{x}) = \varphi^{\hbar}_{\underline{k}}[\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}](\underline{x})$ (for multi-indices $\underline{k} = (k_1, \ldots, k_d)$ with non-negative integers k_j) obeying the three-term recurrence:

$$\left(\sqrt{k_j + 1}\varphi_{\underline{k}+\underline{e}^j}\right)_{j=1}^{d} = \sqrt{\frac{2}{\hbar}}\mathbf{Q}^{-1}\left(\underline{x} - \underline{q}\right)\varphi_{\underline{k}} - \mathbf{Q}^{-1}\overline{\mathbf{Q}}\left(\sqrt{k_j}\varphi_{\underline{k}-\underline{e}^j}\right)_{j=1}^{d}.$$

Therefore we have Hagedorn wavepackets of the following form:

$$\begin{aligned}\varphi^{\hbar}_{\underline{0}}[\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}](\underline{x}) \;&= (\pi\hbar)^{-\frac{d}{4}}(\det\mathbf{Q})^{-\frac{1}{2}} \times \\ &\exp\left(\frac{i}{2\hbar}\left(\underline{x} - \underline{q}\right)^{\mathrm{T}}\mathbf{P}\mathbf{Q}^{-1}\left(\underline{x} - \underline{q}\right) + \frac{i}{\hbar}\underline{p}^{\mathrm{T}}\left(\underline{x} - \underline{q}\right)\right)\end{aligned}$$

where $\underline{q} \in \mathbb{R}^d$ and $\underline{p} \in \mathbb{R}^d$ represent the position and momentum, respectively, and $\mathbf{Q}$ and $\mathbf{P}$ are complex $d \times d$ matrices satisfying some compatibility relations. For a detailed explanation see [9]. In [9] is also a basic time propagation algorithm described. Further improvement and a proof about convergence of semiclassical wavepackets based time-splitting was presented in [11].

This method was implemented by R. Bourquin in his master thesis [3] in Python and is available on github [5]. This code can be used in applications for example tunneling dynamics and spawning [12] or non-adiabatic transition near avoided crossings [6]. Until now only the core of this implementation was already ported to C++. Hagedorn wavepackets was done by M. Bösch [1], inner products and quadrature by B. Vartok [18] and matrix potentials by L. Miserez [17]. This core implementation is available on github [4]. In this project the goal was to enlarge the C++ code functionality with a more useful data serialization. The goal included compatibility between the Python and C++ implementation on how data is stored using the *HDF5* format. This further allowed to use the *GoogleTest* framework to write a data test for the two *HDF5* files. This can be convenient because now we can test if both implementation generate the same data or if there could be a bug in the code.

Chapter 2

HDF5 C++ Interface

For simplicity the namespaces of the standard library `std` and the HDF5 library `H5` are omitted throughout this thesis.

2.1 Overview

The acronym HDF stands for *hierarchical data format* meaning this binary format allows to construct a hierarchy on internal objects after the users demand. This hierarchy is quite similar to a file system where data is ordered in a directory/sub-directory manner beginning at the root "/". For more detailed information consult the official C [14] and C++ [13] documentation. The reason why also the C documentation is very important is based on the fact that the C++ implementation is mostly a nice wrapper based on the C implementation. For the exact dependency see table 2.1.
To use the C++ language features to its most capabilities inheritance is used.

HDF5 C APIs	HDF5 C++ Classes
Attribute Interface (H5A)	Attribute
Datasets Interface (H5D)	DataSet
Error Interface (H5E)	Exception
File Interface (H5F)	H5File
Group Interface(H5G)	Group
Identifier Interface (H5I)	IdComponent
Property List Interface (H5P)	PropList and subclasses
Dataspace Interface (H5S)	DataSpace
Datatype Interface (H5T)	DataType and subclasses

Figure 2.1: Table of correspondence between C and C++

This allows reuse of functions, objects and properties over many hierarchy

layers. This enforces a strict dependence when sharing or creating objects. This hierarchy is shown in figure 2.2.

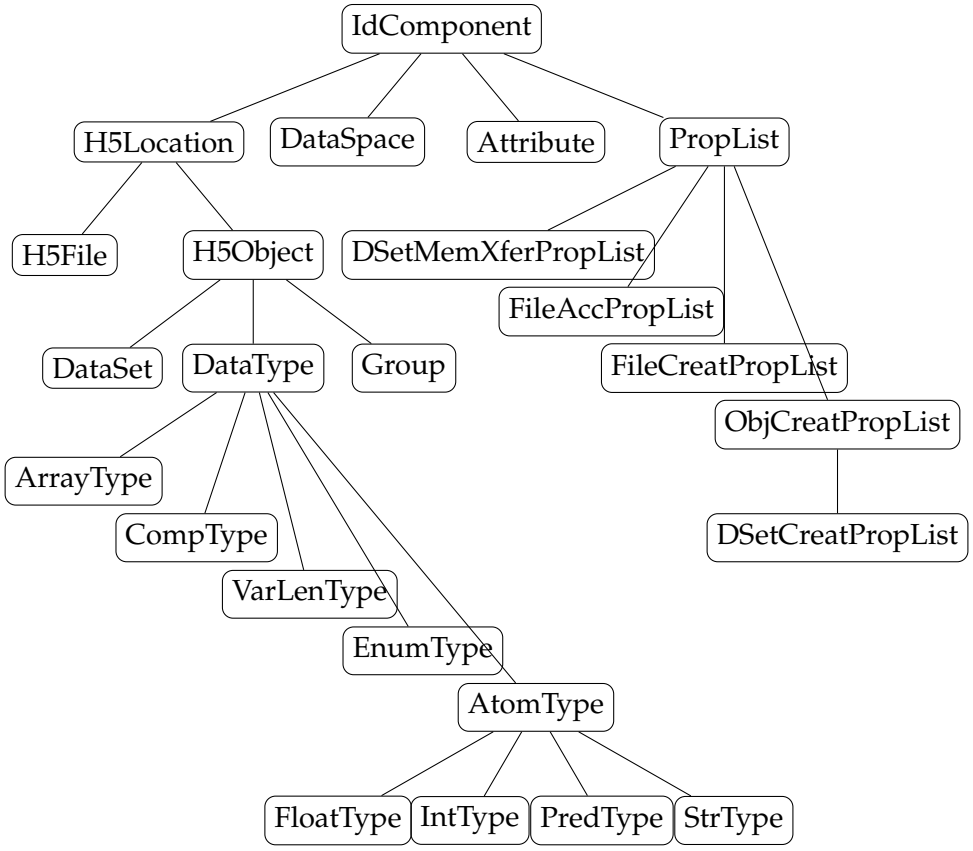

Figure 2.2: Depiction of derivation hierarchy

As stated in the section 1.2 this project works with *Hagedorn* wavepackets over a fixed time horizon. To make it possible that the simulation can be reproduced or restarted at any given time, the state has to be stored in every time step Δt. This can be achieved with the classes shown in figure 2.2, where their functionality will be explained in sections 2.2 to 2.9.

2.2 Internal types and states

An interface in general has supported objects and functions. These functions also have preconditions and postconditions whereas objects have valid states. In case of the HDF5 library when these conditions are not fulfilled, an exception is thrown, which could abort or interrupt the program execution. Therefore it is important to always use valid objects and function calls. There are two most commonly used types. These are `hsize_t` and `H5std_string`.

Variables of type `hsize_t` represent native integers and are always used to describe dimensions. This type substitutes the C++ internal `int` data type. The other is `H5std_string` type, which is just an alias for the `string` data type from the standard library. Furthermore internal valid states are always represented by names using only uppercase letters and with the C prefix for the class category as in table 2.1, separated with underscore. For example a valid property list state is `H5P_DEFAULT`.

2.3 H5File

As the name already suggests this class is used to manage the binary file object itself. When a *H5File* is default constructed it further allocates a default *Group* root named "/". To construct a *H5File* a minimal number of two arguments is needed, demonstrated in the code snippet below. The two additional optional arguments are a *FileCreatPropList* and a *FileAccPropList* which would allow further specification. These would be creation and access properties as suggested by the names themselves. In case of this project they are superfluous and the `H5P_DEFAULT` is sufficient. For the first mandatory argument, which is the filename, a `H5std_string` is required. The second one defines the type of creation, which has two valid values. These possible states are *H5F_ACC_TRUNC* and *H5F_ACC_EXCL*, which are mutually exclusive. The former truncates the file, meaning, if it already exists all contained data gets erased. The latter fails the construction, if the file already exists under the specified name and will result in an exception. It is advised to work with *H5F_ACC_TRUNC* because it is simpler and thus errors will less frequently occur but data is lost more often.

```
H5File file(string "filename", H5F_ACC_TRUNC);
```

2.4 Group

The hierarchy is realized with the usage of *Groups*. In case of a *Hagedorn* wavepackets and its corresponding observables the structure in figure 2.3 is preferred as it corresponds to the data written by the Python version.

The construction of a *Group* requires only one argument, namely the name as a `string`, but more importantly the name must include a path. As previously indicated in section 2.3 a *H5File* has root *Group* "/" by default after allocation. To accomplish the structure, as shown in figure 2.3, the path, beginning at "/", is added to the name. For instance to have a *Group* "Pi" below the *Group* "wavepacket" starting at "/" the name is modified to "/wavepacket/Pi". This implies that every intermediate node in figure 2.3 will be a *Group*. The leafs however will be *DataSets* which we will look

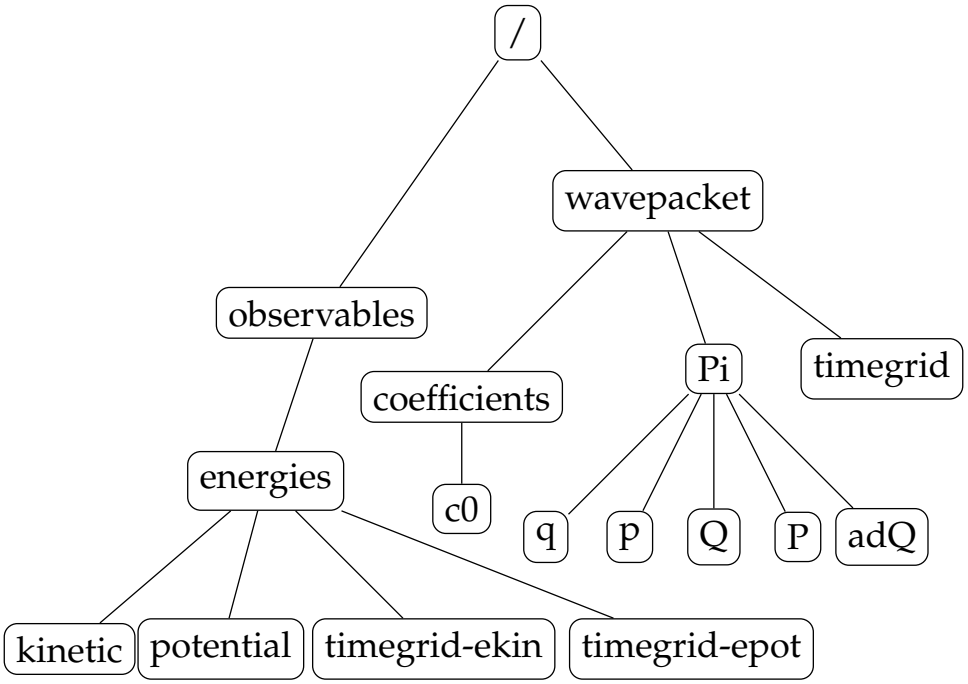

Figure 2.3: Depiction of desired internal structure of a H5File

into in the next section 2.5. The constructor gets invoked from the *H5File* object as the code below shows.

```
file.createGroup(string "/grouppath/groupname");
```

2.5 DataSet

To create a *DataSet* object four arguments are required with types *string*, *DataType*, *DataSpace* and *DSetCreatPropList* respectively. The first argument, analogous to the *Group*, is the name with its path included. E. g. the *DataSet* "ekin" has the complete name "/observables/energies/ekin". As the name indicates the second and third argument specify the type and space of the data. Lastly the forth argument defines the desired properties of the *DataSet*. This incorporates which data layout is chosen. The function code is shown below and again invoked by the *H5File*. The *DataSet* has three types of layouts to store raw data. These are `H5D_COMPACT`, `H5D_CONTIGUOUS` and `H5D_CHUNKED`. Figure 2.4 illustrates the inner workings of these layouts.

```
file.createDataSet(string "datasetpath/datasetname",DataType type
    ,DataSpace space, DSetCreatPropList list);
```

If the data is sufficiently small `H5D_COMPACT` will be used. In this layout the data address follows right after the header as shown on the left. When the data is larger but not growing, `H5D_CONTIGUOUS` in the middle, is the right

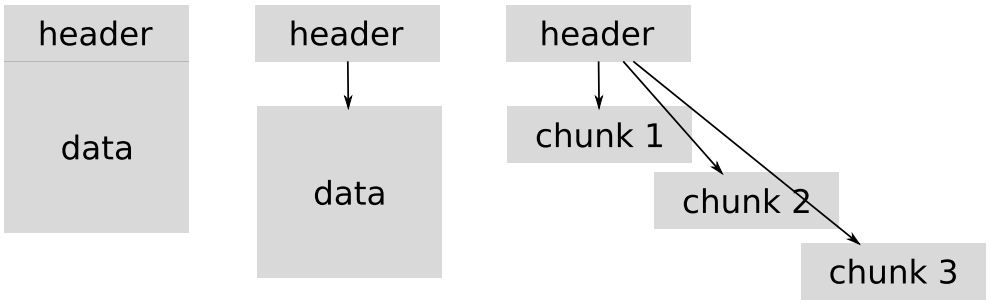

Figure 2.4: The three data layouts in *DataSet*

layout. This one allows the data to be located somewhere arbitrary in memory whereas a link is created into the header. The last layout, `H5D_CHUNKED` on the right, is useful if the size of the data is unknown. The data will be divided into chunks with the same size where the address of each chunk will be stored in the header. This implies when a chunk is full a new one can be easily allocated and linked in the header, which is ideal for this project. The goal is to store the time evolution of a matrix, a vector or simple values into one compact *DataSet*. This can be achieved when the first dimension is reserved for time itself. Therefore a matrix has three dimension where the first dimension is for the time index T_{index} and the others for row and column dimension respectively. Hence the chunk size in this case is defined by the matrix dimension and one chunk corresponds to the matrix belonging to a certain time index. The real time value can be calculated with the following formula:

$$T_{real} = T_{start} + T_{index} \cdot \Delta t \tag{2.1}$$

This will be used in chapter 4 to match data points between two *H5Files* according to their transformed timegrids.

2.6 DataType

The language C++ itself supports data types such as *int*, *double*, *char* etc. but these cannot be used directly in a binary data format. This is caused by the different data representations which are not homogeneous across different operating systems and system architectures. For example the difference between little-endian and big-endian machines. Thus the library has its own definitions which are compatible across platforms which are incorporated into the different classes as seen in figure 2.2. It is intuitively clear from their names which classes must be used to describe which data types. For instance for writing floating point numbers the *FloatType* class is used, where it also would be possible to change from IEEE representation to another one. In this project the *PredType* and *CompType* are the most relevant classes. From *PredType*, which is predetermined through the C++ language,

the *NATIVE_INT* type is utilized for writing timegrids. The *CompType* class is used for writing compositions of simple types such as a *struct* of two *doubles*. This is useful to construct a complex data type which will be explained in section 3.2.

2.7 DataSpace

To describe the shapes of our data the *DataSpace* class is required. The construction of such a *DataSpace* is straightforward. First of all the library needs to know the number of dimensions of the desired space. Second it has to know the number of elements along each dimension. For the second argument an array of `hsize_t` is expected instead of `int` as described in section 2.2. To give the reader an idea the following code prepares the *DataSpace* of a timegrid.

```
int rank = 1;
hsize_t size[rank];
size[0] = number_of_timesteps;
DataSpace limited_timespace(rank,size);
```

The problem herein lies in the number of time steps which is known only at runtime. This leads to another approach which is to define a unlimited *DataSpace*. The library allows this if an additional optional argument is added. This argument has to be of the same data type as the second argument and contains the maximum size of each dimension. The library expects that the values in this argument are greater or equal than in the previous argument otherwise an exception is thrown. For an unlimited space a special state is used namely `H5S_UNLIMITED`. The above example code changes to the following if an unlimited space is desired:

```
int rank = 1;
hsize_t size[rank] = {1};
hsize_t maxsize[rank] = {H5S_UNLIMITED};
DataSpace unlimited_timespace(rank, size, maxsize);
```

2.8 PropList

Most classes require a specific property list at the moment of construction which include additional information. Depending on the object a different property list is needed. All these lists have a common base class namely *PropList* as presented in figure 2.2. The possible applications of these property lists excel the scope of this project easily therefore the default value is enough. The default values are shortly described in table 2.5.

Noteworthy for this project is only the *DSetCreatePropList* mentioned in section 2.5 which will be explained next.

Default name value	Influence
H5P_FILE_CREATE	*H5FILE* creation
H5P_FILE_ACCESS	*H5FILE* access
H5P_DATASET_CREATE	*DataSet* creation
H5P_DATASET_XFER	raw data transfer
H5P_MOUNT	*H5File* mounting
H5P_DEFAULT	base value for all the above

Figure 2.5: Table of property list default values and their influence

2.8.1 DSetCreatePropList

From section 2.5 a *DSetCreatePropList* object is demanded for creating a *DataSet*. As the perceptive reader can guess this property list is used to determine the data layout shown in figure 2.4. The property list is responsible for setting the layout to H5D_CHUNKED. This is important because only data with this layout can be dynamically extended later on during a simulation.

```
//create default H5P_DATASET_CREATE
DSetCreatePropList list;
//set chunked data layout
list.setChunk(int number_of_dim, hsize_t* dim);
//set the fill value after allocation
list.setFillValue(DataType type, void* default_value);
```

The above code configures a chunked layout into the property list for the construction of a *DataSet*. Furthermore a default fill value can be defined which is utilized to fill newly allocated memory.

2.9 Attribute

Additionally to writing simulation data in binary format it should also incorporate saving the corresponding simulation configuration. This can be easily done with *Attributes* which must be attached to an affiliated *Group* or *DataSet*. The allocation of such an *Attribute* is similar to a *DataSet* meaning a *DataType* and *DataSpace* is also demanded. For its property list only the H5P_DEFAULT is allowed which the library enforces otherwise an exception is thrown. For illustration see the subsequent code.

```
Attribute attribute;
//group attribute
attribute = group.createAttribute(string "attributename",DataType
     type,DataSpace space,PropList plist);
//dataset attribute
attribute = dataset.createAttribute(string "attributename",
    DataType type,DataSpace space,PropList plist);
```

Chapter 3

Implementation

3.1 Link to Eigen library

As mentioned in the section 1.2 the simulation boils down to propagating the set $\Pi = \{q, p, \mathbf{Q}, \mathbf{P}, S\}$, where q and p are D dimensional real-valued vectors, $\mathbf{Q}$ and $\mathbf{P}$ hold complex $D \times D$ matrices and S is the global complex phase. A possible interface to work with these matrices and vectors is the *Eigen* library [15]. The simplified definition of an *Eigen* matrix has the following form:

```
Eigen::Matrix<complex<double>,row_dim,column_dim> mat;
```

Note that this is a class template over three parameters namely type, row-dimension and column-dimension. Therefore the overall implementation will also be a template. In the current version only scalar *Hagedorn* wavepackets are supported thus only one dimension parameter D is utilized but the framework can still easily be extended in the future. As discussed in section 2.6 native types are not writable, hence the type is defined through the library. This declared type is not necessarily the same as the template argument from *Eigen*, but is still similar enough to permit basic transformation functions which are discussed in section 3.8.

3.2 DataType Declaration

In this context simulation means manipulation of complex numbers. To build a *DataType* the library needs access to its members thus the standard `complex` class cannot be utilized. Therefore a `struct` is most suitable for defining the fundamental structure since all its members are by default public accessible. Hence our declaration of complex numbers looks like this:

```
struct ctype
{
```

```
  double real;
  double imag;
} instance_of_ctype;
```

This can now be used by the library to create the corresponding *DataType*. In this case the *CompType* is most suitable because `ctype` is a composition of simple data types. To be compatible with Python the same labels for its members have to be used which results in the following code:

```
CompType hdfctype_(sizeof(instance_of_ctype));
hdfctype_.insertMember("r",HOFFSET(ctype,real), PredType::
    NATIVE_DOUBLE);
hdfctype_.insertMember("i",HOFFSET(ctype,imag), PredType::
    NATIVE_DOUBLE);
```

Worth noting is that the constructor of *CompType* relies on the `sizeof` operator which tells how many bytes for an instance of this type are needed. The string labels ”r” and ”i” are utilized for the real respective the imaginary part of a complex number. `HOFFSET` is a simple macro which returns at which position counted in bytes the second argument is located inside the first argument. In this case the first `double`(real) is located at ”0” and the second double(imag) at ”8” since one `double` is 8 Bytes long. Finally the last argument is the type inserted at this position. As already discussed in section 2.6 `double` cannot be used directly and the library provides compatible definitions through the *PredType* class. The members of *PredType* are constant and are fixed through the C++ language itself.

3.3 Constructor

Since the data type declaration only has to be initialized once, it is suitable to pack it into the constructor of this writer template. The constructor only expects one string argument, namely the filename. Therefore the constructor can be written accordingly:

```
template<int D>
...
//constructor
hdf5writer(string name):filename_(name), hdfctype_(sizeof(
    instance_of_ctype)), file_(filename_,H5F_ACC_TRUNC)
{
        hdfctype_.insertMember("r", HOFFSET(ctype,real), PredType
            ::NATIVE_DOUBLE);
        hdfctype_.insertMember("i", HOFFSET(ctype,imag), PredType
            ::NATIVE_DOUBLE);
}
```

Observe that there are two implicit constructor calls after instantiation of the filename. For the former refer to section 3.2 whereas for the latter see section 2.3.

3.4 Write options

To enable customization about what and when something is written an additional layer was inserted. This was done with functions which set the desired configuration. In the current implementation there are three choices to make. By default writing *Hagedorn* wavepackets is enabled, whereas writing the corresponding observables are disabled. This behavior is directed by a boolean. As already mentioned in section 2.5 for each chunk, which is in essence a matrix or vector, also the time index of the current time step is written in an additional *DataSet*. These are shown in figure 2.3 where "timegrid" was used as its name. The corresponding customization thereof is to set the difference between two consecutive entries. More precisely, it is the choice if a *DataSet* should be written in each time step ($d = 1$) or every second time step ($d = 2$) etc. The supported functions are listed subsequently:

```
set_write_packet(true|false);
set_write_norm(true|false);
set_write_energies(true|false);
set_timestep_packet(1|2|3|...);
set_timestep_norm(1|2|3|...);
set_timestep_energies(1|2|3|...);
set_timestep_ekin(1|2|3|...);
set_timestep_epot(1|2|3|...);
```

3.5 DataSet paths

The structure and the names in figure 2.3 is fixed in the implementation as it is analogous to the hierarchy generated by the Python code. The data test is build on this structure as well. This can be problematic, once a change happens in the Python or C++ implementation, where the structure is affected. Luckily the library throws an *invalid path* exception if such a change occurs and the paths are no longer valid. Future work could focus on making data test path independent for *DataSets*.

3.6 Prestructure

This function is a bundle of steps which either have to be done prior to the writing process or are constant and therefore only need to be done once. The function definition is shown subsequently:

```
template<class MultiIndex>
prestructuring(ScalarHaWp<D,MultiIndex> packet, double dt);
```

The dimension D and the class `MultiIndex` are prerequisite for constructing a scalar *Hagedorn* wavepacket and are detailed explained in the thesis of M. Bösch [1]. A scalar *Hagedorn* wavepacket includes a matrix of coefficients and the set $\Pi = \{\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}, S\}$. The coefficient dimension is dependent on D and `MultiIndex`, therefore the number of coefficients is calculated and stored first in this function. Next the *Groups* of figure 2.3 are set up. Additionally some *Attributes* are attached to the root group such as the time step dt. As discussed in section 2.5 each *DataSet* has to be chunked according to a time step. Hence this chunk dimension is set in the next step for all *DataSets* specified in the write options. E.g. for a packet the chunk-dimension for the coefficients, $\underline{q}$, $\underline{p}$, $\mathbf{Q}$, $\mathbf{P}$ and S have to be set into an instance of *DSetCreatPropList* individually. For instance a chunk-dimension of a 2×2 matrix in time is declared in the following form:

```
hsize_t chunk[3] = {1,2,2};//{time_dim,row_dim,column_dim}
```

For further instruction see section 2.8.1. Before it is possible to allocate all *DataSets* according to figure 2.3 also the individual *DataSpaces* have to be declared. Luckily it is possible to reuse all chunk-arrays as arguments for their own *DataSpace* with the additional array argument according to section 2.7. Now all building blocks are ready to allocate in the next step all *DataSets*. It can be observed that there is a source and a destination *DataSpace* with a corresponding selection in each time step. The destination *DataSpace* grows over time within the file and thus is not constant. Different from the destination, the source *DataSpace* and its selection is constant and therefore can be fixed in this step for the whole simulation. Notice that *DataSpace* is the space of possible positions where data can be written to, whereas it needs to be specified which elements in *DataSpace* are utilized. This is done with a selection which is described by a hyperslab and will be explained in the next section.

3.7 Selection

As already mentioned in the last section a selection within a *DataSpace* is represented by a hyperslab. A hyperslab consists of four arrays namely:

```
hsize_t start[3] = {time_index,1,1};
hsize_t count[3] = {time_index,3,2};
hsize_t block[3] = {time_index,1,1};
hsize_t stride[3] = {time_index,2,2};
```

Notice that the first dimension is always reserved for the time dimension. The implementation has an internal index for storing the current simulation step which is used as the label. Thus only the index for the next time step has to be incremented without altering the code for the selection. The following figure 3.1 illustrates why four arrays are required to represent a hyperslab.

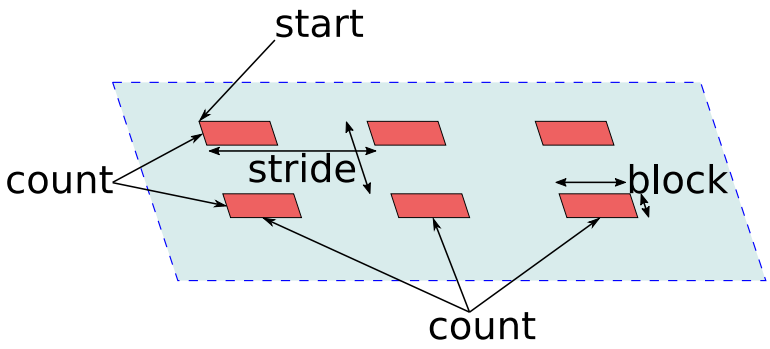

Figure 3.1: hyper-slab illustration

In figure 3.1 the blue dashed plane symbolizes a matrix in three dimensions. The first dimension is reserved for time meaning the blue dashed plane depicts a matrix at some point in time. One red block represents one single element in this matrix. Thus this figure displays the hyperslab from the previous code segment. The selection is done by the *DataSpace* itself whereas the `H5S_SELECT_SET` argument is recommended to overwrite the old selection.

```
dataspace.selectHyperslab(H5S_SELECT_SET, count, start, stride,
    block);
```

In case a contiguous block of data is written the last two arguments can be omitted.

```
dataspace.selectHyperslab(H5S_SELECT_SET, count, start);
```

3.8 Transformation

As indicated in section 3.1 basic transformations are applied to the *Eigen* matrices. The internal data of an *Eigen* matrix can be extracted with the `matrix.data()` function. Either the data is of type `complex<double>` or `double` which can be transferred to `ctype`. From `complex<double>` its members can be simply copied over whereas for a *double* the imaginary part is simply set to zero. After this step all data originally saved in an *Eigen* matrix are now stored in vectors of type `ctype`. Later on the data is again extracted with the `vector.data()` function and passed on to the HDF library for writing. The reason for this transformation from `complex<double>` or `double` to `ctype` is that in the actual writing function call `hdfctype_` will be used as type argument. Since `hdfctype_` is constructed from `ctype`, see section 3.3, it can be safely assumed that the library can write pointers of type `ctype`.

3.9 Writing

All tools are ready to look at the actual writing function call for *DataSets* and *Attributes*:

```
dataset.write(void* src, DataType type, DataSpace srcspace,
    DataSpace dstspace);
attribute.write(DataType type, void* src);
```

Notable is that the size of `src` is hidden in the selection of `srcspace`. Furthermore the destination is implicitly given by the `datasetname` where the function call occurred. When the selection of `srcspace` and `dstspace` does not match in size an exception is thrown. There are only two types used as *DataType*. For simulation data `hdfctype_` is utilized and for timegrids `PredType::NATIVE_INT` is used. Remember also that all `srcspaces` are constant and therefore constructed in section 3.6.

3.10 Extension

The *DataSets* in section 2.5 were explicitly constructed such that they have a chunked memory layout. This is useful now because the library can only extend this type of layout with the succeeding function call:

```
dataset.extend(hsize_t* dimension);
```

The new dimension is as expected an array of `hsize_t` where there are two effects. When an entry in the array is smaller than the *DataSet* itself, it will get shrunk along this direction. Alternatively if an entry is larger than the *DataSet* itself, it will get extended along this direction. There will be no effect in case the argument is of the same size as the *DataSet* itself. It is obvious that the *DataSets* in this project only grow in the direction of time per construction. Thereby the chunk dimension can be reused except for the first(time) entry. For this entry an index for each individual *DataSet* is utilized to keep track of the current size. Bear in mind that this index is not the same as the current time index which is used to write timegrids.

3.11 Update

Naturally since all *DataSets* were extended the corresponding *DataSpace* used as the destination is invalidated. Furthermore each individual index used to keep track of the number of chunks used in section 3.10 has to be incremented. Conveniently the HDF library provides a function to extract the *DataSpace* from a *DataSet* namely:

```
DataSpace newdataspace = dataset.getSpace();
```

Now all outdated *DataSpaces* are updated with the help of the extended *DataSets*. This function also represents the last action done in the current time step. All actions from section 3.7 to 3.11 are repeated in each time step of the simulation. Remember that the transformation is included because in each time step the *Hagedorn* wavepacket is propagated according to the chosen scheme and thus was modified.

3.12 Poststructure

After the time loop of the simulation has ended it is important to finalize all structures from the library. Furthermore the last extension has to be reversed since no further data will be written. This is easily done with the same function but with a smaller index as explained before in 3.11. All variables allocated on the stack are deleted automatically and thus can be neglected. The others allocated on the heap with the `new` operator need to be freed manually. Sensibly in this project all these pointer objects are forwarded to the standard library through the `shared_ptr` container. Likewise `vector` is managed also by the standard library and thus the standard library guarantees that these objects will be freed correctly.

3.13 Simulation in a picture

The functionality of this writer template can be easily visualized when only one *DataSet* is considered, see figure 3.2. The start symbolizes the construc-

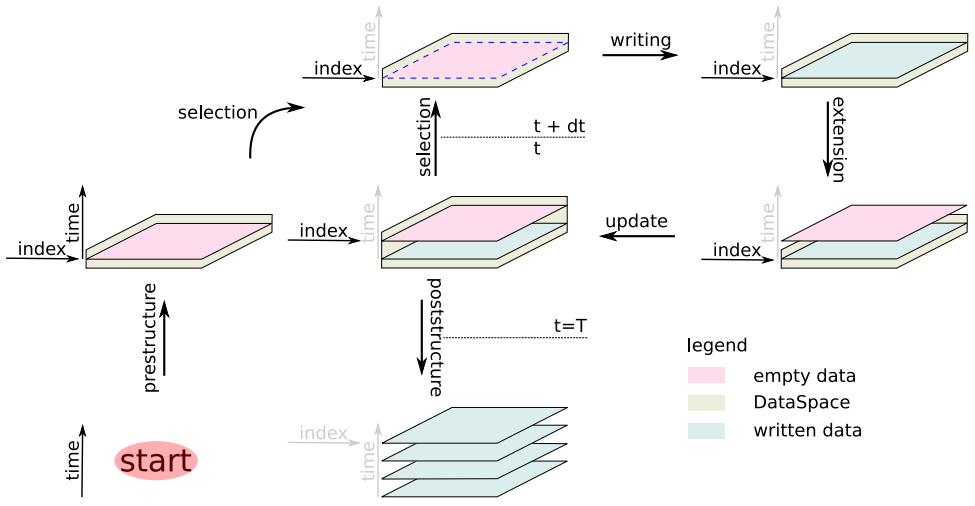

Figure 3.2: Illustration of inner workings for a *DataSet* in the simulation

tor of this class. The cycle begins with the selection process which sets a hyperslab, indicated by the blue dashed line. The sections 3.7 through 3.11

are part of this cycle which halts only when the time step arrived at T. Then the simulation gets finalized with the poststructure function as described in section 3.12.

3.14 Usage in Simulation

The following code snippet shows how to use the key functionality of this class for writing *Hagedorn* wavepackets:

```
//Simulation settings
...
io::hdf5writer<D> writer("simulation_filename.hdf5");

writer.prestructuring<MultiIndex>(packet, dt);

//Time loop(propagation)
for(real_t t = 0; t < T; t += dt)
{
        //Write wavepacket
        writer.store_norm(packet);

        //Propagate wavepacket
        propagator.propagate(packet, dt, V);
}

writer.poststructuring();
```

A more detailed version can be found in the appendix. The ellipsis there is fully expended and show how to generate the simulation setting for a harmonic $2D$ oscillator. To understand the simulation set up the reader can consult the preceding works online [1], [18] and [17] and the documentation therein. Furthermore additional information can be found in the documentation of the C++ implementation [4].

Chapter 4

Data Test

4.1 Introduction to GoogleTest

There are two main ways to test objects with *GoogleTest*. For the interested reader a more detailed description can be found in the git repository [8], where the *Primer.md* and *AdvancedGuide.md* examples are strongly suggested. Of importance is to define a test class for reusing certain objects for all tests. These objects in this case are the two *H5Files*, the *DataType* used in the writing process and the *Attributes* saved in the root group of the two files. One of these *Attributes* is the time step Δt utilized for the data matching. When there are two files in general they do not have necessarily the same time step size. However, mapped with the corresponding timegrid according to equation 2.1, some of the data map to the same timepoint, and therefore can be compared. This is carried out by a function at the beginning, which tests if two *DataSets* have matching timepoints, which will be stored in a list. If the list is empty a message will be printed and the program continues with the next *DataSet*.

4.2 The Main File

The main file consists of the test class for the shared objects, all test fissures which are based on the test class and the main function which invokes the test itself. The subsequent code snipped shows the core of the data test:

```
#include "gtest/gtest.h"
int global_argc;
char** global_argv;
#define abstol 1e-6
...

class TestHDF : public ::testing::Test
{
        protected:
```

```
        TestHDF();
        virtual ~TestHDF();
        void SetUp();
        void TearDown();
        void time_matching(...);

        struct ctype{...};
        H5File cppfile;
        H5File pyfile;
        CompType hdfctype_;
        double dt_cpp;
        double dt_py;
        ...
};

TEST_F(TestHDF, Testpacket)
{...}
TEST_F(TestHDF, Testenergies)
{...}
TEST_F(TestHDF, Testnorm)
{...}

int main(int argc,char* argv[])
{
        global_argc = argc;
        global_argv = argv;
        ::testing::InitGoogleTest(&argc, argv);
        return RUN_ALL_TESTS();
}
```

First the *TestHDF* class gets constructed where we use global `argc` and `argv` arguments to transfer the filenames from the console to the class. Notice also that *TestHDF* needs to derive from `::testing::Test` from *GoogleTest* to work. Then in the main function the test fissures are invoked with the `RUN_ALL_TESTS()` call. For a test fissure `TEST_F()` an additional `SetUp()` function is provided for further specifying test settings. Afterwards each test fissure also gets cleaned up with the `TearDown()` function. In the test fissure itself the data test takes place. First the time matching function is invoked with the timegrids of both files as arguments. If the prepared list is not empty then the correct values are loaded from the *H5Files*. These values are then compared with the `EXPECT_NEAR(...)` function provided by the test framework which compares the difference of two *doubles* to an absolute tolerance. This process will be repeated for each matching *DataSet* in these two files until non-zero return code in `RUN_ALL_TESTS()` finish the execution. If the difference between two *doubles* is larger than the defined absolute tolerance they will be printed out automatically and the test will proceed.

Chapter 5

Conclusion

This project successfully implemented a more sophisticated way to serialize *Hagedorn* wavepackets using the HDF5 C++ interface. Hereby the dependence on an external project [10] was removed. This implementation handles *Eigen* matrices by transforming the actual data, where complicated type derivation was also eliminated. This was achieved by adapting the type and hierarchy of the data for all simulations to the structure used by the Python code. This further allowed us to implement a data test, which based on type and hierarchy, can compare C++ and Python generated data. Though a user of the software [4] has to be now aware that additionally to a compiler, which supports C++11 features, also *Eigen*, *HDF5* and *GoogleTest* have to be installed. Hopefully this does not prevent some users to use [4] since extra effort is required to install those libraries.

Appendix A

Appendix

A.1 Code of 2D harmonic oscillator

```
//Standard libraries
#include <iostream>
#include <fstream>
//This framework
#include "waveblocks/types.hpp"
#include "waveblocks/wavepackets/shapes/tiny_multi_index.hpp"
#include "waveblocks/potentials/potentials.hpp"
#include "waveblocks/potentials/bases.hpp"
#include "waveblocks/wavepackets/hawp_paramset.hpp"
#include "waveblocks/wavepackets/hawp_commons.hpp"
#include "waveblocks/wavepackets/shapes/shape_enumerator.hpp"
#include "waveblocks/wavepackets/shapes/shape_hypercubic.hpp"
#include "waveblocks/innerproducts/gauss_hermite_qr.hpp"
#include "waveblocks/innerproducts/tensor_product_qr.hpp"
#include "waveblocks/propagators/Hagedorn.hpp"
#include "waveblocks/observables/energy.hpp"
#include "waveblocks/io/hdf5writer.hpp"
//namespace for convenience
using namespace waveblocks;
//Main function
int main()
{
//Simulation settings
const int N = 1;
const int D = 2;
const int K = 4;

const real_t sigma_x = 0.5;
const real_t sigma_y = 0.5;

const real_t tol = 1e-14;

const real_t T = 12;
const real_t dt = 0.01;
```

```
const real_t eps = 0.1;

using MultiIndex = wavepackets::shapes::TinyMultiIndex<unsigned
    long, D>;

//The parameter set of the initial wavepacket
CMatrix<D,D> Q = CMatrix<D,D>::Identity();
CMatrix<D,D> P = complex_t(0,1) * CMatrix<D,D>::Identity();
RVector<D> q = {-3.0, 0.0};
RVector<D> p = { 0.0, 0.5};
complex_t S = 0.0;
wavepackets::HaWpParamSet<D> param_set(q,p,Q,P,S);

//Basis shape
wavepackets::shapes::ShapeEnumerator<D, MultiIndex> enumerator;
wavepackets::shapes::ShapeEnum<D, MultiIndex> shape_enum =
    enumerator.generate(wavepackets::shapes::HyperCubicShape<D>(K
    ));

//Gaussian Wavepacket phi_00 with c_00 = 1
Coefficients coeffs = Coefficients::Zero(std::pow(K, D), 1);
coeffs[0] = 1.0;
Coefficients coefforig = Coefficients(coeffs);

//Assemble packet
wavepackets::ScalarHaWp<D,MultiIndex> packet;
packet.eps() = eps;
packet.parameters() = param_set;
packet.shape() = std::make_shared<wavepackets::shapes::ShapeEnum<
    D,MultiIndex>>(shape_enum);
packet.coefficients() = coeffs;

//Defining the potential
typename CanonicalBasis<N,D>::potential_type potential = [sigma_x
    ,sigma_y](CVector<D> x)
        {
    return 0.5*(sigma_x*x[0]*x[0] + sigma_y*x[1]*x[1]).real();
        };

typename ScalarLeadingLevel<D>::potential_type leading_level =
    potential;
typename ScalarLeadingLevel<D>::jacobian_type leading_jac = [
    sigma_x,sigma_y](CVector<D> x)
    {
    return CVector<D>{sigma_x*x[0], sigma_y*x[1]};
    };
typename ScalarLeadingLevel<D>::hessian_type leading_hess = [
    sigma_x,sigma_y](CVector<D> /*x*/)
    {
    CMatrix<D,D> res;
    res(0,0) = sigma_x;
    res(1,1) = sigma_y;
    return res;
```

```
    };

ScalarMatrixPotential<D> V(potential,leading_level,leading_jac,
    leading_hess);

//Quadrature rules
using TQR = innerproducts::TensorProductQR<innerproducts::
    GaussHermiteQR<K+4>,innerproducts::GaussHermiteQR<K+4>>;

//Defining the propagator
propagators::Hagedorn<N,D,MultiIndex, TQR> propagator;

io::hdf5writer<D> writer("harmonic_2D_cpp.hdf5");
writer.set_write_energies(true);
writer.set_write_norm(true);
writer.prestructuring<MultiIndex>(packet,dt);

//Time loop(propagation)
for (real_t t = 0; t < T; t += dt)
        {
    real_t ekin = observables::kinetic_energy<D,MultiIndex>(
        packet);
    real_t epot = observables::potential_energy<
        ScalarMatrixPotential<D>,D,MultiIndex,TQR>(packet,V);

        writer.store_packet(packet);
    writer.store_energies(epot,ekin);
    writer.store_norm(packet);

    //Propagate
    propagator.propagate(packet,dt,V);

    //Assure constant coefficients
    auto diff = (packet.coefficients() - coefforig).array().abs()
        ;
    auto norm = diff.matrix().template lpNorm<Eigen::Infinity>();
    bool flag = norm > tol ? false : true;
    std::cout<<flag<<'\n';
    }
writer.poststructuring();
}
```

Bibliography

[1] Michaja Bösch. Efficient implementation of Hagedorn wavepackets in C++. 2015.

[2] R. Bourquin. Algorithms for non-adiabatic transitions with one-dimensional wavepackets. 2010. http://www.sam.math.ethz.ch/~raoulb/research/bachelor_thesis/tex/main.pdf.

[3] 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.

[4] R. Bourquin, M. Bösch, L. Miserez, and B. Vartok. libwaveblocks: C++ library for simulations with semiclassical wavepackets. https://github.com/WaveBlocks/libwaveblocks, 2015, 2016.

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

[6] R. Bourquin, V. Gradinaru, and G.A. Hagedorn. Non-adiabatic transitions near avoided crossings: theory and numerics. *Journal of Mathematical Chemistry*, 50:602–619, 2012.

[7] Billy Donahue. GoogleTest. https://github.com/google/googletest, 2008.

[8] Billy Donahue. GoogleTest Documentation. https://github.com/google/googletest/tree/master/googletest/docs, 2008.

[9] Erwan Faou, Vasile Gradinaru, and Christian Lubich. Computing semiclassical quantum dynamics with Hagedorn wavepackets. *SIAM Journal on Scientific Computing*, 31(4):3027–3041, 2009.

[10] James R. Garrison. eigen3-hdf5. https://github.com/garrison/eigen3-hdf5, 2013.

[11] Vasile Gradinaru and George A. Hagedorn. Convergence of a semiclassical wavepacket based time-splitting for the Schrödinger equation. *Numerische Mathematik*, 126(1):53–73, 2014.

[12] Vasile Gradinaru, George A. Hagedorn, and Alain Joye. Tunneling dynamics and spawning with adaptive semiclassical wave packets. *Journal of Chemical Physics*, 132, 2010.

[13] HDF Group. HDF5 C++ Documentation. https://www.hdfgroup.org/HDF5/doc/cpplus_RM/index.html, 2001.

[14] HDF Group. HDF5 C Documentation. https://www.hdfgroup.org/HDF5/doc1.6/RM_H5Front.html, 2001.

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

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

[17] Lionel Miserez. Porting WaveBlocksND Matrix Potential Functionality to C++. 2015.

[18] Benedek Vartok. Implementation of WaveBlocksND's Quadrature and Inner Products in C++. 2015.