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. Vasilie Grădinaru
Seminar of Applied Mathematics, ETH Zürich

Abstract

It is of general interest in physics/chemistry to find viable algorithms to solve the time dependent Schrödinger equation. A means to an end are *Hagedorn* wavepackets for a semi-classical approach. With this as basis a python implementation was created. This implementation can be used to simulate different kind of starting conditions which can result in a lot of data. To efficiently store this data it is desirable to use binary formats, which can be easily compressed if needed. A well-known data binary format is HDF which stands for hierarchical data format. To further improve simulation speed the python implementation was used as a basis for a C++ implementation. This thesis explains the intrinsic functionality of using the HDF5 library interface in C++. This was done in such a way that the data from the C++ implementation has the same hierarchy as the python implementation. Also for further use a test was implemented with the well-known GoogleTest framework to easily compare possible data between these two implementations.

Contents

Chapter 1

Introduction

1.1 Motivation

To test an algorithm or approach such as semiclassical wavepackets [2] for solving the time dependent Schrödinger equation a python implementation [5] is at most times sufficient. In case longer simulation times and/or solving higher dimensional problems is demanded the computation time becomes an important factor. As such there is a need for a time efficient implementation which is only doable in languages such as C or C++. A starting implementation in C++ for scalar wavepackets is already done [4] but further improvement is desired. One aspect which has to be taken into account is the storage format of the data but also the speed of these io-operations. As storage format the well-know HDF(hierarchical data format) is most suitable for our case given that it is a binary file format with sufficient io speed. The current form to store data is dependent on a external source [9], which is sufficient for small projects where data is supervised by the user, which uses HDF5 format but doesn't fully use its capabilities. Furthermore the chosen hierarchy is also not compatible with the dual python implementation. In this project a fully compatible hierarchy is implemented in C++ without the usage of an external source. Also for testing simulations under both implementations a data test was implemented using the well-know Google testing framework.

1.2 Background

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

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

where H is the Hamiltonian, $\varphi(x,t)$ represents the wave function dependent on position x and time t and $\langle\varphi|\varphi\rangle$ is the probability density of electrons. This equation can be reformulated in a semiclassical setting for nuclei as:

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

Nevertheless there are still much challenges involved to solve this equation 1.2. One of this is the high dimensionality of this equation. A molecule with N nuclei where each of them has three degrees of freedom results in $3N$ unknowns. 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 by a Fourier based approach. Further there is the problem of long time evolutions.

In this semiclassical setting *Hagedorn* wavepackets with its operators as described in [14] is a viable tool to overcome these challenges. Not only will it be gridfree but also a spectral based method which results in fast approximations in space of localized wave-functions. Lastly it further allows highly oscillating functions.

A basic algorithm of computing quantum mechanics with *Hagedorn* is specified in [8]. In this algorithm splitting method is used to divide the problem into classical propagation of {q,p,Q,P,S} with quantum correction terms. Further improvement was done in [10].

A diverse implementation was done in python [3] where source code is available on github [5]. This implementation further supports various propagators. Interesting applications of this code can be found in the project about tunneling dynamics [11] and non-adiabatic transitions [6].

As python is an interpreted language it is mostly not optimized for execution time. To circumvent this problem a C++ implementation is desired. Until now only the base functionality was transferred to C++. Hagedorn wavepackets was done in [1], inner products in [16] and the potentials in [15]. The current base implementation is available on github [4]. In this project the goal was to enlarge the C++ code functionality with intelligent data serialization. This was done with the well-known HDF5 library to reach compatibility with the structure of the data of the two implementation. This further allowed to use the GoogleTest framework to write a test which en-

abled to check if two simulations with different implementations yield the same data.

Chapter 2

HDF5 C++ Interface

2.1 Overview

The acronym HDF stands for hierarchical data format meaning this binary format is allowing to structure the internal objects after the users demand. This structure is quite similar to a file system where data is ordered with folders and sub-folders. For more detailed information see the official C [12] 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 dependence see table 2.1. To use the C++ language features to its most capabilities

HDF5 C APIs	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++

inheritance is used. This allows reuse of functions, objects and properties over many hierarchy layers. This enforces a strict dependence when sharing or creating objects. To have an overview on the hierarchy look at figure 2.2.

As stated in the background this project is about the simulation of *Hagedorn* wavepackets over a fixed time horizon. To keep track and possibly repro-

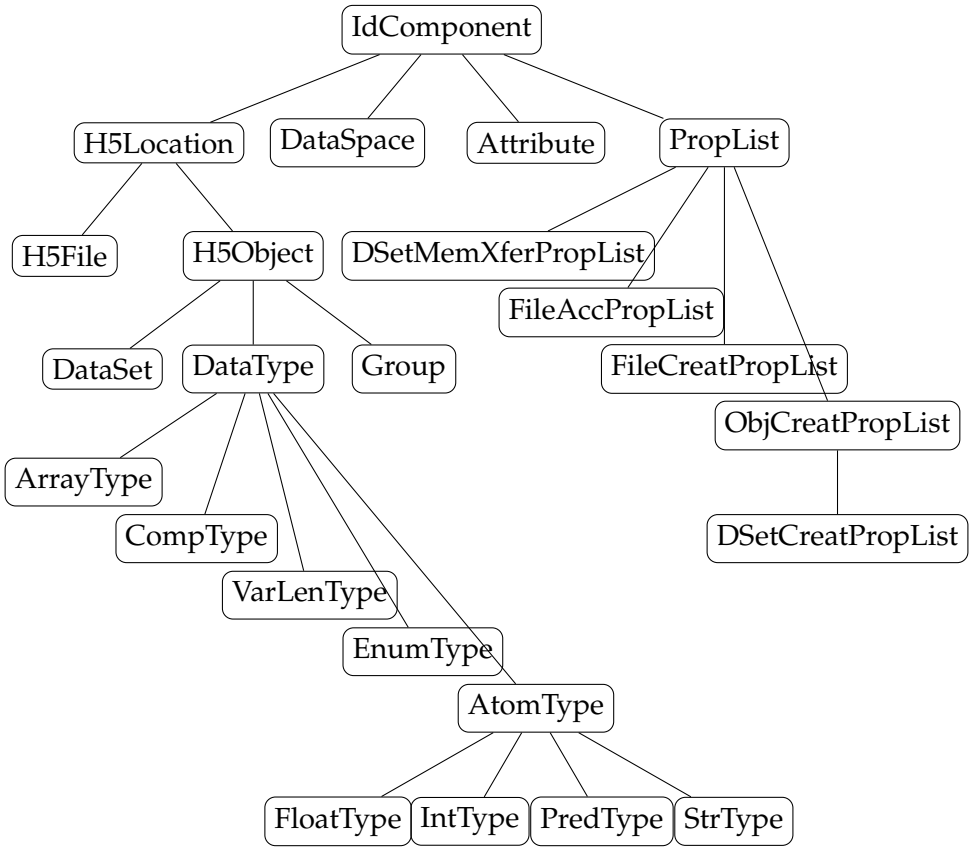

Figure 2.2: Depiction of derivation hierarchy

duce the results the wavepackets have to be saved in every time step Δt of the simulation. This has to be achieved with the classes shown in figure 2.2. To understand the functionality of these classes they will be explained in the following sections.

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 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 noteworthy types which are also used the most. These are `hsize_t` and `H5std_string`. Variable of type `hsize_t` represent native multiple-precision integer. This type substitutes the C++ internal `int` data type. The `H5std_string` type is just an alias for the `std::string` data type from the standard library. Internal valid states are represented in only uppercase letters and with the C

prefix as in table 2.1. For example a valid property list state is `H5P_DEFAULT`.

2.3 H5File

As the name already suggest this class is used to manage the binary file object. When a *H5File* is default constructed it also allocates a default *Group* root named "/". To construct a *H5File* a minimal number of two arguments is needed. 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. In case of this project they are not needed and the `H5P_DEFAULT` is sufficient. For the first mandatory argument, which is the filename, a `H5std_string` or as discussed before a string required. The second mandatory argument defines the type of creation which at default has two valid values. These possible values are *H5F_ACC_TRUNC* and *H5F_ACC_EXCL* which are mutually exclusive. The former truncates the file meaning if it already exists erase all data previously stored in the file. The latter fails if the file already exists under the specified name which ends in an exception. It is advised to work with *H5F_ACC_TRUNC* because it is simpler and thus errors will less frequently occur.

2.4 Group

As previously mentioned it is a hierarchical data format. The hierarchy gets implement through the usage of *Groups*. In case of a *Hagedorn* wavepackets and its corresponding energies the structure in figure 2.3 is preferred as it corresponds to the structure of the python generated data.

To create a *Group* only a `string` argument is required which acts as name but more importantly also as path similar to figure 2.3. As previously indicated a *H5File* has root *Group* "/" by default after allocation. To accomplish the structure as in figure 2.3 the path is included in the name. For instance to have a *Group* "wavepacket" after the root "/" and a *Group* "Pi" after "wavepacket" the first name will be modified to "/wavepacket" and the second to "/wavepacket/Pi". This implies that every intermediate node in figure 2.3 will be a *Group*. The leafs however will be *DataSets* which will be explained in the next section 2.5.

2.5 DataSet

The constructor of the *DataSet* class demands four arguments with individual type *string*, *DataType*, *DataSpace* and *DSetCreatPropList*. The first argument analogous to *Group* is the name with its path included. For example

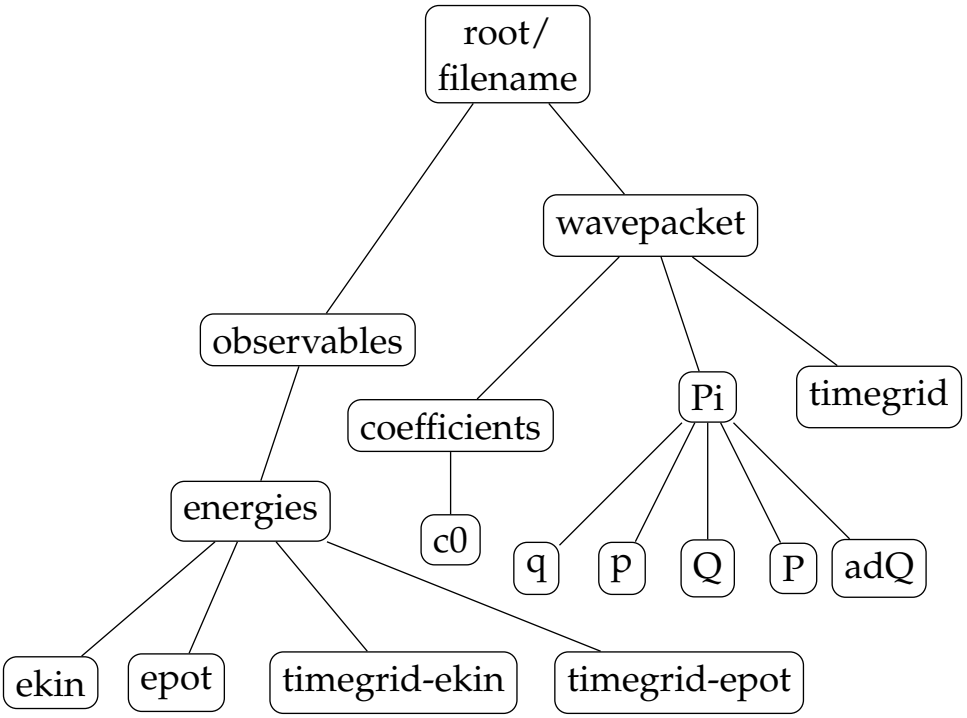

Figure 2.3: Depiction of desired internal structure of a H5File

the *DataSet* "ekin" has the complete name "/observables/energies/ekin". As the name indicates the second and the third argument specifies the type and space of the data. Lastly the forth argument defines the properties at creation. This incorporates which data layout is chosen. The *DataSet* has three types of layouts to store raw data. These are `H5D_COMPACT`, `H5D_CONTIGUOUS` and `H5D_CHUNKED`. Figure 2.4 should illustrate the inner workings of these layouts.

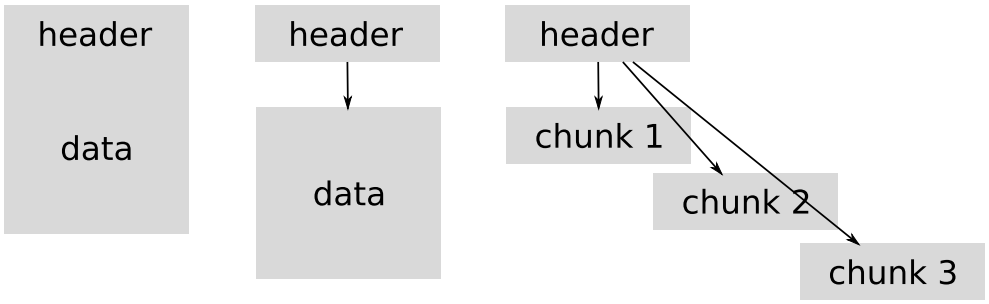

Figure 2.4: The three data layouts in *DataSet*

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 bigger but still has a constant size `H5D_CONTIGUOUS` is the right layout. This one allows that the address of the data can be arbitrary in memory and is saved in the header. The last layout `H5D_CHUNKED` on the right is useful assuming the size of the data is unknown. The data will be divided continuously into chunks with constant size where the address will be saved in the header. This means when a chunk is full a new one gets allocated with its address saved into the header. As the perceptive reader can guess the last layout is of most relevance for this project. A simulation generates new data in each time step which in essence is ideal for the size of a chunk because it further allows to label each chunk to its corresponding time step. This will be later used to match data according to its time label.

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 different data representation 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 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. In this object further changes of characteristics can be done such as changing from default IEEE representation. In this project the *PredType* and *CompType* are of most relevance. The *PredType* is predetermined meaning there is no explicit constructor needed. From this class *NATIVE_INT* is used to write the respective time labels for the data. The *CompType* class is utilized for writing complex numbers and also this class has to be explicit constructed which will be explained in section 3.2.

2.7 DataSpace

To describe the dimensionality of our data the *DataSpace* class is required. The construction of such a *DataSpace* is straightforward. Firstly the library needs to know the number of dimensions of the desired space. Secondly it has know the number of elements in 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 depicts the *DataSpace* of a time grid.

```
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 but the constructor expects it at compiletime. 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 this argument has to be greater or equal than 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 required. All these lists have a common base class namely *PropList* as presented in figure 2.2. The possibilities 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.

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

Noteworthy for this project is only the *DSetCreatePropList* mentioned in section 2.5 which will be explained in the next section 2.8.1.

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 responsi-

ble for setting the layout to H5D_COMPACT. This is important because only data with this layout can be dynamically extended later on during a simulation.

2.9 Attribute

Additionally to writing data in binary format it should also incorporate saving corresponding meta data. 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.

Chapter 3

Writer Template

3.1 Link to Eigen library

As mentioned in background 1.2 the simulation boils down to propagating the set {q,p,Q,P,S} where q and p are D dimensional real-valued vectors, Q and P complex DxD matrices and S the global complex phase. A possible interface to manage these matrices and vectors is the Eigen library. The definition of an Eigen matrix has the following form:

```
Eigen::Matrix<std::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 further works. As discussed in section 2.6 normal 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.

3.2 DataType Declaration

In this context simulation means manipulating 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 the 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* which 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 has to be used which result in the following code:

```
CompType nctp_(sizeof(instance_of_ctype));
nctp_.insertMember("r",HOFFSET(ctype,real),
    PredType::NATIVE_DOUBLE);
nctp_.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 is needed. The string labels ”r” and ”i” are utilized for the real respective the imaginary part of a complex number. `HOFFSET` is a simple function which returns at which position counted in bytes the second argument is located in the first argument. In this simple case because one double is saved in 8 Bytes `real` is at position ”0” and `imag` is at position ”8”. Finally the last argument is the type inserted at this position. Since already discussed in section 2.6 `double` cannot be used directly but the library provides these 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 has only 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 the constructor can be written accordingly:

```
template<int D>
...
//ctor
hdf5writer(std::string name):filename_(name),
    nctp_(sizeof(instanceof)),file_(filename_,H5F_ACC_TRUNC)
{
        nctp_.insertMember("r",HOFFSET(ctype,real),
            PredType::NATIVE_DOUBLE);
        nctp_.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 for the latter to 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 and writing norm and energies are disabled with a boolean. As already mentioned in section 2.5 for each chunk written also a time label is attached. This was achieved by writing for each important *DataSet* an additional *DataSet* at the same level with the same length where each entry is an `int` symbolizing the number of time steps passed since the beginning of the simulation. These are shown in figure 2.3 where "timegrid" was used in 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 in figure 2.3 is fixed in the implementation as it is analogous to the python generated data. Furthermore the names of all *Groups* and *DataSets* are also already determined. The data test is build on these 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 where the paths are no longer valid. Future work could include to make data test path independent for *DataSets*.

3.6 Prestructure

This function is a bundle of steps which either have to be done prior or are constant during the writing process. 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 is detailed explained in the thesis of Michaja Bösch [1]. A scalar *Hagedorn* wavepacket includes a matrix of coefficients and the set {q,p,Q,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 is set. 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, q, p, Q, P, and S have to be set into a instance of *DSetCreatPropList* individually. For instance a chunk-dimension is declared in the following form:

```
hsize_t chunk[3] = {1,2,2};
```

Before it is possible to allocate all *DataSets* according to figure 2.3 also 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 the *DataSets*. It can be observed that there is a source and destination *DataSpace* with their respective 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 for the whole simulation in this step.

3.7 Selection

The *DataSpace* from the data source is a constant sized chunk defined through the time step. In contrast the destination space and its hyper-slab selection is changing every set time step size since it gets extended dynamically after every write. The next figure should demonstrate how this selection process work for two dimensional matrices.

The first entry is always reserved for the time point. The next two dimensions is used as the basis for the matrix. The corresponding code to figure 3.1 could look like this:

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

Thus in every time step the hyper-slab has to be reselected if we want to write this data. This is done internally with an `time_index` which indicates in which time step the simulation is at the moment. The dimension written is always the chunk dimension by construction as such this selection is

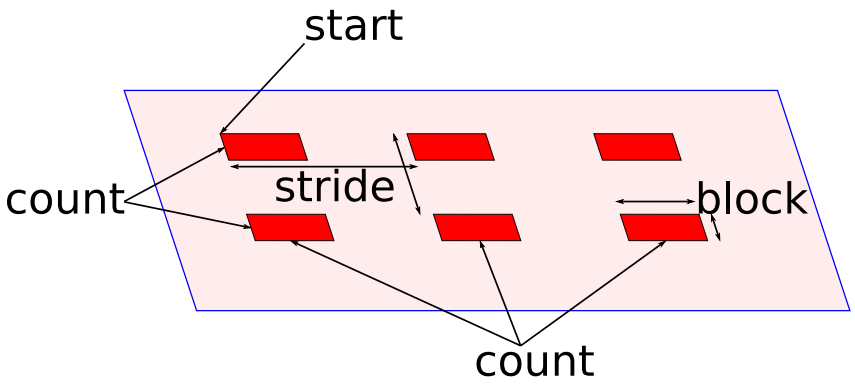

Figure 3.1: hyper-slab illustration

constant over time and such can be set at the beginning as described in the previous section. The selection process is one function statement used on a corresponding *DataSpace*:

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

3.8 Transformation

As previously discussed to write data we use our own created data type *nctp*. To write a buffer we need now a pointer of `ctype`. As such our transformation functions take an `Eigen::Matrix` with arbitrary template arguments and copies the raw data into a `std::vector<ctype>` reference. An important implementation detail for data management is the usage of `std::vector` because the operation system assumes the allocation and deallocation of `std` objects. Thus our code is much safer in respect to data leaks and data corruption. The *Eigen* library self-manages its matrix objects. To extract `ctype` pointer from the `std::vector` object is simple by using the `.data()` function call.

3.9 Writing

The writing function call from the HDF library has the following form:

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

We already have all tools available to use this function. The `src` is delivered from the transformation section, `type` from the declaration section and `srcspace`, `dstspace` from the prestucturing sectoin or selection section respectively.

3.10 Extension

For extending a *DataSet* we need the new dimension in a `hsize_t` array which is used as the extension. Note here that only chunked *DataSets* are possible to extend with the following call:

```
datasetname.extend(hsize_t extension);
```

Also note this function call can also be used to reduce the dimension if the used array `extension` is smaller then the existing *DataSet*. To know the current position of the data an `int index` is used for every existing *DataSet*. This index is also used in setting the dimension of the `hsize_t extension` array.

3.11 Update

After the extension of a *DataSet* the library doesn't automatically also update the corresponding *DataSpace* which was used in the selection section. As such this is done in this update section. Also the corresponding `index` to the *DataSet* is incremented in this step. The library provides an easy way to use the extended *DataSet* to set the new *DataSpace* namely:

```
dataspacename = datasetname.getSpace();
```

which is done for every existing *DataSet* in the file.

3.12 Poststructure

After the last time step of the simulation we have to finalize our *DataSet* and structures used. This is done in this section. From our implementation we know that also in the last time step we did extend the *DataSet*. As such the last extension will be reversed. Afterwards every structure from the library has to be freed from memory. The system does this automatically for all structures allocated on the stack. For structures which are on the heap because they were allocated with the `new` operator we have to free them ourselves. This step can be cumbersome because mostly we don't know anymore which object has internally somewhere a reference to it and therefore it is unsafe data management. The implementation avoids this problem by using the `new` operator to allocate `std::shared_ptr` objects which will be managed by the standard library the same as the `std::vector`. As such the data management is primarily done by the system and not by the implementation.

3.13 Inner workings in a picture

The following figure 3.2 shows the lifetime of a *DataSet* in this implementation. The start signalizes the use of the constructor to allocate the object.

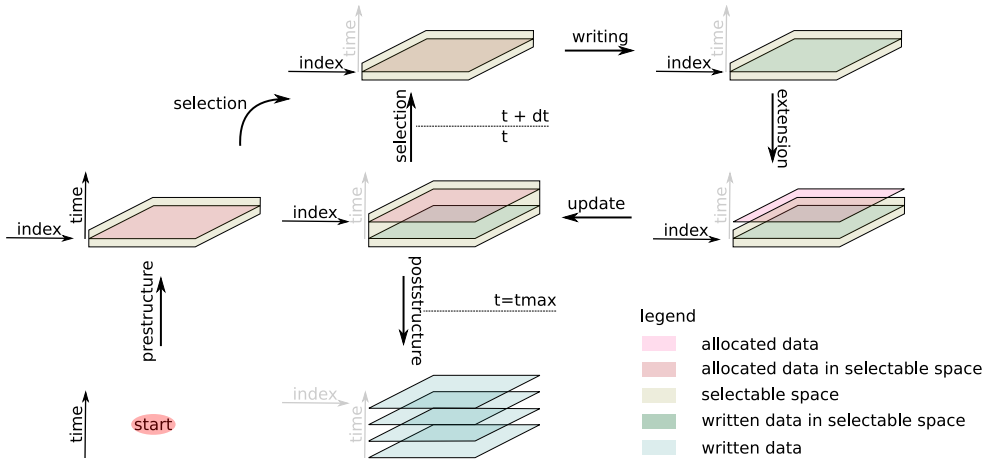

Figure 3.2: Illustration of inner workings for a *DataSet* in the simulation

3.14 Usage in a simulation main file

Note that this implementation is only able to write scalar *Hagedorn* wave packets with the normal *Hagedorn* parameter set for the moment. Further work includes to either extend this HDF5 writer template with matching functions for more complicated *Hagedorn* wave packets or to generalize the template on these packets. In the simulation the packet and its parameter set is constructed in the following form:

```
...
wavepackets::HaWpParamSet<D> param_set(q,p,Q,P,S);
...
wavepackets::ScalarHaWp<D,MultiIndex> packet;
...
```

Hence our HDF5 writer template is constructed and used in the following way:

```
... //setting simulation variables, options etc.

//constructor of this template implementation
io::hdf5writer<D> writer("simulation_filename.hdf5");

//optional setup write options
bool write_packet,write_energy,write_norm;
```

```
write_packet=true;//default value
write_energy=false;//default value
write_norm=false;//default value
writer.set_write_energy(write_packet);
writer.set_write_norm(write_energy);
writer.set_write_packet(write_norm);
int packet_stepsize,energy_stepsize,norm_stepsize;
packet_stepsize=1;//default value writing every time step
energy_stepsize=1;//default value writing every time step
norm_stepsize=1;//default value writing every time step
writer.set_timestep_packet(packet_stepsize)
writer.set_timestep_energy(energy_stepsize)
writer.set_timestep_norm(norm_stepsize)

//optional setup write paths
std::string datablockstring,wavepacktstring,packetgroupstring;
std::string coefficientgroupstring,energiesgroupstring;
std::string normsgroupstring,observablesgroupstring;
//default paths see figure 2.3
datablockstring="/datablock_0";
wavepacktstring="/wavepacket";
packetgroupstring="/Pi";
coefficientgroupstring="/coefficients"
energiesgroupstring="/energies"
normsgroupstring="/norm";
observablesgroupstring="/observables";
writer.set_datablockstring(datablockstring);
writer.set_wavepacketstring(wavepacktstring);
writer.set_packetgroupstring(packetgroupstring);
writer.set_coefficientgroupstring(coefficientgroupstring);
writer.set_energiesgroupstring(energiesgroupstring);
writer.set_normsgroupstring(normsgroupstring);
writer.set_observablesgroupstring(observablesgroupstring);

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

//simulation loop body
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,MultiInd
        writer.store_packet(packet); //function call for storing
            packet in file
        writer.store_energies(epot,ekin); //function call for
            storing energies in file
        writer.store_norm(packet); //function call for storing
            norm in file
}

//finalization with postructure
writer.poststructuring();
```

Remember that only the constructor, prestructuring and poststructuring are always mandatory independent of the simulation. Also note that if write option of norm and/or energies is set to true but the `.store_...` function call is missing that an error message will be printed without aborting the simulation.

Chapter 4

Data Test

4.1 Introduction to GoogleTest

There are two main ways to test objects with this framework. For the interested reader a more detailed documentation can be found in the git repository [7] 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 used time step Δt in the simulation which will be needed later for a data matching.

4.2 The Main C++ File

The data test main file is of the following structure:

```
#include "gtest/gtest.h"
int global_argc;
char** global_argv;
...

class TestHDF : public ::testing::Test
{
        protected:
        TestHDF();
        virtual ~TestHDF();
        void SetUp();
        void TearDown();
        void time_matching(...);

        struct ctype{...};
        H5File cppfile;
        H5File pyfile;
        CompType nctp;
        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();
}
```

Note that `TEST_F` are test fissures which uses the same class object `TestHDF`. In the constructor the filenames are loaded in the `H5File` objects and the *CompType* is also set the same as in section *DataType* declaration. For each test fissure the `SetUp` function is used for further construction and at the end cleaned with `TearDown` function. To transfer the filenames, which are delivered over the command line arguments, global variables are used. In the respective test fissures the data of the two files will be compared relative to an absolute tolerance error. Hereby it has to be known which time points can be compared to in a general case. This is done with the `time_matching` function which saves the matching time points in vector, where every entry are two indices, and as arguments two paths, represented as strings with the same form as in group section, to the respective time grid *DataSet*.

Chapter 5

Conclusion

Appendix A

Dummy Appendix

You can defer lengthy calculations that would otherwise only interrupt the flow of your thesis to an appendix.

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 Documentation. https://github.com/google/googletest/tree/master/googletest/docs, 2008.

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

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

[10] 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.

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

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

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

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

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

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