ETH Zurich

Porting WaveBlocksND Matrix Potential Functionality to C++

Author:
Lionel Miserez

Supervisor:
Dr. Vasile Gradinaru

Advisor:
Raoul Bourquin

HS15

ETH zürich

Contents

1 Introduction and Motivation

A number of projects aim to port `WaveBlocksND` (see [2]) from a python implementation to a `C++` implementation. My task was the porting of the functionality responsible for handling matrix potentials and the Hagedorn propagator. In this report I will outline the desired functionality and elaborate on my design decisions leading to the current template-heavy implementation.

1.1 Background

`WaveBlocksND` is a python library to simulate the time-dependent behavior of semiclassical wavepackets on energy surfaces. These energy surfaces are described by matrix potentials $V : \mathbb{R}^D \to \mathbb{R}^{N \times N}$. The simulation is advanced using a propagator which updates the parameters of a given wavepacket given V and a timestep dt. One such propagator is the Hagedorn propagator (see [3], [7]). The following sections give a quick recapitulation of the relevant operations and properties required from these matrix potentials in `WaveBlocksND` (see [1]).

1.1.1 Matrix Potentials

Consider matrix valued mappings from real space $\mathcal{C} : \mathbb{R}^D \to \mathbb{R}^{N \times N}$ [1]. One can write these mappings in the form of a matrix parameterized with an $\mathbb{R}^D$ vector as follows:

$$\mathcal{C}(v) = \begin{pmatrix} c_{1,1}(v) & c_{1,2}(v) & \cdots & c_{1,N}(v) \\ c_{2,1}(v) & c_{2,2}(v) & \cdots & c_{2,N}(v) \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,1}(v) & c_{N,2}(v) & \cdots & c_{N,N}(v) \end{pmatrix} \tag{1.1}$$

where $c_{i,j} : \mathbb{R}^D \to \mathbb{R}$.

1.1.2 Derivatives

Using formulation 1.1, one can define the Jacobian of $\mathcal{C}$ as:

$$\begin{bmatrix} Dc_{1,1}(v) & Dc_{1,2}(v) & \cdots & Dc_{1,N}(v) \\ Dc_{2,1}(v) & Dc_{2,2}(v) & \cdots & Dc_{2,N}(v) \\ \vdots & \vdots & \ddots & \vdots \\ Dc_{N,1}(v) & Dc_{N,2}(v) & \cdots & Dc_{N,N}(v) \end{bmatrix} \tag{1.2}$$

where the use of the square brackets indicates that we are merely thinking of the matrix as a container rather than the mathematical object.

The Hessian of $\mathcal{C}$ can be defined accordingly.

[1] For certain set-ups it might be easier to consider complex matrix valued functions but the following statements can still be applied easily in that case as well.

1.1.3 Quadratic Approximation

The quadratic approximation $Q[f] : \mathbb{R}^D \to \mathbb{R}$ of a function $f : \mathbb{R}^D \to \mathbb{R}$ around some point $q \in \mathbb{R}^D$ is written as

$$Q[f](x) = f(q) + Df(q) \cdot (x - q) + \frac{1}{2}(x - q)^T \cdot D^2 f(q) \cdot (x - q) \tag{1.3}$$

Similarly to the derivatives one can also express the element-wise quadratic approximation $Q[\mathcal{C}]$ of $\mathcal{C}$ as:

$$\begin{bmatrix} Q[c_{1,1}](v) & Q[c_{1,2}](v) & \cdots & Q[c_{1,N}](v) \\ Q[c_{2,1}](v) & Q[c_{2,2}](v) & \cdots & Q[c_{2,N}](v) \\ \vdots & \vdots & \ddots & \vdots \\ Q[c_{N,1}](v) & Q[c_{N,2}](v) & \cdots & Q[c_{N,N}](v) \end{bmatrix} \tag{1.4}$$

1.1.4 Eigenvalues

While in principle, one can compute the eigenvalues and eigenvectors of $\mathcal{C}(x)$ point-wise as $\lambda_i(x)$ and $v_i(x)$ and then define the mapping $\mathcal{D} : \mathbb{R}^D \to \mathbb{C}^{N \times N}$ as:

$$\mathcal{D}(x) = \begin{pmatrix} \lambda_1(x) & 0 & \cdots & 0 \\ 0 & \lambda_2(x) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N(x) \end{pmatrix} \tag{1.5}$$

Furthermore, one can define the mapping $\mathcal{V} : \mathbb{R}^D \to \mathbb{C}^{N \times N}$ as:

$$\mathcal{V}(x) = \begin{pmatrix} v_1(x) & v_2(x) & \cdots & v_N(x) \end{pmatrix} \tag{1.6}$$

one has to take care that the ordering of these *eigenfunctions* does not change across evaluation points. Since my implementation ignores this problem by forcing the user to provide the *eigenfunctions* explicitly if he or she desires to use them, there is no need to go into further detail.

1.1.4.1 Derivatives and Quadratic Remainder

The derivatives and quadratic approximations are defined for $\mathcal{D}$ analogously to $\mathcal{C}$.

1.1.5 Quadratic Remainder

The quadratic remainder $R[f] : \mathbb{R}^D \to \mathbb{R}$ of a real valued function $f : \mathbb{R}^D \to \mathbb{R}$ around some point $q \in \mathbb{R}^D$ is simply defined as $R[f](x) = f(x) - Q[f](x)$. However, the quadratic remainder of a matrix potential $\mathcal{C}$ is not defined point-wise for the purposes of `WaveBlocksND` but as follows (see [1]):

$$R[\mathcal{C}](x) = \mathcal{C}(x) - Q[\mathcal{L}(x)] \tag{1.7}$$

where either $\mathcal{L}(x) = \lambda_\chi(x) I$ for some *leading level* χ or in the homogeneous case or $\mathcal{L} = \mathcal{D}$.

2 Programming Techniques

In this chapter, I outline some programming techniques and the issues tackled with them during the design of the code.

2.1 A static approach to abstract base classes

In this section, I quickly outline a common technique to obtain some of the functionality that virtual function calls provide without incurring the runtime costs associated with such calls.

2.1.1 Curiously recurring template pattern (CRTP)

CRTP (term coined in [8]) is a template pattern where a class template `B` expecting a class as parameter exists and next a class `D` is defined to inherit from an instantiation of `B` with the subtype `D` as argument.

```
template<class Subtype>
struct Base {};

struct Derived : Base<Derived> {};
```

2.1.2 Static Polymorphism

Static polymorphism[1] is an attempt to simulate some behavior of dynamic polymorphism (such as virtual functions) without the run-time costs associated. To achieve this one can make use of the CRTP.

```
template<class Subtype>
struct Base {
        int foo(int x) {
        return static_cast<Subtype*>(this)->foo_implementation(x);
    }
    int bar(int x) {
        return foo(x)+1;
    }
};

struct Derived : Base<Derived> {
        int foo_implementation(int x) { return x;}
};

struct OtherDerived : Base<OtherDerived> {
        int foo_implementation(int x) { return 2*x;}
};
```

In the example above we can see that there now is an interface `Base` indicating which functions are available and two implementations `Derived` and `OtherDerived` which implement the interface in different manners. Note that `Derived` and `OtherDerived` are not subtypes of the same class as each template instantiation defines its own base class. We

[1] See for example the Wikipedia article https://en.wikipedia.org/wiki/Template_metaprogramming#Static_polymorphism

therefore lose the flexibility of choosing the implementation at runtime but gain the speed-up of statically linking to the implementation.

This really becomes useful when we want an abstract base class to provide some general functionality and then introduce specializations without the cost of having virtual function calls.

In the above example both implementations inherit the common functionality `bar` from their respective base class while defining their own implementation of `foo`.

2.2 Multiple Inheritance and Linearization

`C++` allows a class to inherit from multiple base classes. While this means that we can easily combine implementations it is not without disadvantages. In this section, I wish to outline a commonly occuring issue with multiple inheritance, possible solutions thereof and finally the application of one of these solutions to my implementation.[2] I particularly wish to discuss a problem arising from diamond shaped inheritance.

2.2.1 Problem

```
struct A {
    virtual int bar(int x) {
        return x+1;
    }
};

class B1 : public A {
};

class B2 : public A {
};

class C : public B1, public B2 {
        int foo() {
                return bar(2); // Which bar?
    }
};
```

An ambiguity arises with multiple inheritance whenever two or more of the base classes share a base class. In the example above, one has to explicitly select either `B1::bar` or `B2::bar` in the example above. Alternatively, one can make the inheritance from `A virtual`. While this isn't without issue itself (especially since it requires foresight from the writers of `B1` and `B2` to inherit only virtually), my main concern here was the virtual inheritance itself since I wanted to avoid runtime penalties at all costs.

2.2.2 Solution

A common trick is to simply template the middle classes on their super class and forcing the writer of the child class `C` to choose the linearization order when inheriting[3]. While `B1` and `B2` can now inherit from `A` it is also possible to

[2] The elaborations in this section are largely based on Chapters "Multiple Inheritance" and "Linearization" of the lecture "Concepts of Object Oriented Programming" by Prof. Peter Müller. See the lecture notes at https://www1.ethz.ch/pminf/education/courses/coop.

[3] This somewhat emulates the linearization of traits as in Scala.

linearize the diamond shaped inheritance to single inheritance. While this solution certainly isn't without issues of its own[4], I found it quite useful.

```
struct A {
    virtual int bar(int x) {
        return x+1;
    }
};

template<class Super>
class B1 : public Super {
};

template<class Super>
class B2 : public Super {
};

class C : public B1<B2<A>> {
        int foo() {
                return bar(2); // Which bar? Last override!
    }
};
```

2.2.3 Application

I make use of multiple inheritance to create a collection of modules which can be mixed and matched to provide only the functionality required to the class. To avoid diamond shaped inheritance every implementation that needs to inherit from some other implementation is templated on its super class. As an example of the mixing of modules consider a user that wants to create a potential only to evaluate it in some points and a user that wishes to also evaluate the derivatives. Both can make use of the same implementation for evaluation which requires both users to specify the potential. However, only the user that wishes to mix in (via inheritance) the functionality to also evaluate the derivatives is required to provide these during construction. This allows the users to easily create classes which only require and offer the functionality they desire.

2.3 Partial specializations

C++ does not allow partial specialization of template functions. It does, however, allow partial specialization of template classes. Therefore, a well known trick is to use a static member function of a template class, to effectively emulate partial specializations of functions.

[4]See again Prof. Peter Müller's lecture "Concepts of Object Oriented Programming" and its discussion of Scala traits as an example.

```
template<int N, int D>
int bar() {return 0;}

template<int N>
int bar<N,1>(){return 1;} // not allowed

template<int N, int D>
struct A {
        static int foo(){return 0;}
};

template<int N>
struct A<N,1> {
        static int foo(){return 1;} // allowed
};

int main() {
        A<2,2>::foo(); // 0
        A<2,1>::foo(); // 1
}
```

2.4 Template specialization and template aliases

During the coding I came across one strange issue related to template aliases which stumped me so much that I had to resort to asking a question on `stackoverflow.com`[5].

Consider an alias template `A`. Now let `B` be an alias template of `A`. In the code below these class templates are used as template arguments for a struct `C` which is only specialized for one of the templates (`A`). `clang -std=c++11` exits with `error: implicit instantiation of undefined template 'C<B>'` indicating that another specialization for `B` is needed.

```
template<int N>
using A = int;

template<int N>
using B = A<N>;

template<template<int> class I>
struct C;

template<>
struct C<A> {};

int main() {
  C<A> c;
  C<B> d; // clang error: implicit instantiation
}
```

It is strange that - despite not allowing specializations of aliases and therefore guaranteeing that both aliases will *behave* the same - `A` and `B` are treated as different class templates because the standard only guarantees that `A<X>` and `B<X>` are the same for every class `X`. Logically we should then be able to assume that `A` and `B` are in fact the same

[5]The following description of the issue is very similar to the one I submitted to http://stackoverflow.com/questions/32723988/alias-of-class-template and the explanation of the issue is adapted from the accepted answer http://stackoverflow.com/a/32724096/3139931 by user 'Barry'.

since they behave exactly the same but there are no guarantees from the standard (and `clang` for example does treat them as two different templates). This is CWG issue #1286[6]

Concretely, this problem means that we initially could not use an alias template for the really wordy typenames `matrixPotentials::bases::Canonical` and `matrixPotentials::bases::Eigen` since some implementations used to be specialized on those templates and would not have been guaranteed to be specialized on an alias template of these typenames. To spare the user from typing this wordy symbol one could either move them out of their nested namespaces, resort to using a macro or refactor the code to remove any templates specialized on a template themselves. I opted for the refactoring. Unfortunately, this issue does now restrict any future implementations from specializing on these basis templates.

[6] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1286

3 Code

3.1 Libraries

The implementation relies heavily on the `Eigen`[6] template library for linear algebra. I have introduced a handful of template aliases to simplify the typenames of the most commonly used objects. Additionally, the code utilizes `eigen3-hdf5`[4] for the serialization of the `Eigen` matrices into `hdf5` files.

3.2 Matrix Potential

In this section, I list all of the modules that can be used to create a matrix potential and elaborate on how to combine them.

3.2.1 Abstract and Standard Implementation

Most modules contain a class called `Abstract` which defines the functionality the module offers and implements some common functionality. This is usually the extension of an evaluation from a single point to a grid of points. Each module then provides at least one implementation (if only one implementation is provided, it is called `Standard`) of `Abstract` which inherits from `Abstract` using static polymorphism as described in the previous chapter. To add his or her own implementations the user can simply follow the pattern of the standard implementation. As an example, a user could define his or her own implementation of the `taylor` module wherein all of the evaluations are hard-coded instead of delegated to lambda functions for a more efficient evaluation.

3.2.2 Namespaces

To keep the global namespace and the `waveblocks` namespace unpolluted with implementation details all matrix potential related implementations are put within their own namespace `matrixPotentials` which contains a namespace `modules` which in turn contains a namespace for each module. Only standard implementations are exported via template aliases.

3.2.2.1 Bases

The `bases` namespace contains the class templates for the basis classes `Canonical` and `Eigen`. These are merely a collection of constants and aliases to make templating easier. Instead of a module depending on a large number of types it simply depends on a *basis* and imports the required types via a macro. `Canonical` and `Eigen` are exported to the `waveblocks` namespace via template aliases.

3.2.2.2 Potentials

The `potentials` namespace contains template aliases for often used configurations of modules such as a scalar valued potential which supports evaluation of the local quadratic remainder.

3.2.2.3 Modules

- `evaluation`: allows to evaluate a potential in one or many points
- `jacobian`: allows to evaluate the Jacobian as defined in Chapter 1 in one or many points
- `hessian`: allows to evaluate the Hessian as defined in Chapter 1 in one or many points
- `exponential`: allows to evaluate the exponential of the resulting matrix of an evaluation in one or many points
- `taylor`: allows to evaluate the potential, Jacobian and Hessian all at once in one or many points
- `localQuadratic`: allows to evaluate the local quadratic of a matrix potential as defined in Chapter 1
- `leadingLevelOwner`: allows to define a potential which owns another potential
- `localRemainder`: allows to define a potential whose local quadratic remainder can be computed as defined in Chapter 1

Standard Implementations Figure 3.1 shows the class diagram of the standard implementations of the modules available. Note that this class diagram only holds when using the template aliases provided by each module (i.e. using `waveblocks::Evaluation` rather than `waveblocks::matrixPotentials::modules::evaluation::Standard` as the standard implementations are usually more general and allow passing the super class as a template argument in most cases.

3.3 Hagedorn Propagator

The Hagedorn propagator is implemented as a template class templated on the number of levels N, the dimension D, as well as the class `MultiIndex`, used for indexing the wavepacket, and the class `TQR`, which defines the tensor quadrature rule. It offers the static method propagate which is overloaded for homogeneous and inhomogeneous wavepackets and specialized for scalar wavepackets. The propagator needs different implementations for $N = 1$, $N > 1$, $D = 1$, $D > 1$ as well as different implementations for homogeneous and inhomogeneous wavepackets. However, there exists a lot of overlap in these implementations which I make heavy use of by delegating non-overlapping parts to partially specialized template functions. As an example we can see that for the first step the inhomogeneous case simply iterates over all components and applies the same operation to the parameter set of the Hagedorn wavepacket propagated as the homogeneous case does. Similarly, there are only a few lines of code that need to be adjusted when dealing with $D = 1$.

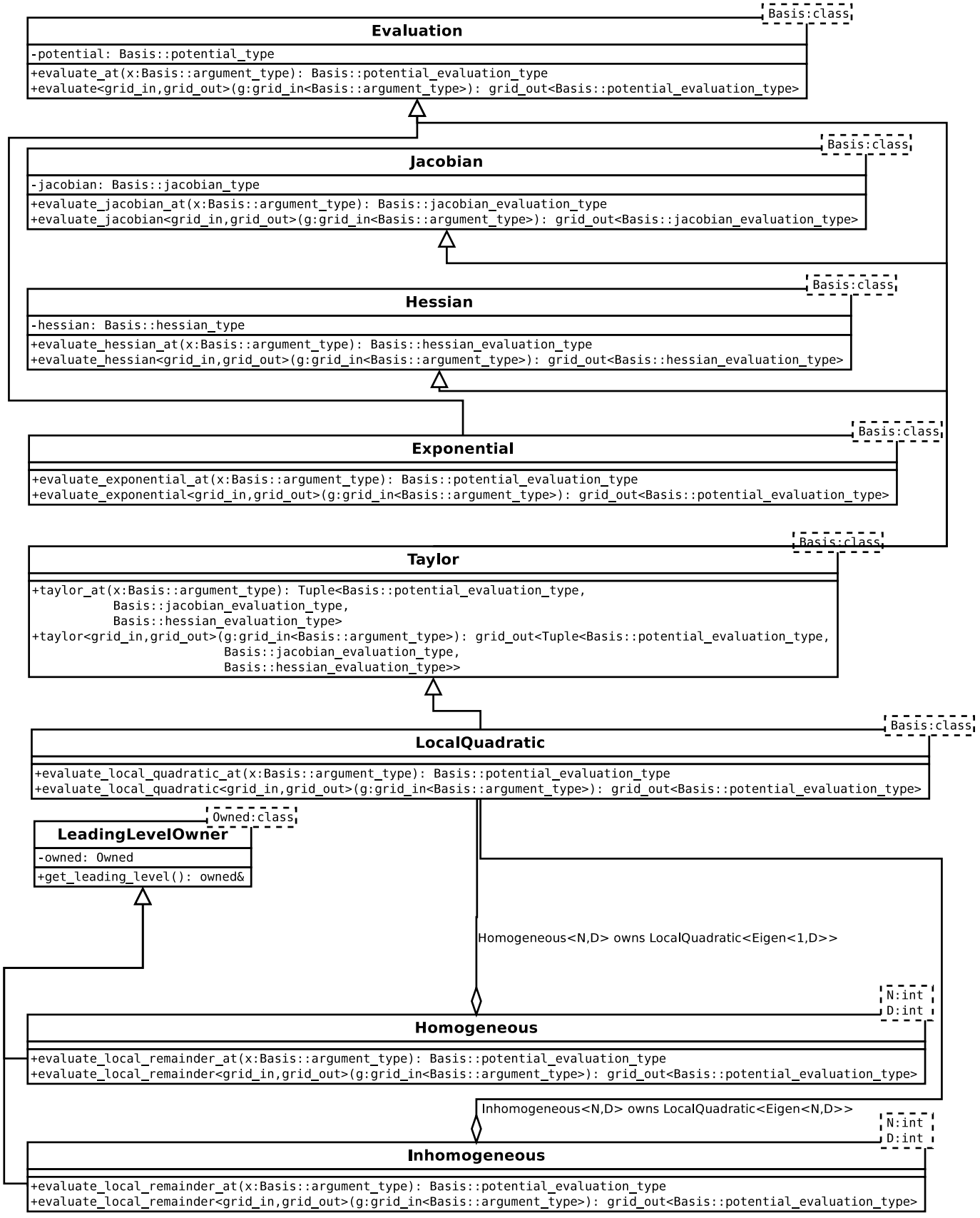

FIGURE 3.1: Diagram of the standard implementations of the modules available and their class hierarchy. All of these classes are available in the `waveblocks` namespace.

4 Simulations

4.1 Verification

To verify the implementation, I applied it to a 2D harmonic oscillator as well as a 1D anharmonic oscillator and compared the results to the results obtained with `WaveBlocksND`. The computation attempts to find solutions to the time-dependent Schrödinger equation in the semi-classical scaling:

$$i\varepsilon \frac{\partial}{\partial t}\Psi(\mathbf{x},t) = H\Psi(\mathbf{x},t) \tag{4.1}$$

where $H = T + V$ and $T = -\sum_{j=1}^{D} \varepsilon^2 \frac{\partial^2}{\partial \mathbf{x}_j^2}$ and V is a potential.

4.1.1 Harmonic Oscillator

Consider a 2D harmonic potential. A Gaussian Hagedorn wavepacket with 5 cubic shape basis functions with the following initial parameters:

q	p	Q	P	S	ε
$\begin{pmatrix} -3 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0.5 \end{pmatrix}$	I	iI	0	0.1

is propagated using the Hagedorn propagator with $dt = 0.01$ up to $T = 12$.

In Figures 4.1, 4.2, 4.3 and 4.4 one observes conservation of energy and periodic behavior in the parameters of the wavepacket as well as constant coefficients, which is as expected since the local remainder must be zero. These results align neatly with the ones computed by `WaveBlocksND` giving confidence that at least the harmonic part of the implementation in the higher dimensional case is correct.

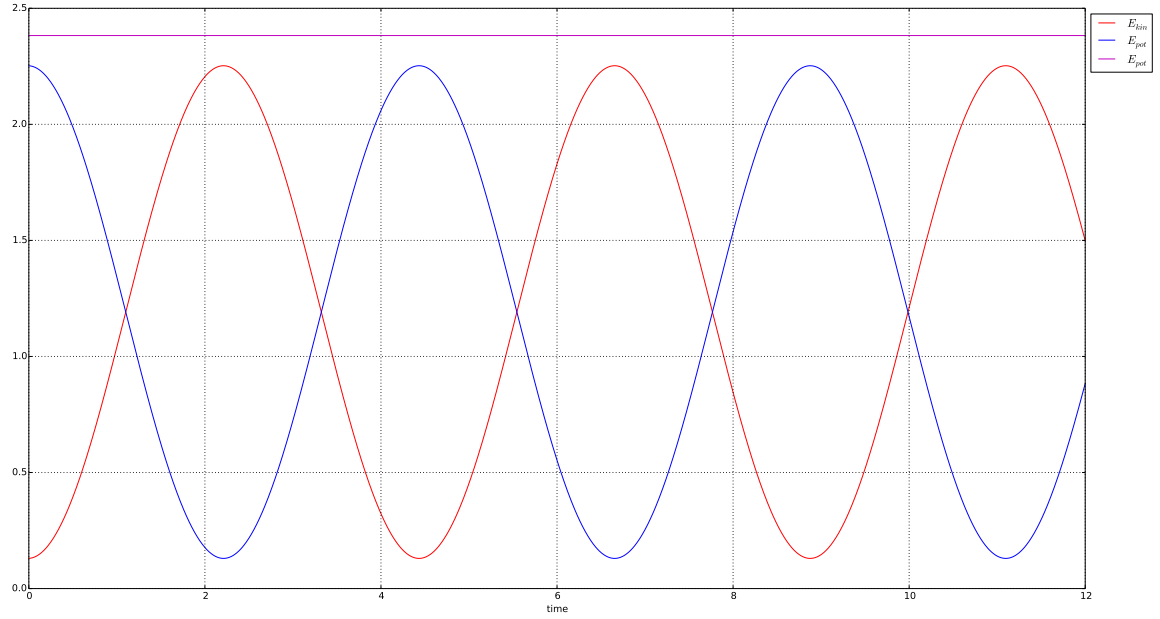

FIGURE 4.1: The energies of the harmonic simulation.

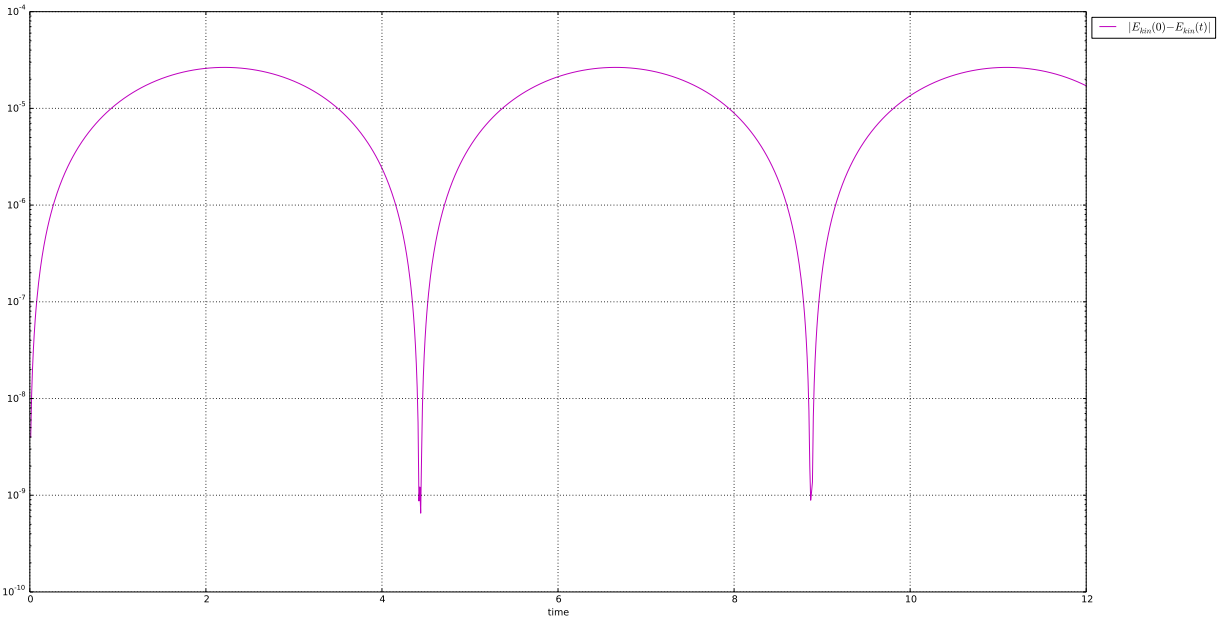

FIGURE 4.2: The drift of the total energies in the harmonic simulation.

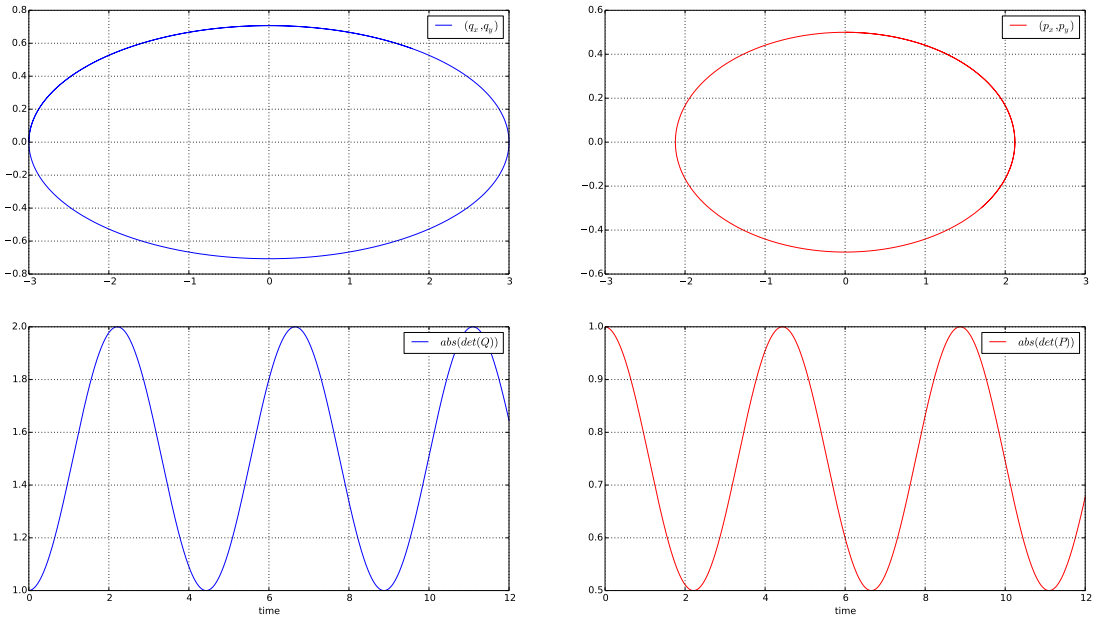

FIGURE 4.3: The position and momentum parameters of the harmonic simulation.

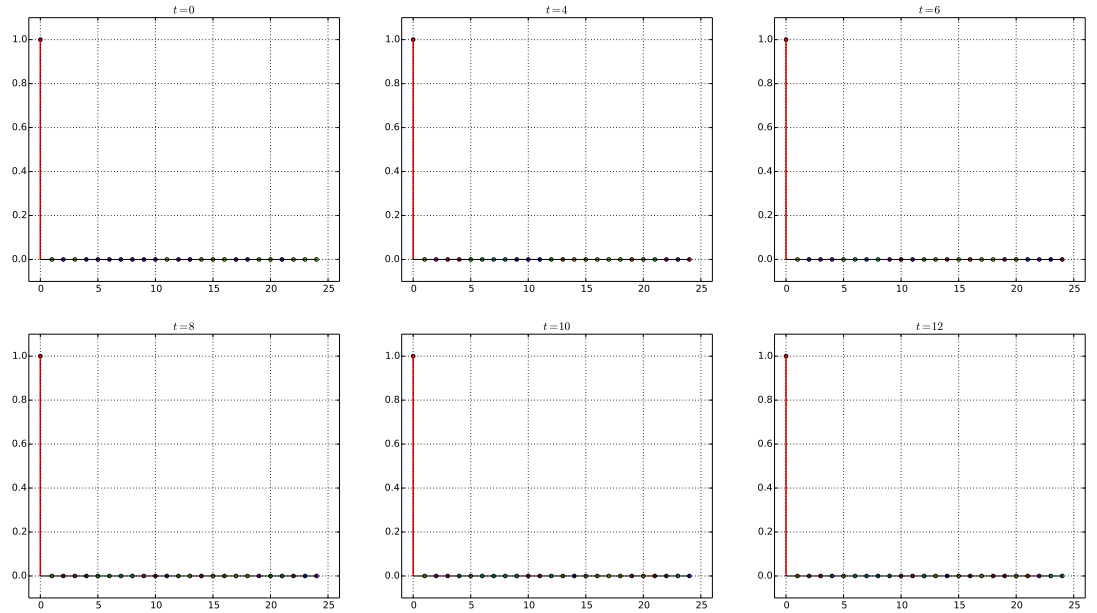

FIGURE 4.4: The coefficients of the harmonic simulation.

4.1.2 Anharmonic Oscillator

This example considers the potential $V : \mathbb{R} \to \mathbb{R}$ where $V(x) = 1 + x^4$.

A Gaussian Hagedorn wavepacket with 128 cubic shape basis functions with the following initial parameters:

q	p	Q	P	S	ε
0	1	1	i	0	0.1

is propagated using the Hagedorn propagator with $dt = 0.01$ up to $T = 6$.

Figure 4.6 shows quite neat conservation of energy until the second reflection at which point the simulation seems to break down. This observation as well as the other plots are consistent with the `WaveBlocksND` implementation of the same example which serves as an indicator that the implementation is also correct in the anharmonic part.

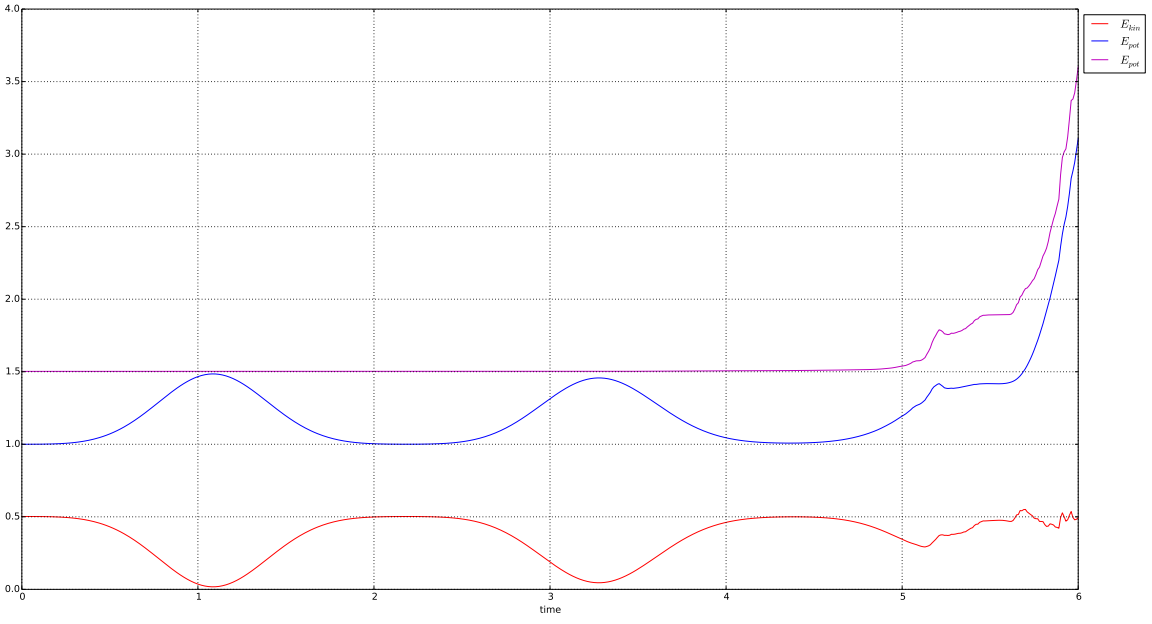

FIGURE 4.5: The energies of the anharmonic simulation.

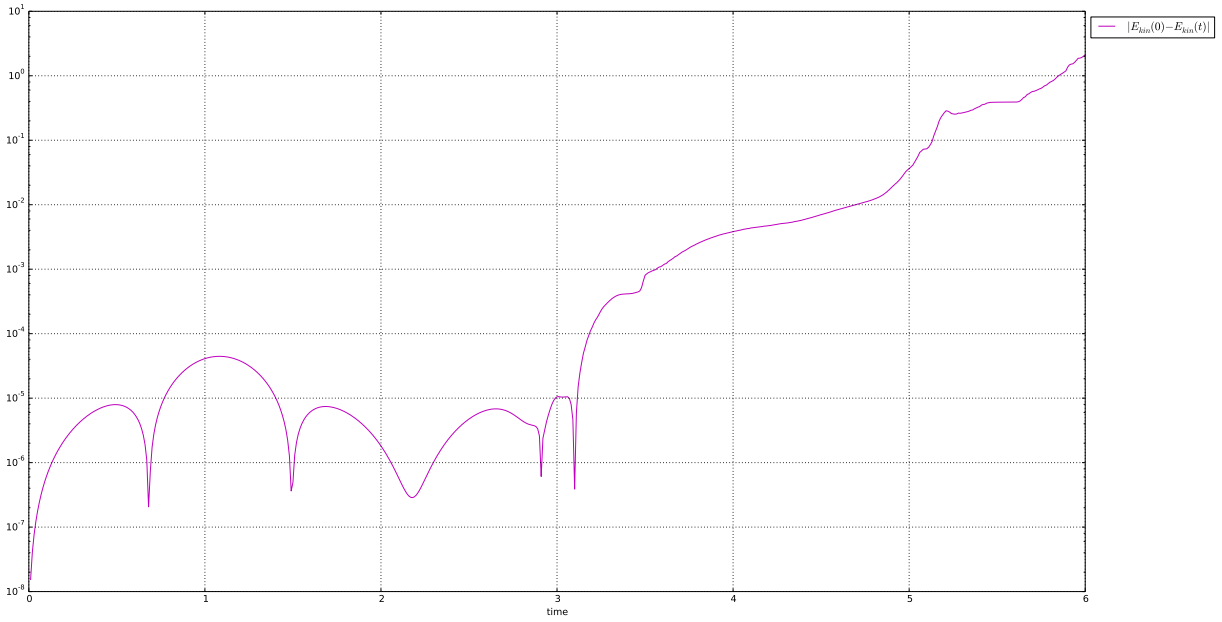

FIGURE 4.6: The drift of the total energies in the anharmonic simulation.

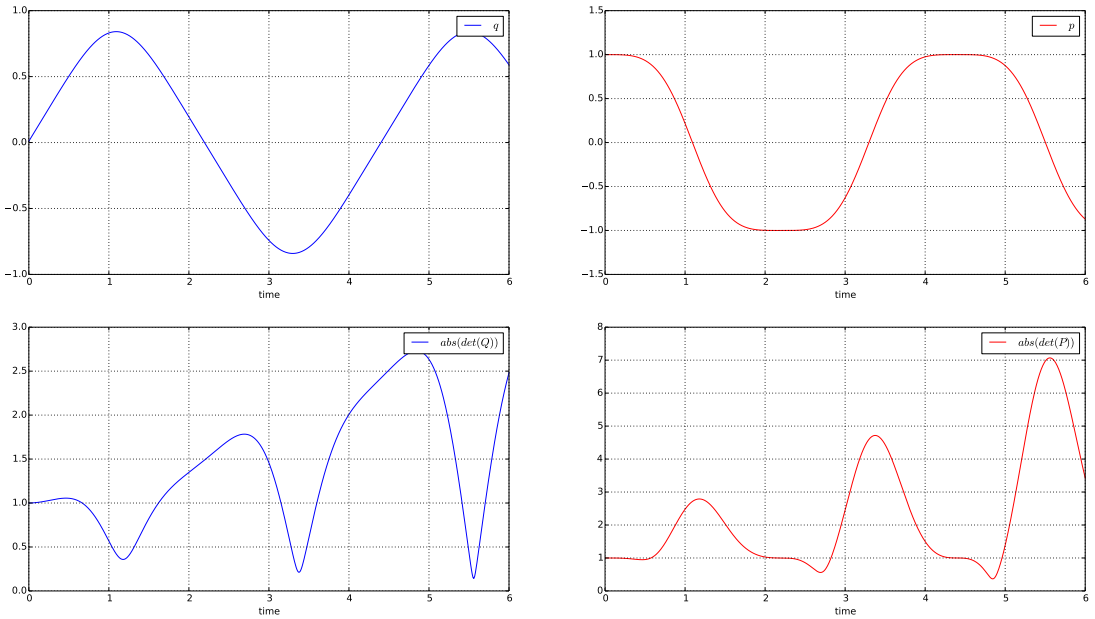

FIGURE 4.7: The position and momentum parameters of the harmonic simulation.

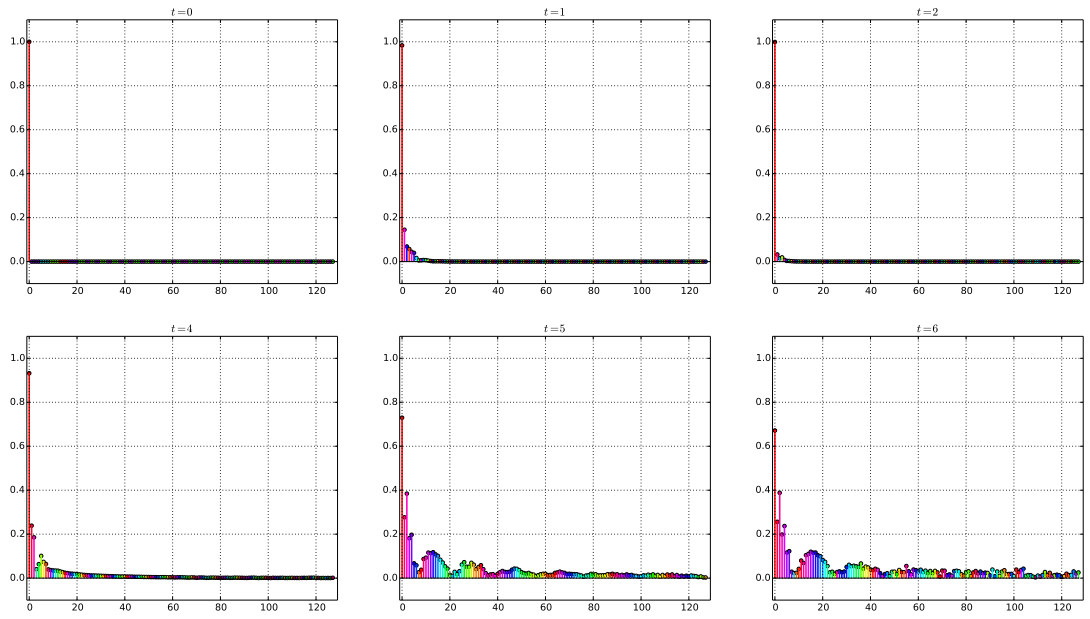

FIGURE 4.8: The coefficients of the harmonic simulation.

4.1.3 Conclusion

Both simulations were consistent with the `WaveBlocksND` implementation which gives confidence that the implementation is correct for $N = 1$.

4.2 1D Tunneling

I attempt to recompute the simulation in [5] as an application of the `C++` framework.

4.2.1 Setup

Consider the scalar potential $V : \mathbb{R} \to \mathbb{R}$ with $V(x) = \frac{\sigma}{\cosh(x/a)^2}$, where $\sigma = 100\frac{\text{kJ}}{\text{mol}} = 0.038088$ Eh and $a = 0.5$ Å $= 0.944858$ alu (as in Figure 4.9).

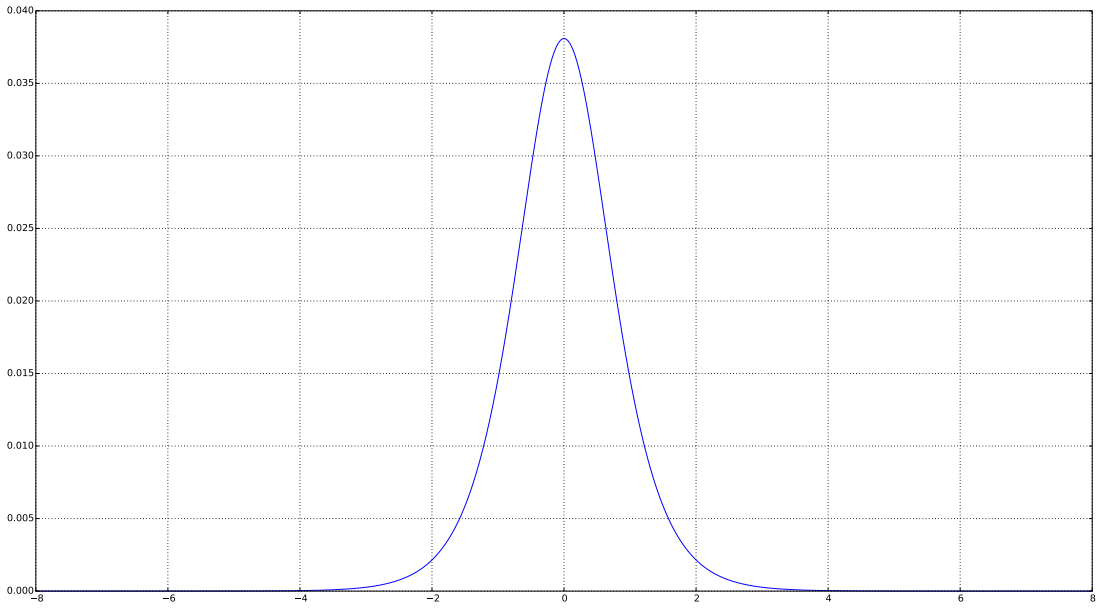

FIGURE 4.9: Potential of the Tunneling Simulation

A Gaussian Hagedorn wavepacket with 512 cubic shape basis functions with the following initial parameters:

q	p	Q	P	S	ε
-7.5589045088306	0.2478854736792	3.5355339059327	0.28284271247461	0	0.02342

is propagated using the Hagedorn propagator with $dt = 0.005$ up to $T = 70$.

4.2.2 Observations

Figures 4.11 and 4.10 show the potential and kinetic energies and the conservation of the total energy. In Figure 4.12 one can observe the packet being reflected at the origin and thus reversing its momentum. The `C++` implementation appears to reproduce the data for Figure 1 in [5] as evidenced by Figures 4.16, 4.16 and 4.14. Moreover, Figure 4.13 shows the increase in higher order coefficients when the packet gets closer to the origin which is responsible for the bump on right side of Figures 4.15 and 4.16.

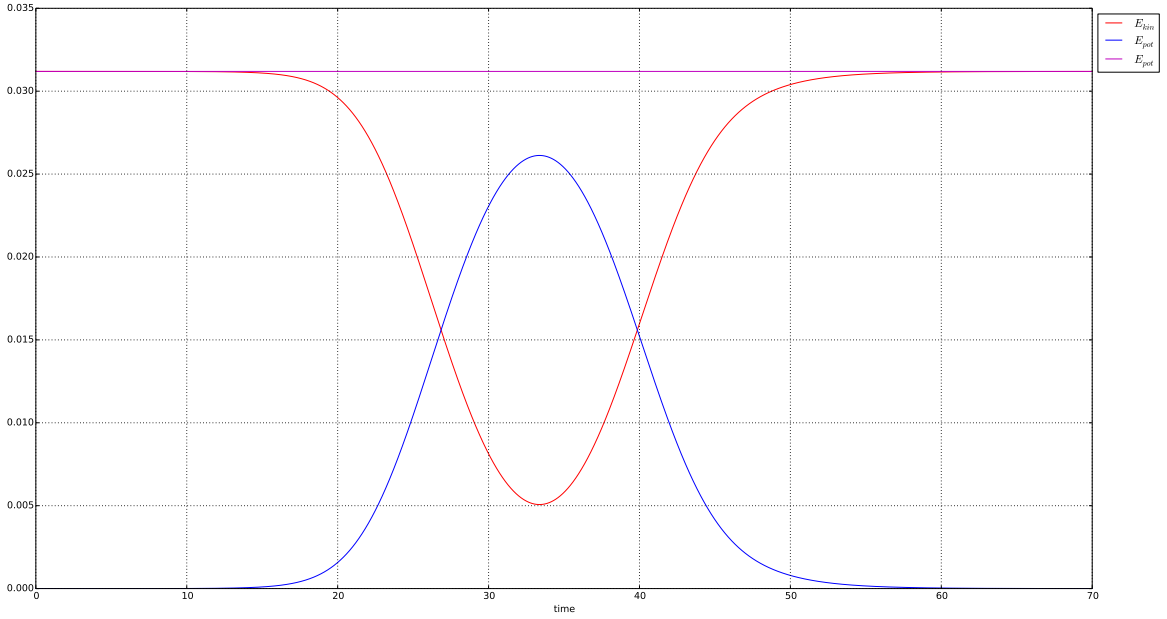

FIGURE 4.10: Energies of the Tunneling Simulation

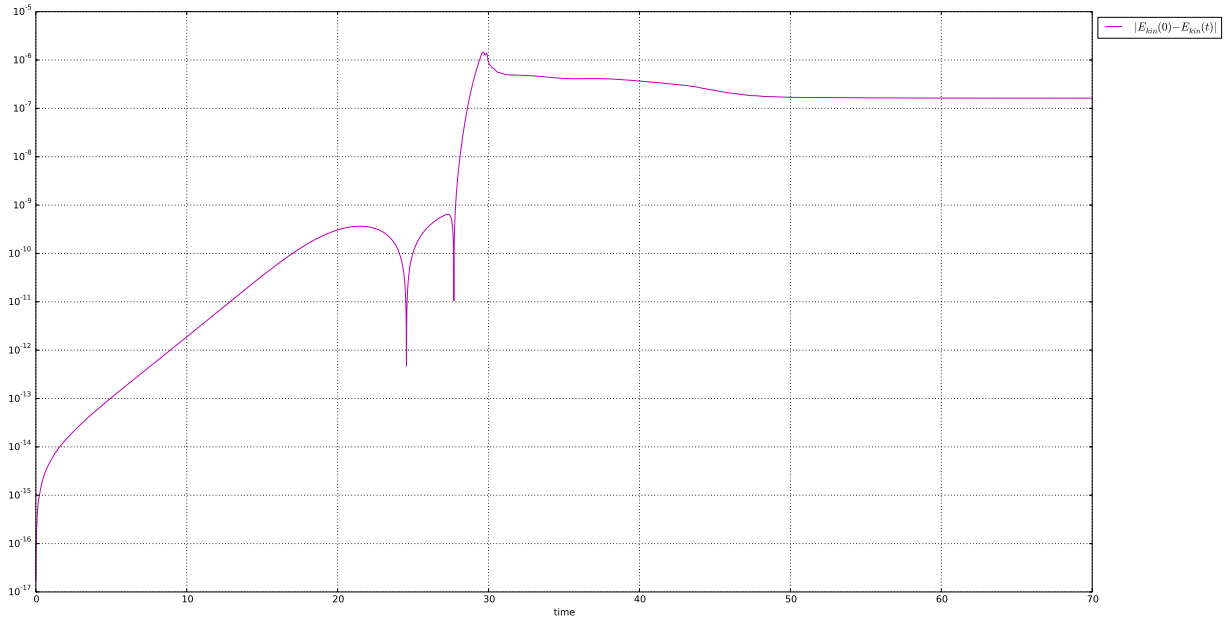

FIGURE 4.11: Drift of Total Energy of the Tunneling Simulation

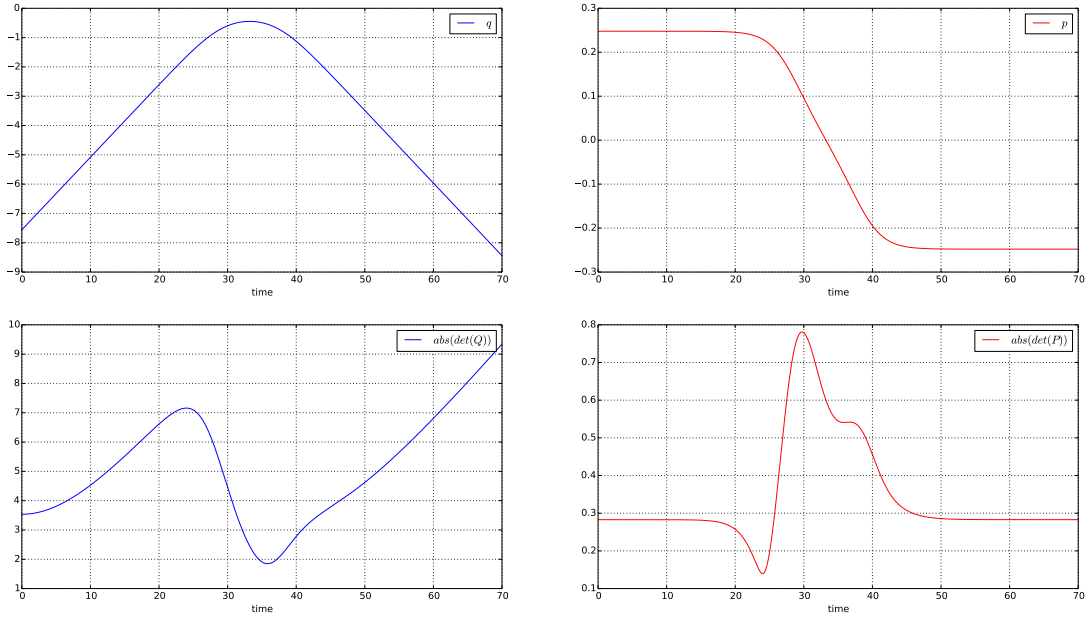

FIGURE 4.12: Position and Momentum Parameters

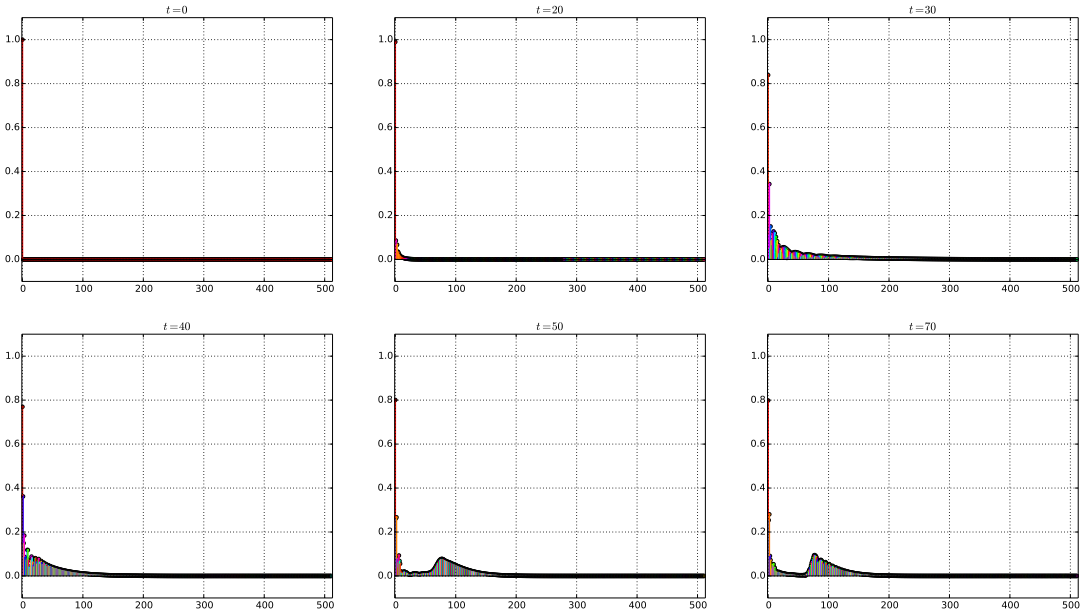

FIGURE 4.13: Coefficients

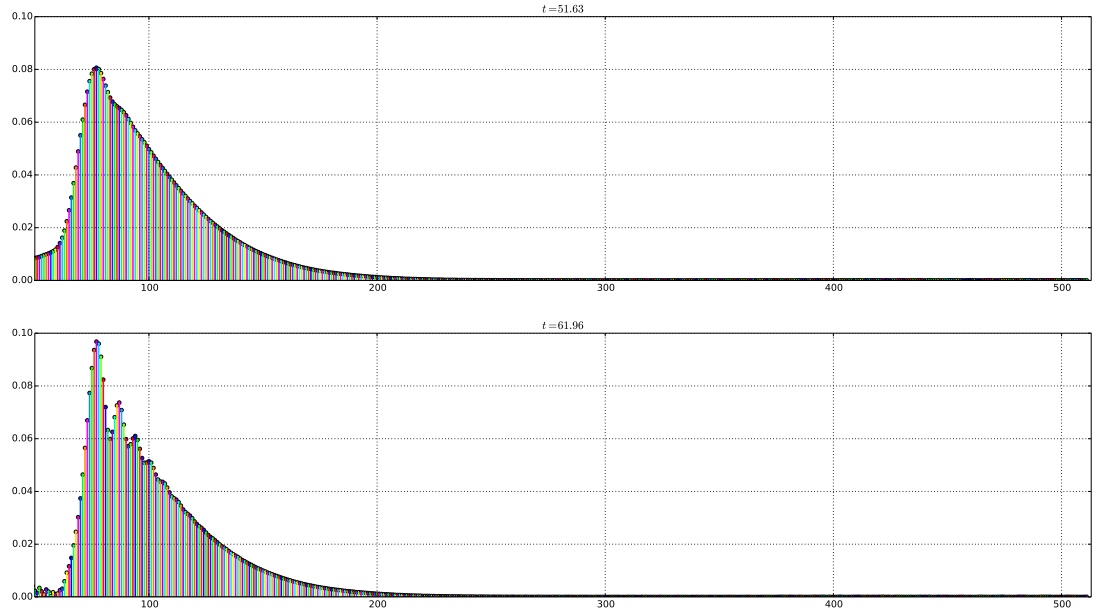

FIGURE 4.14: close-up of Higher Order Coefficients

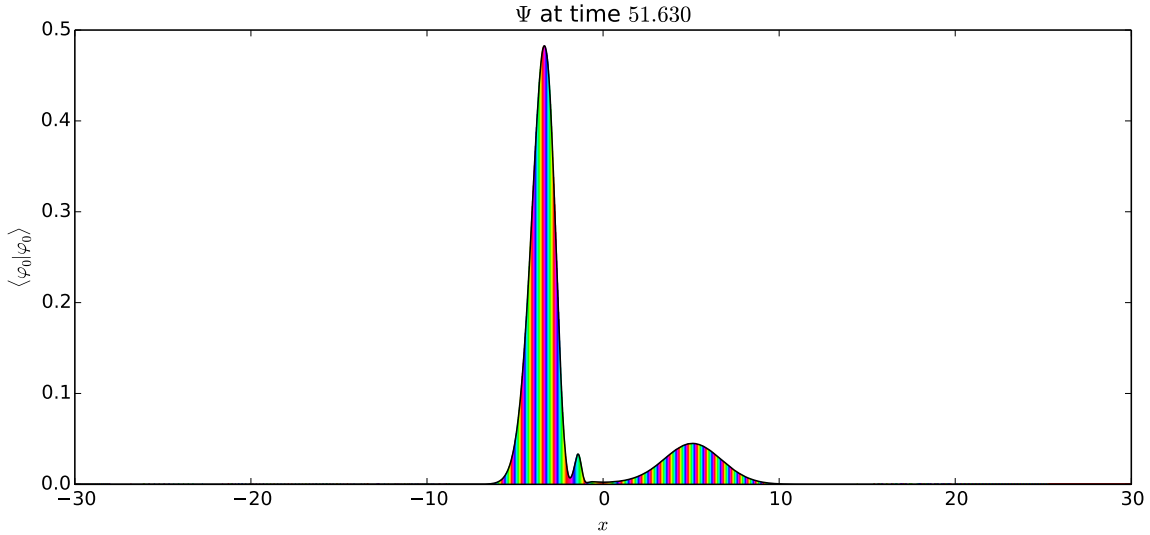

FIGURE 4.15: $|u^2|$ at $t = 51.63$

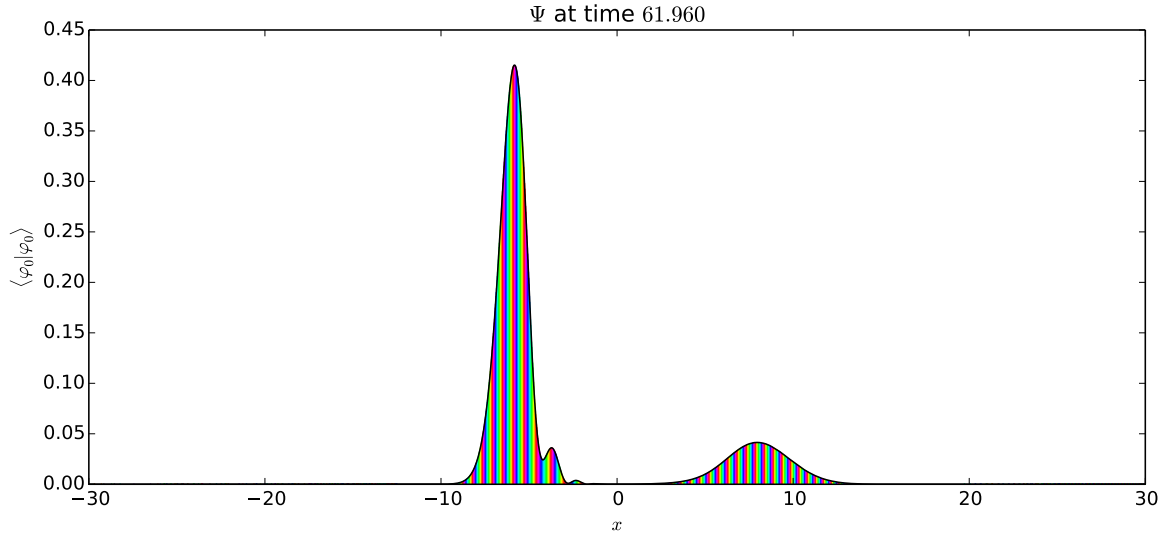

FIGURE 4.16: $|u^2|$ at $t = 61.96$

Bibliography

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

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

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

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

[6] Gal Guennebaud, Benot Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

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

[8] Coplien James O. Curiously recurring template patterns. *C++ Report*, pages 24–27, 1995. http://sites.google.com/a/gertrudandcope.com/info/Publications/InheritedTemplate.pdf.