Time Propagation of Hagedorn Wave Packets

an efficient implementation in C++

Bachelor Thesis

written by
Matthias Untergassmair

advised by
Raoul Bourquin

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

Seminar for Applied Mathematics
ETH Zurich

Fall semester 2016

This article explains the central design decisions and most important code details related to the efficient implementation of time propagation for Hagedorn wave packets in the WaveBlocks framework. A short introduction to operator splitting and a summary of some of the most important time propagation schemes for Hagedorn wave packets are also given. Finally, some benchmarking results are presented which analyze energy conservation and compare different propagators, splitting schemes, step sizes and dimensionalities.

The code to the project can be found on Github at [5].

Contents

List of Algorithms

Introduction

With the growing understanding for quantum mechanics and dynamics, one can attempt to solve increasingly complex systems and computational problems. Without doubt, time propagation is at the core of simulating quantum systems and getting a better understanding for them.

The aim of this project was to give a summary of some state of the art quantum time propagators and provide an efficient C++ implementation for evolving Hagedorn wave packets in time. The produced code is an addition to the C++ WaveBlocks framework whose functionality has already been extended in the context of preceding student projects. The entire WaveBlocks project, which is partially based on an existing Python project with the same name, can be found at [5].

Section 1 repeats some of the most important mathematical ideas underlying the time propagation of wave packets and introduces the notations that will be used throughout the rest of the document. Based on these mathematical observations, section 2 prepares a toolbox of essential building blocks that will help in the construction of propagators as well as for the final understanding of the source code. In section 3, a series of numerical propagators for quantum wave packets are presented and expressed in terms of the components provided earlier. Some detailed insights into the most important technical ideas of the C++ implementation are given in section 4. Finally, section 5 presents the results of a series of numerical experiments to show the correctness of the implemented algorithms and analyze some performance aspects like the scaling with step size, dimensionality of the wave packet and order of the splitting coefficients.

1. Operator Splitting for Time Propagation

In this report, we will use the variable D for describing the number of dimensions and N for the number of energy levels in the system. The semiclassical time-dependent Schrödinger equation is of the form

$$\imath\varepsilon\, \partial_t \psi(\boldsymbol{x}, t) = \mathcal{H}(\varepsilon)\, \psi(\boldsymbol{x}, t) \tag{1}$$

with Hamiltonian operator $\mathcal{H}$, wave function $\psi(\boldsymbol{x}, t)$ depending on the spatial variables $\boldsymbol{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$ and the time variable $t \in \mathbb{R}$. The parameter ε, also known as the semiclassical parameter, determines the degree of influence of quantum dynamics, approaching classical dynamics in the limit $\varepsilon \to 0$ and full quantum mechanics for $\varepsilon = 1$.
Moreover, ε plays an important role in the stability of numerical schemes. As underlined in [8], a small parameter ε can often impose severe constraints on the step size of splitting methods and significantly increase the error. For example, the error was shown to be proportional to ε^{-2} for Lie-Trotter splitting, Strang splitting and other related methods, meaning that smaller ε would lead to a larger error.

The solution to the above Schrödinger equation 1 can be approximated as a finite linear combination of Hagedorn functions

$$\psi(\boldsymbol{x}, t) \approx u(\boldsymbol{x}, t) = e^{\imath S(t)/\varepsilon^2} \sum_{k \in \mathfrak{K}} c_k(t) \varphi_k^{\varepsilon}[\boldsymbol{\Pi}(t)](\boldsymbol{x}) \tag{2}$$

where $\mathfrak{K}$ represents a finite multi-index set, $c_k \in \mathbb{C}$ are coefficients that depend on time only, $S(t)$ is a phase factor and $\boldsymbol{q}, \boldsymbol{p} \in \mathbb{R}^D$ and $\boldsymbol{Q}, \boldsymbol{P} \in \mathbb{C}^{D \times D}$ are some parameters. For convenience, the notation for the parameter pack $\boldsymbol{\Pi}(t) = (\boldsymbol{q}(t), \boldsymbol{p}(t), \boldsymbol{Q}(t), \boldsymbol{P}(t))$ is introduced. It is worth noting that $\varphi_k^{\varepsilon}[\boldsymbol{\Pi}(t)]$ depends on time t only implicitly through the time dependency of $\boldsymbol{q}(t), \boldsymbol{p}(t), \boldsymbol{Q}(t), \boldsymbol{P}(t)$. A more detailed analysis of Hagedorn functions is given in [6] and a comprehensive overview over the relevant mathematical details can be found in [4].

The Schrödinger equation 1 can be solved using the ansatz

$$\psi(\boldsymbol{x}, t + dt) = \exp\left(-\frac{\imath}{\varepsilon^2} \mathcal{H} t\right)\, \psi(\boldsymbol{x}, t)\ . \tag{3}$$

However, the matrix exponential is very expensive to compute. Instead, in the search for a numerical solution, one can exploit the structure of the Hamiltonian operator $\mathcal{H}$, by splitting it into its kinetic part $\mathcal{T}$ and potential part $\mathcal{V}$. Furthermore, for the rest of the report we will denote by $\mathcal{U}$ and $\mathcal{W}$ the quadratic and non-quadratic component of $\mathcal{V}$ respectively and it will be used that

$$\mathcal{V} = V(\boldsymbol{x}) \tag{4}$$

$$\mathcal{U} = U(\boldsymbol{q}, \boldsymbol{x}) := V(\boldsymbol{q}) + \nabla V(\boldsymbol{q})(\boldsymbol{x} - \boldsymbol{q}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{q})^T \nabla^2 V(\boldsymbol{q})(\boldsymbol{x} - \boldsymbol{q}) \tag{5}$$

$$\mathcal{W} = W(\boldsymbol{q}, \boldsymbol{x}) := V(\boldsymbol{x}) - U(\boldsymbol{q}, \boldsymbol{x})\ . \tag{6}$$

In other words, $\mathcal{U} = U(\boldsymbol{q}, \boldsymbol{x})$ is the second order Taylor expansion of the potential energy $\mathcal{V} = V(\boldsymbol{x})$ around $\boldsymbol{q}$ (the coordinate $\boldsymbol{q}$ will be omitted for simplicity of notation) and $\mathcal{W} = W(\boldsymbol{q}, \boldsymbol{x})$ is the corresponding remainder.

By further recalling the form of the kinetic operator $\mathcal{T}$ (for a particle of mass $m = 1$)

$$\mathcal{T} = -\sum_{j=1}^{D} \frac{\varepsilon^4}{2} \frac{\partial^2}{\partial x_j^2} \tag{7}$$

one can then split the Hamiltonian operator $\mathcal{H}$ into the components

$$\mathcal{H} = \mathcal{T} + \mathcal{V} = \mathcal{T} + \mathcal{U} + \mathcal{W} \, . \tag{8}$$

This splitting is so central to the construction of the time propagators in the following sections that it is worthwhile repeating the most important findings from [6]:

- The free particle Schrödinger equation ($\mathcal{V} = 0$) can be solved exactly and the wave packet remains in Hagedorn wave packet form (equation 2). For time propagation, only the parameters $\mathbf{\Pi}, S$ need to be updated, the coefficients $\{c_k\}_{k \in \mathfrak{K}}$ remain unchanged.

- The Schrödinger equation 1 can be solved exactly in a pure quadratic potential, i.e. in a potential $\mathcal{V} = U(\boldsymbol{x})$ with $\mathcal{T} = 0$. Again, time propagation only affects the parameters $\mathbf{\Pi}, S$, not the coefficients $\{c_k\}_{k \in \mathfrak{K}}$.

- In the case of an arbitrary potential $\mathcal{V} = W(\boldsymbol{x})$ that is not quadratic, a variational approach can be used. A set of Galerkin functions can be propagated by adapting the coefficients $\{c_k\}_{k \in \mathfrak{K}}$ without changing the parameters $\mathbf{\Pi}, S$.

The following sections will heavily make use of these properties in order to build composition methods for time integration of quantum wave packets. A more in depth analysis of composition methods can be found in [9].

2. Common building blocks for time evolution schemes

In order to be able to create an efficient, still flexible, implementation of various time propagators, it is helpful to break them down in common, optimized building blocks in terms of which all (or most) propagators can be expressed. The present section will briefly introduce the most important, generic building blocks that were found necessary in order to build the propagators in section 3. For each of them, it will be assumed that the following wave packet variables are implicitly accessible, without passing them as an argument to the procedure: the parameter pack $\mathbf{\Pi} = (\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{Q}, \boldsymbol{P})$, the phase factor S, the coefficients $\{c_k\}_{k \in \mathfrak{K}}$, the potential function $V(\boldsymbol{q})$ (including Jacobian and Hessian thereof) and the potential remainder $W(\boldsymbol{q})$.

Many of the building blocks depend on the number of energy levels and on whether the wave packet is homogeneous or inhomogeneous. If any of these cases require a special implementation, it is denoted in the signature of the corresponding algorithm.

2.1. Evolve

The convenience function Evolve (algorithm 1) is a wrapper that will carry out the PrePropagate, Propagate and PostPropagate functions. The concrete implementation of the methods PrePropagate, Propagate and PostPropagate is delegated to the derived classes (see section 4 for more details), but will generally be a combination of the basic building blocks that follow in this section.

Algorithm 1 Evolve the wave packet for a time period T

procedure Evolve($T, \Delta t$)

$M_{tot} = T / \Delta t$
PrePropagate(Δt)
for $m = 1, \ldots, M_{tot}$ **do**
  Propagate(Δt)
end for
PostPropagate(Δt)

end procedure

2.2. StepT, StepU and IntSplit

The time stepping with operators $\mathcal{T}$ and $\mathcal{U} = U(\boldsymbol{x})$ follows directly from the propositions in [6] and is outlined in the algorithms 2 and 3 respectively. Both functions take a variable ξ as an argument which describes the size of the time step.
In the inhomogeneous implementation of the algorithms, the subscript n is used to denote the relative parameters at energy level n.

Algorithm 2 Propagate with Kinetic Energy Operator $\mathcal{T}$

procedure StepT[homogeneous](ξ)

$\boldsymbol{q} = \boldsymbol{q} + \xi \boldsymbol{p}$ ▷ update $\boldsymbol{q}$
$\boldsymbol{Q} = \boldsymbol{Q} + \xi \boldsymbol{P}$ ▷ update $\boldsymbol{Q}$
$S = S + \frac{\xi}{2} \boldsymbol{p}^T \boldsymbol{p}$ ▷ update S

end procedure

---

procedure StepT[inhomogeneous](ξ)

for $n = 1, ..., N$ **do** ▷ loop over all energy levels
  $\boldsymbol{q}_n = \boldsymbol{q}_n + \xi \boldsymbol{p}_n$ ▷ update $\boldsymbol{q}$ for level n
  $\boldsymbol{Q}_n = \boldsymbol{Q}_n + \xi \boldsymbol{P}_n$ ▷ update $\boldsymbol{Q}$ for level n
  $S_n = S_n + \frac{\xi}{2} \boldsymbol{p}_n^T \boldsymbol{p}_n$ ▷ update S for level n
end for

end procedure

In the practical implementations of the propagators that will be presented in the next section, propagation with $\mathcal{T}$ and $\mathcal{U}$ is usually replaced by a series of smaller, alternating steps with the two operators. In order to facilitate the use of such an alternating pattern, the helper function IntSplit (algorithm 4) is introduced: it splits the timestep Δt into M_{split} smaller timesteps of size δt and uses the pair of weight lists $\{w_T, w_U\}$ to alternatingly apply operators $\mathcal{T}$ and $\mathcal{U}$ on each of the sub-intervals. More detailed information on the individual parameters to the IntSplit function and the practical implementation can be found in section 4

2.3. StepW and building of the interaction matrix

As pointed out in the previous section, the time propagation for non-quadratic potentials $W(\boldsymbol{x})$ (see algorithm 5) can be achieved by updating only the coefficients $\{c_k\}_{k \in \mathfrak{K}}$. The update rule is

$$\boldsymbol{c}(t + dt) = \exp\left(-\frac{\imath t}{\varepsilon^2} \boldsymbol{F} \right) \boldsymbol{c}(t) \tag{9}$$

where the matrix $\boldsymbol{F} = [f_{k,l}]_{k,l \in \mathfrak{K}}$ has entries

$$f_{k,l} = \langle \varphi_k | W | \varphi_l \rangle = \int_{\mathbb{R}^D} \overline{\varphi_k(\boldsymbol{x})} W(\boldsymbol{x}) \varphi_l(\boldsymbol{x}) \, d\boldsymbol{x} \, . \tag{10}$$

The calculation of such interaction matrices $\boldsymbol{F}$ makes heavy use of the work on inner products that has been presented in [15], another student project in the WaveBlocks

Algorithm 3 Propagate with (Quadratic) Potential Energy Operator $\mathcal{U}$

procedure StepU[homogeneous](ξ)

$\boldsymbol{p} = \boldsymbol{p} - \xi \nabla V(\boldsymbol{q})$ ▷ update $\boldsymbol{p}$
$\boldsymbol{P} = \boldsymbol{P} - \xi \nabla^2 V(\boldsymbol{q}) \boldsymbol{Q}$ ▷ update $\boldsymbol{P}$
$S = S - \xi V(\boldsymbol{q})$ ▷ update S

end procedure

---

procedure StepU[inhomogeneous](ξ)

for $n = 1, ..., N$ **do** ▷ loop over all energy levels
$\boldsymbol{p}_n = \boldsymbol{p}_n - \xi \nabla V(\boldsymbol{q}_n)$ ▷ update $\boldsymbol{p}$ for level n
$\boldsymbol{P}_n = \boldsymbol{P}_n - \xi \nabla^2 V(\boldsymbol{q}_n) \boldsymbol{Q}_n$ ▷ update $\boldsymbol{P}$ for level n
$S_n = S_n - \xi V(\boldsymbol{q}_n)$ ▷ update S for level n
end for

end procedure

framework. For the further treatment in this report, it is assumed that an efficient implementation of BuildF is readily available.

Algorithm 4 Split a time interval and alternatingly apply $\mathcal{T}$ and $\mathcal{U}$

procedure IntSplit($\Delta t, M_{split}, \{w_T, w_U\}$)

$\delta t = \Delta t / M_{split}$ ▷ step size for splitting
for $n = 1, \ldots, M_{split}$ **do** ▷ split interval in M_{split} substeps
for α in w_T and β in w_U **do** ▷ alternatingly apply $\mathcal{T}$ and $\mathcal{U}$
StepT($\alpha \cdot \delta t$)
StepU($\beta \cdot \delta t$)
end for
end for

end procedure

Algorithm 5 Propagate with (Non-Quadratic) Potential Energy Operator $\mathcal{W}$

procedure StepW(ξ)
$\boldsymbol{\Sigma} = -\imath \frac{\xi}{\varepsilon^2} \cdot$ BuildF(Π)
$\boldsymbol{c} = \exp\{\boldsymbol{\Sigma}\}\boldsymbol{c}$
end procedure

3. Propagators

This section gives a short overview of the numerical properties of the propagators that were implemented in the context of time propagation. Where appropriate, references are given to a more detailed mathematical analysis of the propagators.

The methods are called in the sequence determined by the EVOLVE function (1) and generally are expressed in terms of the basic building blocks that were presented in the previous section. In order to specify to which propagator each of the implementations belongs, the C++ inspired notation PROPAGATORNAME.PROPAGATE is adapted also for pseudo algorithms. If the methods PREPROPAGATE and POSTPROPAGATE are omitted, it means that they are empty.
Whenever convenient, the direct succession of functions STEPT and STEPU was replaced by a call to INTSPLIT which effectively corresponds to a series of M_{split} alternating calls to STEPT and STEPU (see explanation in 2.2) for better accuracy.

3.1. Hagedorn Propagator

The Hagedorn propagator, shown in algorithm 6, is one of the simplest propagators that can be built by exploiting the numerical properties of the Hamiltonian splitting $\mathcal{H} = \mathcal{T} + \mathcal{U} + \mathcal{W}$ discussed earlier. As pointed out in [6], this simple time stepping scheme has many beneficial properties like the preservation of the L^2 norm of the wave packet, time reversibility and stability in the classical limit $\varepsilon \to 0$. Also, in the limit of $\mathfrak{K}$ approaching the full basis set, the variational approximation used for the propagation with the non-quadratic part $\mathcal{W}$ becomes exact.

Algorithm 6 Single timestep with Hagedorn propagator

procedure HAGEDORN.PROPAGATE(Δt)

STEPT($\frac{\Delta t}{2}$)	▷ Step of size $\Delta t/2$ with $\mathcal{T}$
STEPU(Δt)	▷ Step of size Δt with $\mathcal{U}$
STEPW(Δt)	▷ Step of size Δt with $\mathcal{W}$
STEPT($\frac{\Delta t}{2}$)	▷ Step of size $\Delta t/2$ with $\mathcal{T}$

end procedure

3.2. Semiclassical Propagator

The central idea of the semiclassical splitting, as introduced in [8], is to split the propagation with operators $\mathcal{T}$ and $\mathcal{U}$ into many smaller, alternating steps, thereby reducing the dominating error[1] in the update of $\mathbf{\Pi}$ and S. While there is some additional computational cost caused by a higher number of updates for the parameters $\mathbf{\Pi}$ and

[1] for small ε, the main source of error lies in the updating of $\mathbf{\Pi}$ and S

S, the extra effort is usually negligible compared to the propagation with $\mathcal{W}$ which requires numerical evaluation of multi-dimensional integrals.

In addition, due to the numerical properties of the semiclassical splitting, it even allows to take larger timesteps Δt than conventional splitting methods like the YL-splitting (see [8]), while maintaining the same error.

Finally and most importantly, the error is no longer proportional to $1/\varepsilon^2$ but instead scales linearly in the semiclassical parameter ε, meaning that a smaller ε will now reduce the error instead of increasing it. The error scales with $\varepsilon(\Delta t)^2$ for the semiclassical splitting using the Y-splitting (see [8]), but the dependency on the timestep Δt can be improved even further by using different splittings which effectively corresponds to higher order coefficient pairs w_T and w_U.

The steps for the semiclassical propagator are shown in algorithm 7 and further details can be found in the original paper [8].

Algorithm 7 Single timestep with Semiclassical propagator

procedure SEMICLASSICAL.PROPAGATE(Δt)

$M_{split} := \lceil 1 + \frac{\sqrt{\Delta t}}{\varepsilon^{3/4}} \rceil$ ▷ Divide Δt into smaller steps

INTSPLIT($\frac{\Delta t}{2}, M_{split}, \{w_T, w_U\}$) ▷ M_{split} split steps with $\mathcal{T} + \mathcal{U}$
STEPW(Δt) ▷ Single step with $\mathcal{W}$
INTSPLIT($\frac{\Delta t}{2}, M_{split}, \{w_T, w_U\}$) ▷ M_{split} split steps with $\mathcal{T} + \mathcal{U}$

end procedure

3.3. Magnus Propagator

As was noted by Magnus in [12], the solution to a differential equation of the form

$$y'(t) = a(t)y(t) \qquad t \geq 0 \tag{11}$$

with initial value $y(0) = y_0$ can be written as

$$y(t) = e^{\sigma(t)} y_0 \tag{12}$$

where $\sigma(t)$ is an infinite sum of iterated integrals and commutators, also known as the Magnus series. The idea behind the Magnus approximation was extensively studied using the Baker-Campbell-Hausdorff (BCH) formula and rooted trees techniques, more details can be found in [3], [2], [10].

In order to approximate the solution from equation 12, one can take only a finite number of terms from the infinite series whereby a truncation error is committed (additional error sources in the Magnus method are the discretization of integrals

and the approximation of matrix exponentials). The Magnus Propagator has several beneficial numerical properties that are described in [10]. In particular, for solutions that evolve within a Lie group, the same holds for the approximate numerical solution calculated through a truncated Magnus series. It was also shown in the same paper that the Magnus series can compete with - and in fact may even outplay - classical schemes like Runge-Kutta or Gauss-Legendre. Although the method is not a symplectic scheme in the usual sense, [10] has shown that in practical applications it conserves the Hamiltonian energy just as well as symplectic integrators.

Moreover, the numerical stability and good performance of the Magnus propagator are not limited to problems where the solution evolves within a Lie Group, but also apply to various problems where this is not the case. Algorithm 8 shows the *MG4* method as presented in [10] and implemented in C++ in the scope of this project.

Algorithm 8 Single timestep with Magnus propagator

procedure MAGNUS.PROPAGATE(Δt)

$h_1 = \frac{3-\sqrt{3}}{6}\Delta t \; h_2 = \frac{2\sqrt{3}}{6}\Delta t$ ▷ Gauss-Legendre coefficients on $[0, \Delta t]$
$M_k = 1 + \lfloor \sqrt{h_k \varepsilon^{-3/8}} \rfloor, \quad k = 1, 2$ ▷ number of timesteps for splitting

INTSPLIT($h_1, M_1, \{w_T, w_U\}$) ▷ advance till $\frac{3-\sqrt{3}}{6}\Delta t$
$\boldsymbol{A}_1 = -\frac{\imath}{\varepsilon^2} \cdot$ BUILDF() ▷ temporarily store interaction matrix
INTSPLIT($h_2, M_2, \{w_T, w_U\}$) ▷ advance till $\frac{3+\sqrt{3}}{6}\Delta t$
$\boldsymbol{A}_2 = -\frac{\imath}{\varepsilon^2} \cdot$ BUILDF() ▷ temporarily store interaction matrix
$\boldsymbol{\Sigma} = \frac{1}{2}\Delta t(\boldsymbol{A}_1 + \boldsymbol{A}_2) + \frac{\sqrt{3}}{12}(\Delta t)^2(\boldsymbol{A}_2 \cdot \boldsymbol{A}_1 - \boldsymbol{A}_1 \cdot \boldsymbol{A}_2)$ ▷ compute $\sigma(t)$
$\boldsymbol{c} = \exp(\boldsymbol{\Sigma})\,\boldsymbol{c}$ ▷ update coefficients
INTSPLIT($h_1, M_1, \{w_T, w_U\}$) ▷ advance till $\frac{6}{6}\Delta t$

end procedure

3.4. McLachlan Propagators

McLachlan (see [13]) has investigated various symplectic schemes for computing the effect of operators of the form $\mathcal{X} = \mathcal{A} + \epsilon\mathcal{B}$ where $\mathcal{A}$ describes an exactly solvable system and $\mathcal{B}$ is a perturbation. The factor ϵ is a small parameter that indicates the limited influence of $\mathcal{B}$, not to be confused with the semiclassical parameter ε.

The *McL* integration schemes are characterised by pairs of weighing coefficients a_i and b_i that allow to represent the exponential $e^{\mathcal{X}}$ as a product

$$e^{\mathcal{X}} = \prod_i e^{b_i t \epsilon \mathcal{B}} e^{a_i t \mathcal{A}} \,. \tag{13}$$

McLachlan also goes through the process of deriving the coefficients for the following schemes

McL(2s,2)	with error of order $\epsilon(\Delta t)^{2s} + \epsilon^2(\Delta t)^2$
McL(6,4)	with error of order $\epsilon(\Delta t)^6 + \epsilon^2(\Delta t)^4$
McL(8,4)	with error of order $\epsilon(\Delta t)^8 + \epsilon^2(\Delta t)^4$.

Note that the error depends on two small parameters, the timestep Δt and the parameter ϵ that quantifies the influence of $\mathcal{B}$.

In our case, $\mathcal{X} = \mathcal{H} = \mathcal{T} + \mathcal{U} + \mathcal{W}$, the computationally expensive operator $\mathcal{W}$ takes the role of $\epsilon\mathcal{B}$, while the remaining Hamiltonian $\mathcal{T} + \mathcal{U}$ corresponds to $\mathcal{A}$. Our preferred method will therefore be of the form $\mathcal{ABA}$ because such a scheme minimizes the number of steps with $\mathcal{B} = \mathcal{W}$.

The source code of the project contains the C++ implementation of the propagators *McL(4,2)* and *McL(8,4)* that require only two respectively five evaluations of $\mathcal{W}$ per step.

3.5. Processing Propagators

The idea of processing propagators is to carry out the time evolution with a Hamiltonian that is slightly perturbed from the original one, which can be achieved by applying a pre and post processing step. The exponential of the Hamiltonian can be re-formulated as

$$e^{-\Delta t\mathcal{H}(\Delta t)} = e^P e^{-\Delta t K} e^{-P} \tag{14}$$

where the pre and post processing steps are applied via the multiplication with matrices e^P and e^{-P} (also referred to as *processors*). While the processors only need to be applied once at the very beginning and end of the propagation (in order to return to the original Hamiltonian), the multiplication with the *kernel* $e^{-\Delta t K}$ is repeated M_{tot} times, once for every timestep.
As a consequence, one wants to choose the processor e^P in such a way that the evaluation of the kernel $e^{-\Delta t K}$ is as simple as possible in order to minimize the work associated with the computation of the kernel.

Unlike most commonly used integration schemes, the family of processing propagators are symplectic integrators. They are not suited for every kind of Hamiltonian, and the work of Blanes, Casas and Ros in [1] gives a method for finding the necessary conditions that need to be satisfied in order for processing methods to be applicable.

Algorithm 10 presents the *Pre764* processing method as derived in [1]. The coefficient arrays α and β (both of length $k = 4$) are used for propagating the wave packet with StepW and IntSplit in the Propagate method, while the coefficient arrays Y and Z (both of length $v = 6$) are used in PrePropagate and PostPropagate to transform the wave packet with the processor.

Algorithm 9 Single timestep with McL42 propagator

procedure McL42.Propagate(Δt)

$M = \lceil \left(\Delta t^4/\varepsilon^3\right)^{\frac{1}{8}} \rceil$ ▷ compute number of substeps
intSplit($A_0\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_0\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_1\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_1\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_2\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$

end procedure

---

procedure McL84.Propagate(Δt)

$M = \lceil \left(\Delta t^4/\varepsilon^3\right)^{\frac{1}{8}} \rceil$ ▷ compute number of substeps
intSplit($A_0\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_0\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_1\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_1\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_2\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_2\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_3\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_3\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_4\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$
stepW($B_4\Delta t$) ▷ $\mathcal{W}=B$
intSplit($A_5\Delta t, M, \{w_T, w_U\}$) ▷ $\mathcal{T}+\mathcal{U}=A$

end procedure

Algorithm 10 Single timestep with Pre764 propagator

procedure PRE764.PREPROPAGATE(Δt)

$M_{pre} = 1 + \left\lfloor \sqrt{\Delta t \varepsilon^{-\frac{3}{4}}} \right\rfloor$ ▷ compute number of time steps
for $j = 1, ..., v$ **do** ▷ $v = 6$ for Pre(7,6,4)
  INTSPLIT($-Z_j \Delta t, M_{pre}, \{w_T, w_U\}$) ▷ M_{pre} alternating steps with $\mathcal{T}$ and $\mathcal{U}$
  STEPW($u[\Pi, \boldsymbol{c}], -Y_j \Delta t$) ▷ single step with $\mathcal{W}$
end for

end procedure

---

procedure PRE764.PROPAGATE(Δt)

$M = 1 + \left\lfloor \sqrt{\Delta t \varepsilon^{-\frac{3}{4}}} \right\rfloor$ ▷ compute number of time steps
for $j = 1, ..., k$ **do** ▷ $k = 4$ for Pre(7,6,4)
  STEPW($\alpha_j \Delta t$) ▷ single step with $\mathcal{W}$
  INTSPLIT($\beta_j \Delta t, M, \{w_T, w_U\}$) ▷ M alternating steps with $\mathcal{T}$ and $\mathcal{U}$
end for

end procedure

---

procedure PRE764.POSTPROPAGATE(Δt)

$M_{post} = 1 + \left\lfloor \sqrt{\Delta t \varepsilon^{-\frac{3}{4}}} \right\rfloor$ ▷ compute number of time steps
for $j = v, ..., 1$ **do** ▷ $v = 6$ for Pre(7,6,4)
  STEPW($Y_j \Delta t$) ▷ single step with $\mathcal{W}$
  INTSPLIT($Z_j \Delta t, M_{post}, \{w_T, w_U\}$) ▷ M_{post} alternating steps with $\mathcal{T}$ and $\mathcal{U}$
end for

end procedure

4. Implementation in C++

This section will explain the most important design decisions and technical details on which the implementation of the presented algorithms is based. While the first priority was clearly to make the code efficient and reduce computation time, another aim was also to create simple and readable code interfaces that make it easy to use and re-use components and extend the existing functionality.
Even though these two requirements often seem to mutually exclude each other, in this case it was possible to meet both targets. In fact, it is quite remarkable how similar the C++ implementations of the individual propagators look to their pseudo-algorithms as presented in section 3.

Also, it should be mentioned here that some of the most important bottlenecks of the code were not strictly related to the implementation of the propagators, such as for example the building of the $\boldsymbol{F}$-matrix through numerical quadrature of $\langle\varphi_k|W|\varphi_l\rangle$. In fact, the current implementation of the BUILD_MATRIX routine for computing the inner product requires to always compute a full $N \times N$ sub-matrix, even if just one single entry is needed, which is a very severe inefficiency if the number of levels $N > 1$.

However, the aim of the work presented here was to provide a fast implementation of the propagators by re-using existing functions, therefore possible performance enhancements through changing of the existing code was out of the scope of this project and could be targeted in future work. Consequently, providing an efficient implementation for time propagation mainly translates into finding an efficient way for calling existing functions with minimal overhead and by making use of compiler optimizations where possible.

First, subsection 4.1 outlines how C++ features were used in order to achieve class inheritance and polymorphism witout any additional runtime cost. Secondly, in subsection 4.2, it will be discussed how the EVOLVE method (see algorithm 1) was implemented in order to provide a minimal working interface to the user while still preserving all the flexibility that may be required for intermediate measurements or other interaction. Finally, subsection 4.3 dives into the technical details of achieving inlined, highly optimized function calls for the INTSPLIT method (see algorithm 4) that is used intensively in all the presented propagators.

Some details about the code were not found worthy to be mentioned here, but more information may be found in the Doxygen documentation.

The Hagedorn Propagator was already implemented in C++ as part of another project on the waveblocks library [5]. Many components of that code were re-used for the present implementation, but they were fundamentally re-organized into a new structure.

4.1. Static Polymorphism through CRTP and enable_if

In the current software design, all propagators from section 3 inherit from a common base class PROPAGATOR, see the UML diagram 1. The strategy for the implementation was

the following: Provide an efficient and generic implementation of all the building blocks from section 2 in a common base class (PROPAGATOR) and then let the derived class decide which combination of building blocks to use for its own, specific implementation of the propagation methods. This idea is essentially an application of the Strategy Pattern as introduced by Gamma et al. in [7], one of the most important references for software design patterns.

A popular C++ design pattern for achieving static polymorphism is the *Curiously Recurring Template Pattern*, *CRTP* for short, which is described in more detail in [11] and in earlier work on the WaveBlocks project [14].

While CRTP is an excellent candidate for providing static polymorphism for derived classes, there was also the need for some polymorphism in the Propagator base class itself, since some of the building blocks need to have different implementations based on the dimensionality or homogeneity of the wave packet. Providing a partially specialized class for every possible combination of parameters is often tedious and unnecessarily bloats the code. Therefore, the C++11 ENABLE_IF keyword was used, which is a handy C++ metafunction from the TYPE_TRAITS header. It allows to define multiple versions of a function and conditionally (de)activate them based on a compile-time boolean value. If the boolean is true, the function is enabled. Otherwise it is omitted. In the code, such boolean values are indicating whether the number of levels N is equal or greater to one or to distinguish between functions for homogeneous or inhomogeneous wave functions. The presented use of ENABLE_IF effectively provides a form of compile time polymorphism.

4.2. Evolve and callback function

The simple EVOLVE method as described in algorithm 1 in the section about propagator building blocks provides a handy wrapper for calling the PREPROCESSING, PROCESSING and POSTPROCESSING methods in sequence and dividing the propagation time into smaller steps.
However, one may still want to interact with the wave packet during the simulation, for example for writing the wave packet norm to file every 100 steps, throwing errors when energy conservation is violated or to carry out any other task that requires intermediate simulation results. In order not to lose any flexibility in this regard, an optional callback-function argument was added to the EVOLVE function. The callback function is passed as a lambda function taking two arguments, the current simulation time T and the index of the current timestep M. If no callback function is passed, it defaults to an empty function.

Furthermore, since the wave packet is passed to the PROPAGATOR class *by reference*, it is of course available in the callback function as well and does not need to be passed as an argument explicitly (provided that it is captured in the angular brackets [] of the lambda function).
Disclaimer if the value of the wave packet is (intentionally or unintentionally) changed in the callback function, the further results of the simulation will of course also be

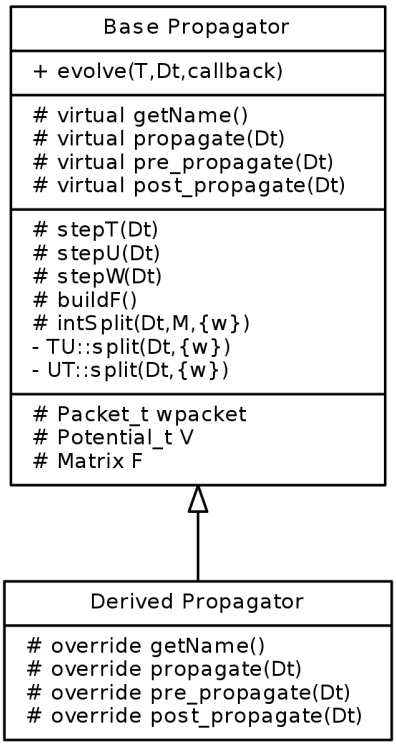

Figure 1: Simplified UML diagram for the propagator implementation. The derived propagator provides implementations for PrePropagate, Propagate and PostPropagate which are called from the public Evolve function. Some functions and function parameters are omitted for better readability of the graph.

affected. Also, care must be taken when measuring energies of packets that are being propagated with propagators such as *Pre764* which apply a pre-processor to the packet before starting the simulation. In order to get the "physically correct" wave packet, the corresponding post-processor needs to be applied before making any meaningful measurement.

The callback function is called once prior to simulation start and then in every time step until termination.

Finally, a few helper functions were introduced in order to facilitate pretty printing of console output and some information about the currently propagated wave packet is printed at the beginning of the Evolve function. These console outputs can (and should) of course be removed for optimal performance, but in most cases it was found that some feedback about the current status of the simulation is highly valuable.

4.3. IntSplit using alternating templates

The IntSplit function is merely a helper function that alternatingly applies StepT and StepU. Steps with the operators $\mathcal{T}$ and $\mathcal{U}$ are usually much cheaper than steps with $\mathcal{W}$ (which requires numerical integration for approximating the inner product $\langle\varphi_k|W|\varphi_l\rangle$), but are in many cases found to be one of the main soures of error. Therefore the idea of sub-dividing the steps into many smaller intervals - an approach introduced in the semiclassical splitting 3.2 - is applied in many other propagators as well. Consequently, it is desirable to have a generic and efficient implementation available.

The current IntSplit algorithm described in 4 takes three arguments: the size of the time interval to be covered Δt, the number M_{split} of subintervals of size $\delta t = \Delta t / M_{split}$ in which the initial interval should be divided, and a pair of coefficient lists $\{w_T, w_U\}$ denoting the weights with which the single steps of size δt should be scaled. In the simplest case, the lists w_T and w_U are both only of length one (Lie-Trotter), but higher order splitting strategies can have much larger coefficient sets. In general, it must hold that $\text{length}(w_T) = \text{length}(w_U)$ (for TU..TU schemes) or $\text{length}(w_T) = \text{length}(w_U) + 1$ (for TU..UT schemes).

The obvious implementation of using a dynamic container (like a C++ vector) for the coefficients and a simple *for loop* over all its entries brings a two (minimal) drawbacks: First, the overhead of the loop itself and second, the fact that the number of coefficients is not known at compile time and thus no optimizations can be done. Given that the functions StepT and StepU themselves are very cheap (just simple updates of parameters $\mathbf{\Pi}$ and S), it is preferable to remove any kind of additional cost, even if only marginal.

By making use of C++ arrays and templates, one can remove the issues mentioned above.
Firstly, by replacing C++ vectors (dynamically allocated) through C++ arrays (statically allocated), the container size is known at compile time. For convenience, a new struct SplitCoefs was introduced that holds a pair of such arrays. Secondly, and with the information about container size readily available at compile time, one can then implement two sets of templates TU and UT that execute a step with operator $\mathcal{T}$ or $\mathcal{U}$ respectively and then mutually invoke each other, until both arrays of coefficients are used up. As a termination condition, when all coefficients have been used, a partially specialized instance of the templates will be called and interrupt the mutual invocation. One technical subtlety of the presented approach is, that the C++ language does not allow partially specialized function templates. This can, however, be overcome quite easily by declaring TU and UT as classes with a static function. In contrast to function templates, class templates can be partially specialized. By using static member functions, the actual class objects do not need to be created and all the function calls will be resolved at compile time[2]. All templates, no loops, and the resulting sequential code can then be further optimized by the compiler.

[2] One limitation is the maximal template recursion depth. The default value on gcc compiler is 900 and can be increased if required. Typical template nesting depths for coefficient pairs should lie significantly below 100.

Some minimal code snippets explaining the working principle of IntSplit are shown in Figure 2. In order to prevent the inappropriate use of the IntSplit function through incompatible coefficient lists w_T and w_U, their sizes are checked in the code at compile time via static assertions.

The fact that $\mathcal{W}$ is significantly more costly to compute than its counterparts $\mathcal{T}$ and $\mathcal{U}$ also justifies the approach of using templates that are hard-coded to call functions StepT and StepU, as these are the only logical candidates for IntSplit.

```
// Specialized Template
// N_T == S_T
// N_U == S_U
template <int S_T, int S_U>
struct TU<S_T,S_U,S_T,S_U> {
  static void split(
     array<float,S_T>& w_T,
     array<float,S_U>& w_U,
     float dt)
  { /* terminate */ }
};

template <int N_T, int N_U,
           int S_T, int S_U>
struct TU {
  static void split(
    array<float,S_T>& w_T,
    array<float,S_U>& w_U,
    float dt)
    /* T */
    stepT(w_T(N_T)*dt);
    /* UT... */
    UT<N_U,N_T+1,S_U,S_T>
       ::split(w_U,w_T,dt);
  }
};
```

```
// Specialized Template
// N_U == S_U
// N_T == S_T
template <int S_U, int S_T>
struct UT<S_U,S_T,S_U,S_T> {
  static void split(
     array<float,S_U>& w_U,
     array<float,S_T>& w_T,
     float dt)
  { /* terminate */ }
};

template <int N_U, int N_T,
           int S_U, int S_T>
struct UT {
  static void split(
    array<float,S_U>& w_U,
    array<float,S_T>& w_T,
    float dt)
    /* U */
    stepU(w_U(N_U)*dt);
    /* TU... */
    TU<N_T,N_U+1,S_T,S_U>
       ::split(w_T,w_U,dt);
  }
};
```

Figure 2: Simplified C++ snippets for outlining the template invocation sequence in the IntSplit function. The template parameters S_T and S_U denote the total size of the coefficient arrays w_T and w_U while the N_T and N_U are the indices of the current coefficient to be applied. The TU::split and UT::split methods first apply their own step and then invoke each other by gradually incrementing the incices N_T and N_U.
When the conditions N_T == S_T and N_U == S_U are satisfied, the partially specialized templates become active and the recursion is terminated.

5. Results

The newly developed code was used to carry out some numerical experiments which are presented in this section. Aspects that need to be checked are the correctness of the propagators from a quantum mechanical point of view (subsection 5.1), convergence of the methods (subsection 5.2) and benchmarking of the compute time (subsection 5.3).

The potentials used for the numerical experiments were the simple harmonic potential and the Henon-Heiles potential. In both cases, the basis size $|\mathfrak{K}| = 4$ was used.

The generalized harmonic potential

$$V(\boldsymbol{x}) = \frac{1}{2}\sum_{i=1}^{D} x_i^2 \tag{15}$$

was used with initial values $\boldsymbol{q} = (1, \ldots, 1)^T$ and $\boldsymbol{q} = (1, \ldots, 1, -1, \ldots, -1)$ that were normalized to a L_2 norm of 0.5 and 0.8 respectively. $\boldsymbol{Q}$ was the identity matrix and $\boldsymbol{P} = \imath\boldsymbol{Q}^{-1}$.

The two dimensional Henon-Heiles potential is given by

$$V(\boldsymbol{x}) = \frac{1}{2}\left(x_1^2 + x_2^2\right) + \sigma_* \left(x_1^2 x_2 - \frac{1}{3}x_2^3\right) \tag{16}$$

where σ_* is a mixing coefficient that was chosen to be equal to 0.2. The initial values for this potential were $\boldsymbol{q} = (1.8, 0)^T$, $\boldsymbol{p} = (0, 1.2)^T$, $\boldsymbol{Q} = \sqrt{2} \cdot \mathrm{diag}(\sqrt{0.56}, \sqrt{0.24})$ and $\boldsymbol{P} = \imath\boldsymbol{Q}^{-1}$)

5.1. Physical Correctness

In order to analyze the physical correctness of the implemented propagators, a two dimensional wave packet was propagated in the Harmonic potential and in the Henon-Heiles potential presented above with each of the presented propagators for a time $T = 10$ with timestep $\Delta t = 0.01$. All propagators used the LT splitting coefficients for the IntSplit method.

Figure 3 shows the time evolution of energy and the energy drift for the semiclassical propagator. The corresponding energy plots for the remaining propagators can be found in the appendix to the report.
As is clearly shown by the graphs, the propagators conserve the total energy (apart from very small oscillations).

5.2. Effect of Stepsize

A convergence analysis was carried out in which the error of each propagator was recorded while reducing the stepsize Δt. A Semiclassical propagator with *KL10* splitting

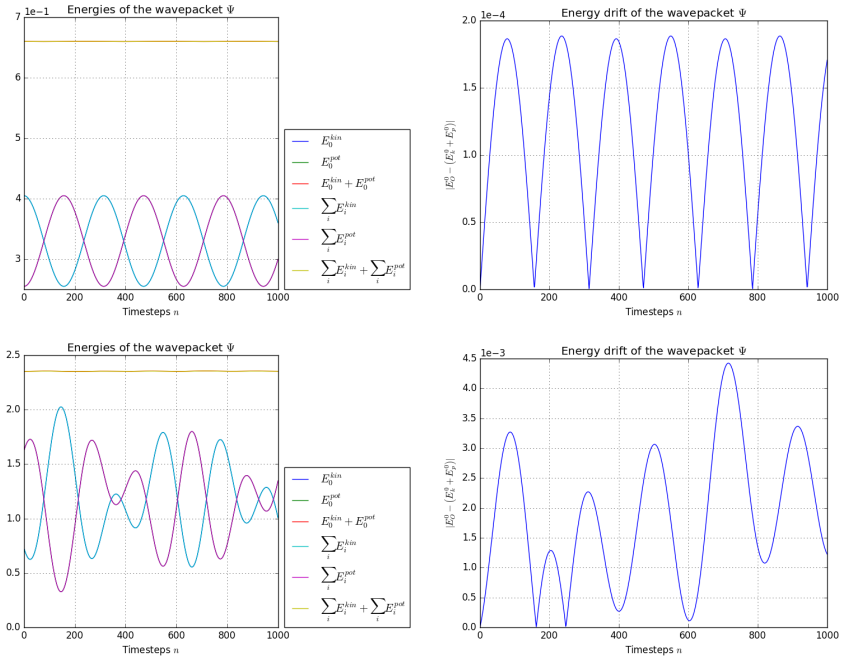

Figure 3: Energy evolution and drift for a 2D wave packet propagated with the Semiclassical propagator in a harmonic potential (top) and a Henon-Heiles potential (bottom). ($T = 10$, $\Delta t = 0.01$, $|\mathfrak{K}| = 4$, LT splitting)

coefficients and step size $\Delta t = 0.0001$ was used to propagate the wave packet to a final time $T = 10$ and create a reference solution.

The error between two wave packets was computed by evaluating them on a grid with 1000 grid points and taking the L_2 norm of the differences. Figure 4 shows the error for different step sizes Δt. It is quite surprising how the Hagedorn Propagator has higher precision than the other schemes in the case of LT splitting. Only for higher order splitting coefficients like $Y4$ the advantages of IntSplit start paying off and the other propagators show their advantages over the simplistic Hagedorn Propagator.

5.3. Benchmark

In order to benchmark the code, three timing experiments were carried out: a simple comparison of runtimes for different propagators, an investigation of the computational cost associated with the usage of high order splitting schemes, and an analysis of the

compute time as a function of the dimension D. In all cases, the harmonic potential introduced at the beginning of this section was used.

The runtime analysis was carried out on a Linux (Kernel 4.8.7, Fedora 24 Workstation) Quad-Core machine with an Intel Core i5-3210M Processor and 4GB of RAM. The C++ code was compiled with the GNU compiler, version 6.2.1, and using the *-Ofast* optimization flag.

5.3.1. Comparison of propagators

A quick comparison of timings for the different propagators is shown in table 1, the same setup as for the energy analysis was used (2D wave packet, $T = 10$, step size $\Delta t = 0.01$, LT time splitting coefficients). The Magnus Propagator *MG4* is cheaper than expected, as it involves two evaluations of $\langle \varphi_k | W | \varphi_l \rangle$ per time step but is only slightly slower than the Semiclassical operator with one evaluation of the inner product. The slowest propagator is *McL84* which involves five propagations with $\mathcal{W}$ per step.

Propagator	**Timing [s]**
Hagedorn	1.06
Semiclassical	1.14
MG4	1.23
McL42	1.39
McL84	2.08
Pre764	1.63

Table 1: Runtimes for propagating a 2D wave packet to a time $T = 10$ with timestep $\Delta t = 0.01$ with different propagators in C++. All the timings in the table were measured by taking the average of 10 independent runs. The standard deviation of these measurements was below 1%.

5.3.2. Splitting Schemes

As mentioned previously, the splitting coefficients $\{w_T, w_U\}$ that are used as weights on the timestep δt in the IntSplit method can have vastly different complexity, ranging from the *Lie-Trotter* coefficients (one coefficient for propagation with $\mathcal{T}$, one for propagation with $\mathcal{U}$) up to the *KL10* coefficients (34 coefficients for each operator). Higher order schemes are usually preferred in terms of accuracy, but they come at the price of longer computation time.

Therefore, a numerical experiment was carried out in order to analyze how much computational time is consumed for coefficient pairs of different sizes. In order to create a reference solution, the Semiclassical Propagator was used to propagate a two dimensional wave packet in a harmonic potential over a time of $T = 400$ and with stepsize $\Delta t = 0.01$. The same simulation was carried out for coefficient pairs of various different sizes and were run in Python as well as C++. The results are listed in table 2 and plotted in figure 5.

Looking at the Python timings only, the measurements suggest that in the case of the *KL10* coefficient set, at least 80% of the total runtime is spend in the IntSplit function (since larger coefficient sets $\{w_T, w_U\}$ only affect the number of steps with operators $\mathcal{T}$ and $\mathcal{U}$, but not with operator $\mathcal{W}$).

When bringing the timings in context with the C++ code, a comparison of absolute runtimes is of course not fair since C++ is intrinsically faster than a scripting language like Python and the C++ version has been optimized for speed in many different ways. However, a very significant result in the context of the work on Propagators is how the speedup increases for larger coefficient pairs. Using the *KL10* coefficient set instead of a simple *LT* set, the Python code takes more than six times longer. The same comparison on C++ code shows that the code takes only 30% longer for the large coefficient set than it takes for the minimal coefficient set. This is a very encouraging result as it means that high order splitting coefficients come at almost no extra cost.

Splitting	**No. of coefs**	**Python timing [s]**	**C++ timing [s]**	**Speedup**
LT	1	174.6	5.766	**30.3**
S2	2	202.9	5.706	**35.6**
Y4	4	258.6	5.861	**44.1**
PRKS6	7	341.0	5.995	**56.9**
Y61	8	366.0	6.004	**61.0**
KL6	10	421.4	6.124	**68.8**
BM63	15	557.1	6.467	**86.2**
KL8	18	634.8	6.637	**95.6**
KL10	34	1067.2	7.457	**143.1**

Table 2: Table comparing the computation time of Python code vs. C++ code for different sizes of the splitting coefficients $\{w_T, w_U\}$. All the timings in the table were measured by taking the average of 10 independent runs. The standard deviation of these measurements was below 1% for the Python timings and below 0.1% for the C++ timings.

5.3.3. Scaling with dimensionality D

The question that was addressed in this benchmark is how the compute time of the code scales with increasing dimension D of the wave packet. Figure 6 shows a clear exponential scaling with the dimension D. With a computation time of under 20 minutes for 100 time steps with the Semiclassical Propagator, dimension $D = 5$ is still quite feasible.

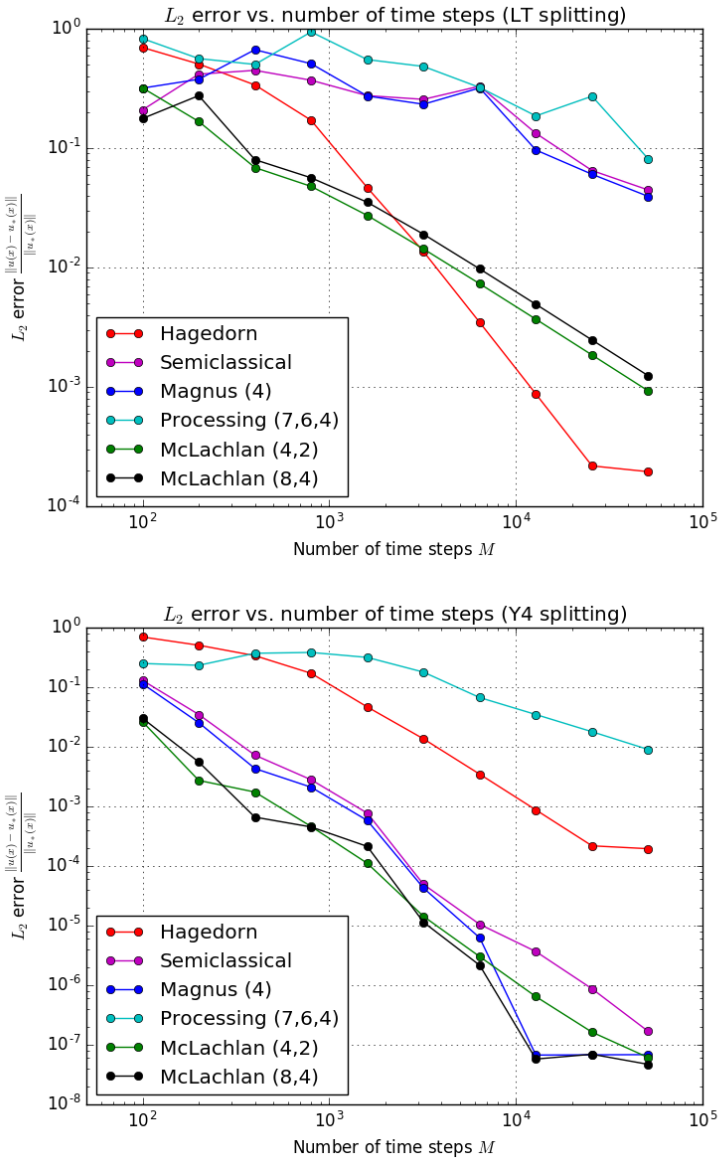

Figure 4: Convergence of the various propagators towards the reference solution for LT splitting (top) and $Y4$ splitting (bottom). The L_2 norm was measured by projecting the wave function on a grid with 1000 nodes in the range $[-1, 1]$ and taking the differences of the current solution and the reference solution at $T = 10$.

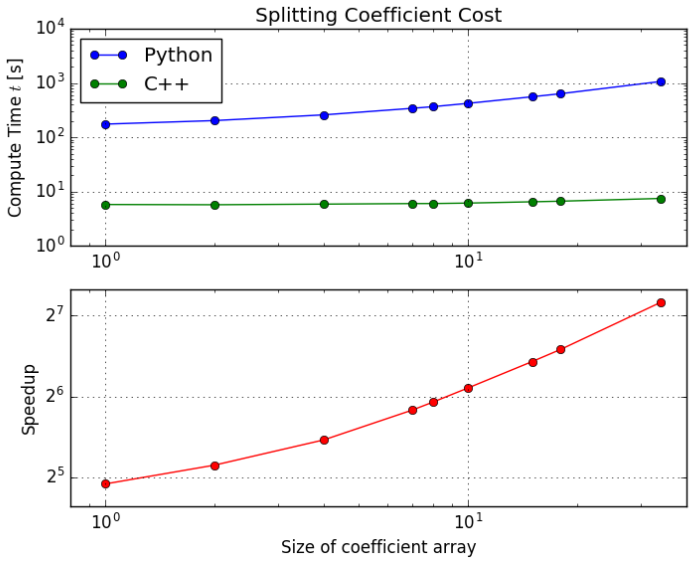

Figure 5: Comparison of computation times for different splitting coefficients $\{w_T, w_U\}$. The absolute timings are shown in the top graph, the speedup on the bottom. It is remarkable that the initial speedup factor is about 30, but grows for larger coefficient sets. The speedup factor for the largest tested coefficient pair *KL10* is over 140.

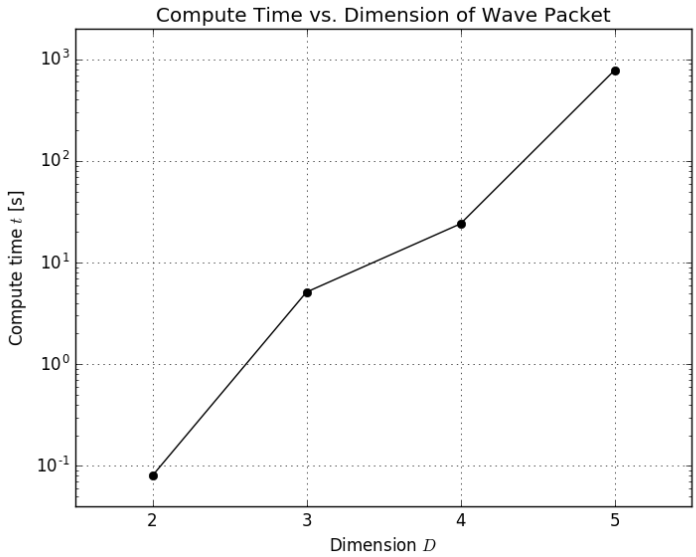

Figure 6: Scaling of the computation time with wave packet dimension D. Measurements done for a wave packet in a generalized harmonic potential, propagated for 100 steps with the Semiclassical Propagator.

6. Conclusion

In this project, a series of time propagators for Hagedorn wave packets was studied, analyzed and implemented. The C++ code, which is the main outcome of the conduced work, is based on a clean software design that makes it very easy to add new Propagators and integrate them into the existing framework. At the same time, as shown by the benchmarking results, the performance of the resulting code is by orders of magnitude better than that of the corresponding Python code. In particular, the additional cost for using higher-order splitting coefficients is very contained thanks to a templated implementation of the IntSplit method which is resolved at compile time and avoids the usage of any loops. Given these performance improvements, more complex quantum mechanical simulations are now possible with the WaveBlocks framework.

A. Energy Evolution and Drift Plots

Hagedorn Propagator

Energy Evolution and Drift

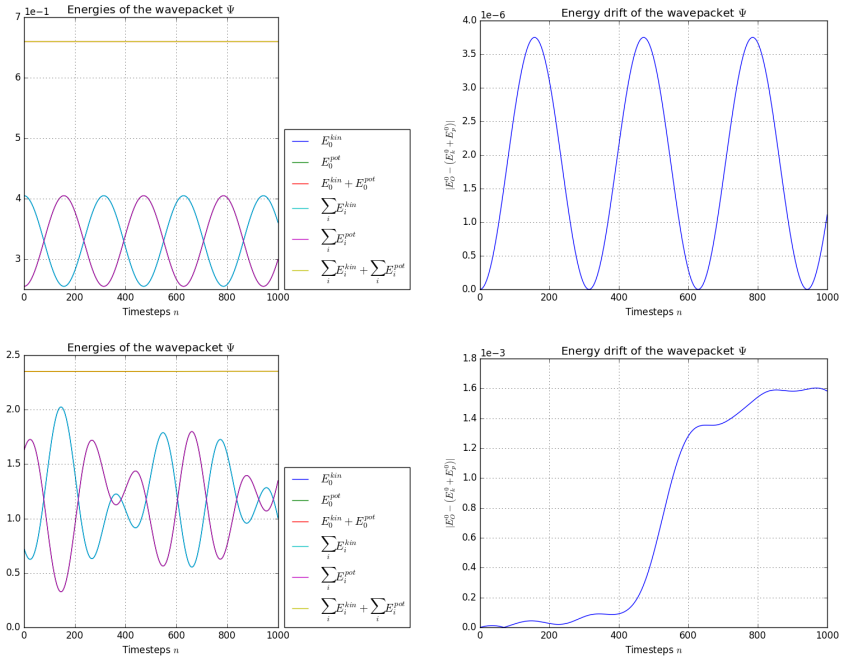

Figure 7: Energy evolution and drift for a 2D wave packet propagated with the Hagedorn propagator in a harmonic potential (top) and a Henon-Heiles potential (bottom). ($T = 10$, $\Delta t = 0.01$, $|\mathfrak{K}| = 4$, LT splitting)

MG4 Propagator

Energy Evolution and Drift

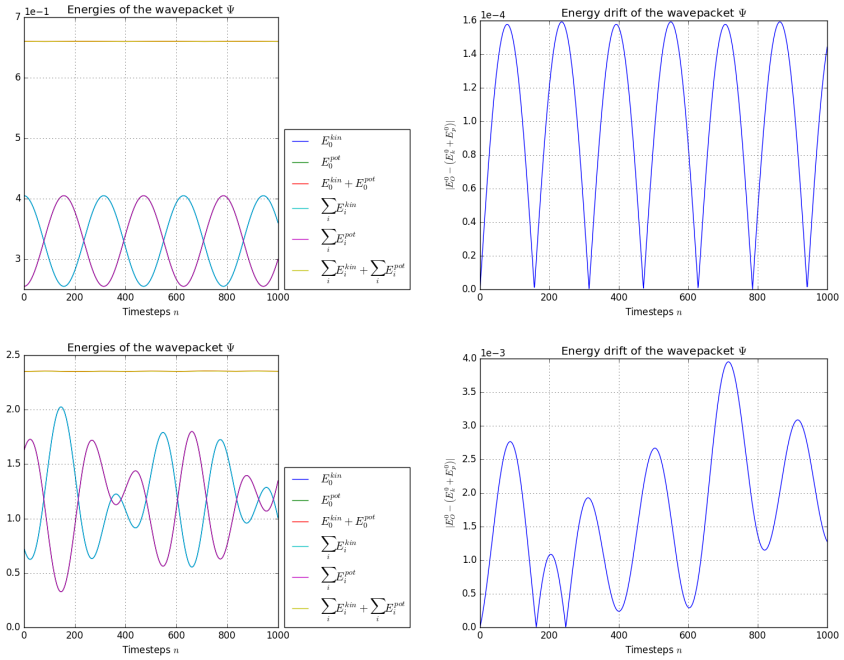

Figure 8: Energy evolution and drift for a 2D wave packet propagated with the MG4 propagator in a harmonic potential (top) and a Henon-Heiles potential (bottom). ($T = 10$, $\Delta t = 0.01$, $|\mathfrak{K}| = 4$, LT splitting)

McL42 Propagator

Energy Evolution and Drift

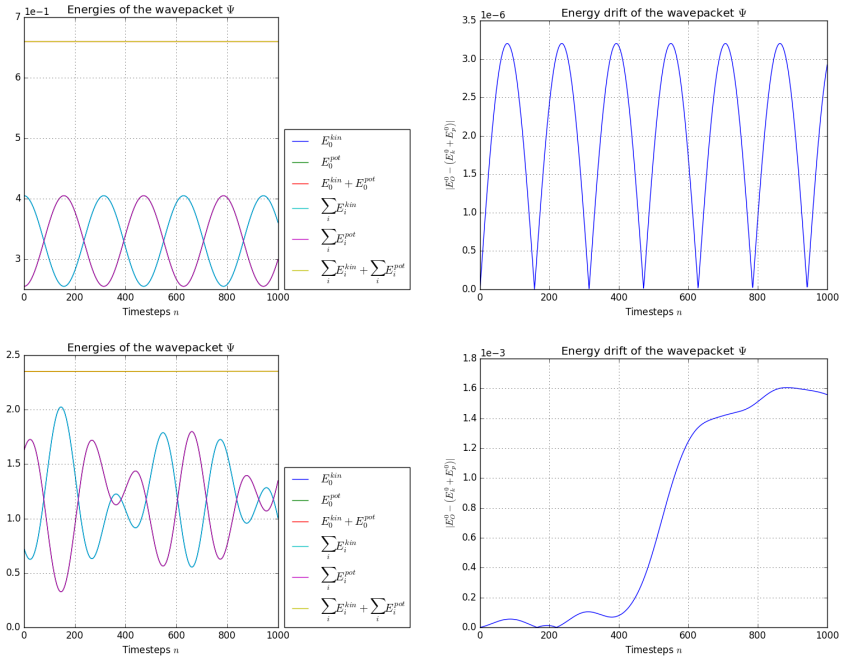

Figure 9: Energy evolution and drift for a 2D wave packet propagated with the McL42 propagator in a harmonic potential (top) and a Henon-Heiles potential (bottom). ($T = 10$, $\Delta t = 0.01$, $|\mathfrak{K}| = 4$, LT splitting)

McL84 Propagator

Energy Evolution and Drift

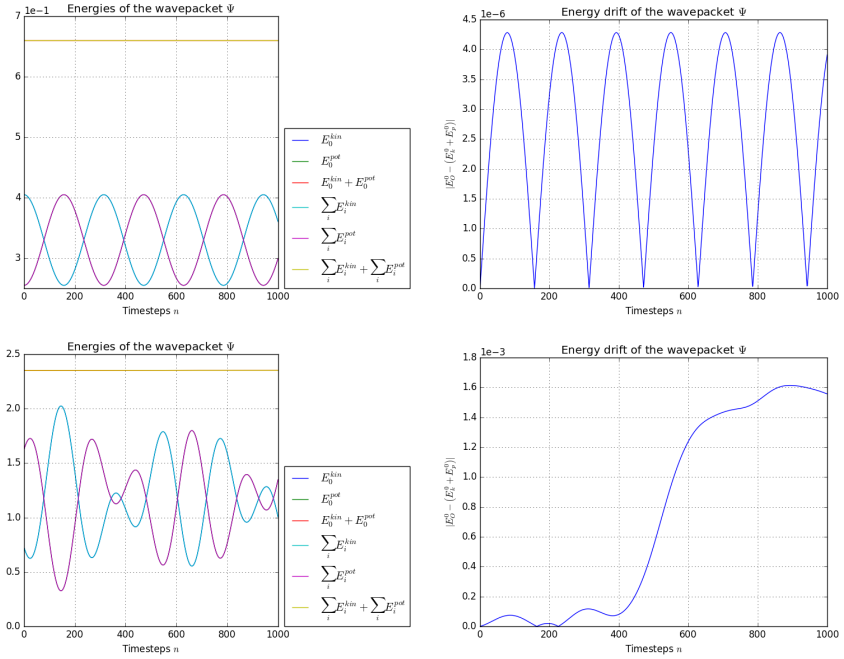

Figure 10: Energy evolution and drift for a 2D wave packet propagated with the McL84 propagator in a harmonic potential (top) and a Henon-Heiles potential (bottom). ($T = 10$, $\Delta t = 0.01$, $|\mathfrak{K}| = 4$, LT splitting)

Pre764 Propagator

Energy Evolution and Drift

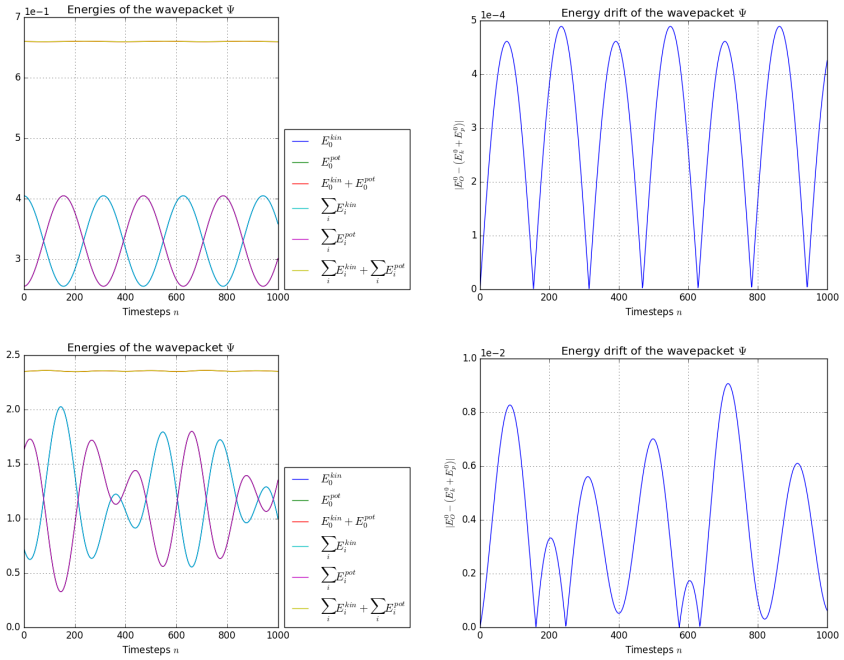

Figure 11: Energy evolution and drift for a 2D wave packet propagated with the Pre764 propagator in a harmonic potential (top) and a Henon-Heiles potential (bottom). ($T = 10$, $\Delta t = 0.01$, $|\mathfrak{K}| = 4$, LT splitting)

References

[1] S. Blanes, F. Casas, and J. Ros. Symplectic integration with processing: A general study. *SIAM Journal on Scientific Computing*, 21(2):711–727, 1999. (Cited on page 15.)

[2] S. Blanes, F. Casas, and J. Ros. Improved high order integrators based on the magnus expansion. *BIT Numerical Mathematics*, 40(3):434–450, 2000. (Cited on page 13.)

[3] S. Blanes and P.C. Moan. Fourth- and sixth-order commutator-free magnus integrators for linear and non-linear dynamical systems. *Applied Numerical Mathematics*, 56(12):1519 – 1537, 2006. (Cited on page 13.)

[4] 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. (Cited on page 6.)

[5] R. Bourquin, M. Bösch, L. Miserez, and B. Vartok. libwaveblocks: C++ library for simulations with semiclassical wavepackets. https://github.com/WaveBlocks/libwaveblocks, 2015, 2016. (Cited on pages 2, 5 and 18.)

[6] Erwan Faou, Vasile Gradinaru, and Christian Lubich. Computing semiclassical quantum dynamics with Hagedorn wavepackets. *SIAM Journal on Scientific Computing*, 31(4):3027–3041, 2009. (Cited on pages 6, 7, 8 and 12.)

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Cited on page 19.)

[8] 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. (Cited on pages 6, 12 and 13.)

[9] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration*. Springer, Berlin, Germany, 2002. (Cited on page 7.)

[10] A. Iserles, A. Marthinsen, and S. P. Nørsett. On the implementation of the method of magnus series for linear differential equations. *BIT Numerical Mathematics*, 39(2):281–304, 1999. (Cited on pages 13 and 14.)

[11] Coplien James O. Curiously recurring template patterns. *C++ Report*, pages 24–27, 1995. http://sites.google.com/a/gertrudandcope.com/info/Publications/InheritedTemplate.pdf. (Cited on page 19.)

[12] Wilhelm Magnus. On the exponential solution of differential equations for a linear operator. *Communications on Pure and Applied Mathematics*, 7(4):649–673, 1954. (Cited on page 13.)

[13] Robert I. McLachlan. Composition methods in the presence of small parameters. *BIT Numerical Mathematics*, 35(2):258–268, 1995. (Cited on page 14.)

[14] Lionel Miserez. Porting waveblocksnd matrix potential functionality to c++. *Libwaveblocks*, 2015. (Cited on page 19.)

[15] Benedek Vartok. Implementation of waveblocksnd's quadrature and inner products in c++. *Libwaveblocks*, 2015. (Cited on page 9.)