Implementation of WaveBlocksND's Quadrature and Inner Products in C++

Term Paper

Author:
Benedek Vartok

Supervisor:
Dr. Vasile Gradinaru

Advisor:
Raoul Bourquin

Seminar for Applied Mathematics
ETH Zurich

Autumn Semester 2015

Contents

1 Introduction

This work is part of an effort to port the WaveBlocksND project [1] from Python to C++, using the Eigen template library for linear algebra. WaveBlocksND provides a framework for quantum-physical simulations, in particular to calculate the time propagation of wavepackets in different potentials. The necessary theory for the calculations and algorithms is given in [2].

In this work, we focus on implementing quadrature and inner product calculations for the wavepackets, which is a prerequisite for the time-propagation algorithms.

This report will show formulas and rules used for numerical quadrature, the code structure for building inner products, as well as describing the additional complexity arising from multi-component wavepackets.

Because one of the main reasons of porting the code to C++ was to increase performance, run-time measurements will also show the achieved improvements.

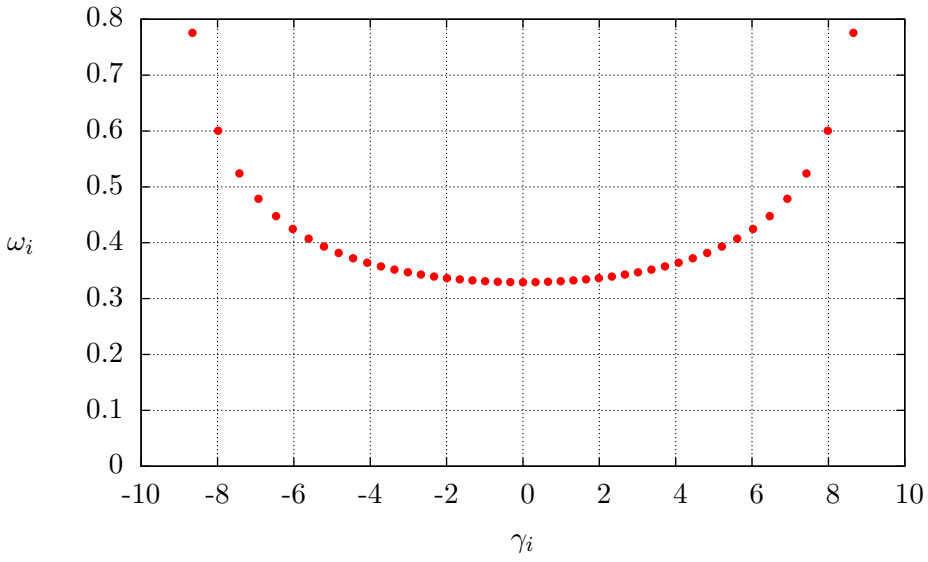

Figure 1: 45-point Gauss-Hermite nodes and weights

2 Quadrature

The central building block for evaluating inner products is the calculation of integrals between wavepackets. This is done numerically with a quadrature rule, of which there are many different kinds.

2.1 Gauss-Hermite Quadrature

The basic integration algorithm used in this project is the one-dimensional Gauss-Hermite quadrature. Like related rules, this method works by evaluating the integrand function at a defined set of nodes γ_i, the values of which are summed up with specific weights ω_i:

$$\int_{-\infty}^{\infty} g(x)\,dx \approx \sum_{i=1}^{n} f(\gamma_i)\omega_i \tag{1}$$

In its basic form, the Gauss-Hermite rule works on special integrals of the following form:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)\,dx \tag{2}$$

However, as we require the quadrature to work on general functions, transformations are done to the usual weights and nodes which is described in [2]. The result is a rule in the form of (1). Figure 1 shows an example of the node distribution of a Gauss-Hermite rule.

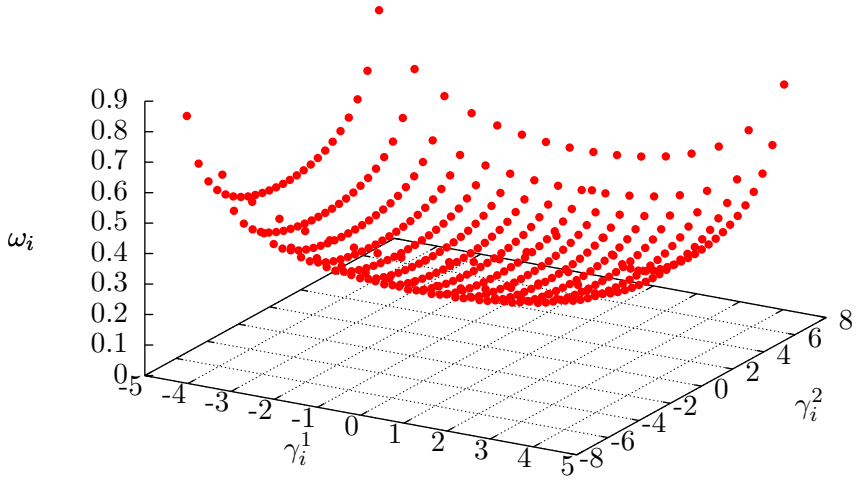

Figure 2: Tensor-product built from 16- and 25-point Gauss-Hermite rules

In the implementation of this rule, an auxiliary Python script was made to automatically generate C++-code with hardcoded tables of nodes and weights for different orders.

The source code can be found under `scripts/create_tables.py` and `waveblocks/gauss_hermite_qr.hpp`.

2.2 Tensor-Product Quadrature

The wavepackets that are processed by WaveBlocks are generally multi-dimensional. To handle this, tensor products of D one-dimensional quadrature rules can be built up.

Denoting the i-th node-weight-pair of the d-th scalar quadrature rule $(\gamma_i^d,\ \omega_i^d)$, the resulting pair at the multi-index $j = (j_1, \ldots, j_d)$ for the resulting tensor quadrature rule is:

$$(\gamma_j,\ \omega_j) = \left(\begin{pmatrix} \gamma_{j_1}^1 \\ \vdots \\ \gamma_{j_d}^d \end{pmatrix},\ \prod_{d=1}^{D} \omega_{j_d}^d \right) \tag{3}$$

A two-dimensional quadrature rule is shown in figure 2.

The implementation resides in `waveblocks/tensor_product_qr.hpp`.

2.3 Result Caching

As the node positions and weight values are tabulated with an external script for Gauss-Hermite Quadrature (2.1), the code for looking up and returning those numbers is simple and fast.

However, in the case of Tensor-Product Quadrature (2.2), some calculations must be performed to yield these values. We wish to avoid recalculating node and weight vectors when the same quadrature rule is used many times, as is done in the case of wavepacket propagation over many iterations.

For this reason, the results are only calculated once for a given Tensor-Product rule, stored and re-used automatically on subsequent queries. The quadrature functions have been implemented as `static` methods, with the rule parameters given as template parameters. Therefore, the caching is done in `static` fields.

This has the advantage that the rule only has to be calculated once in the entire run of the program as long as the quadrature rule is the same, even if it is invoked in different scopes, without having to pass down an object explicitly that holds a state. On the other hand, no clean-up can be done automatically, so a method must be called manually to free the memory used by the cache.

3 Inner Products

In order to calculate wavepacket norms, various energies and other observables, it is necessary to evaluate the braket

$$M = \langle \Psi | F | \Psi' \rangle, \tag{4}$$

where Ψ and Ψ' are wavepackets and F specifies the operator.

For the mathematical derivation of the braket evaluation algorithms, again refer to [2].

3.1 Scalar Inner Products

In the case of scalar wavepackets Φ and Φ' the braket is written as

$$M = \langle \Phi | f | \Phi' \rangle. \tag{5}$$

The operator f is given as a function parameter of the form

$$f(x, q) : \mathbb{C}^{D \times R} \times \mathbb{R}^D \to \mathbb{C}^R \tag{6}$$

mapping D-dimensional nodal points x and position q to the operator's values.

Here, R is the order of the quadrature. Instead of calling the operator function for the different nodes separately, all points are given to the function at once to make efficient vector calculations in its definition possible.

The inner product calculation implementation provides two different methods, `build_matrix` and `quadrature`. The first returns a $|\mathfrak{K}| \times |\mathfrak{K}'|$-sized matrix, one entry for each pair of basis functions of Φ and Φ', where $\mathfrak{K}$ is the basis shape size. `quadrature` reduces the matrix to a scalar by multiplying in the coefficient vectors c and c' of Φ and Φ', returning $c^H M c$.

The "inhomogeneous" evaluation code for different wavepackets Φ and Φ' can be found in `waveblocks/inhomogeneous_inner_product.hpp`. If the two wavepackets are the equivalent, $\Phi = \Phi'$, a more efficient "homogeneous" algorithm can be used which is implemented in `waveblocks/homogeneous_inner_product.hpp`

3.2 Multi-component Inner Products

In the more general case the wavepackets can have multiple components:

$$\Psi = \begin{pmatrix} \Phi_1 \\ \vdots \\ \Phi_N \end{pmatrix} \tag{7}$$

$$M = \langle \Psi | F | \Psi' \rangle = \left\langle \begin{pmatrix} \Phi_1 \\ \vdots \\ \Phi_N \end{pmatrix} \middle| \begin{pmatrix} \ddots & \vdots & \iddots \\ \cdots & f_{i,j} & \cdots \\ \iddots & \vdots & \ddots \end{pmatrix} \middle| \begin{pmatrix} \Phi'_1 \\ \vdots \\ \Phi'_N \end{pmatrix} \right\rangle \tag{8}$$

The calculation of these brakets builds on top of the inhomogeneous scalar evaluation code, calling it for every pair of wavepackets (Φ_i, Φ'_j) and corresponding function $f_{i,j}$.

The source code for this algorithm is found in `waveblocks/vector_inner_product.hpp`.

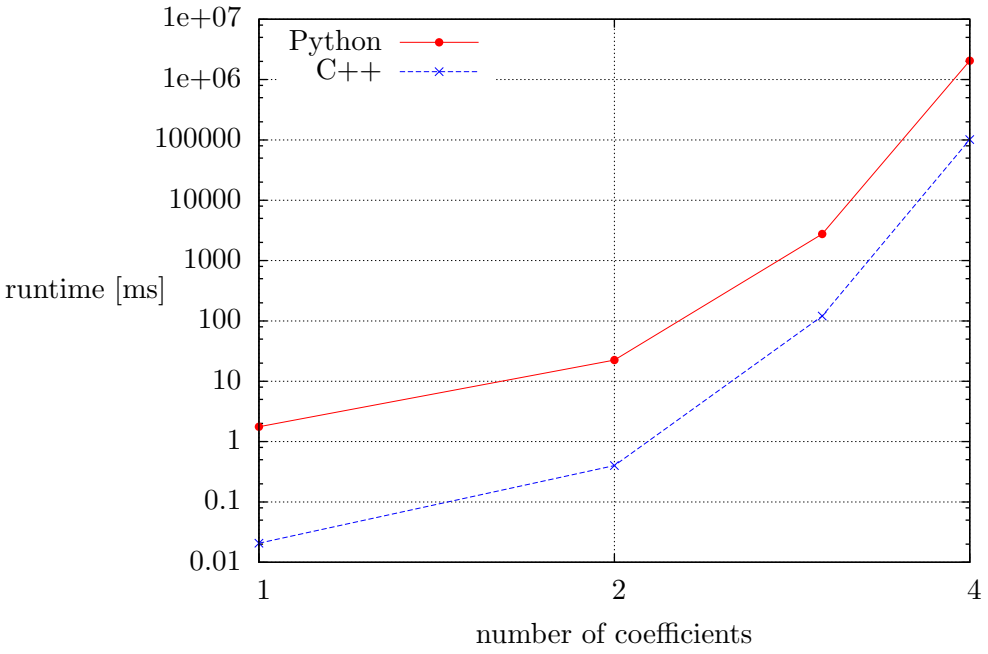

Figure 3: 1D

4 Results

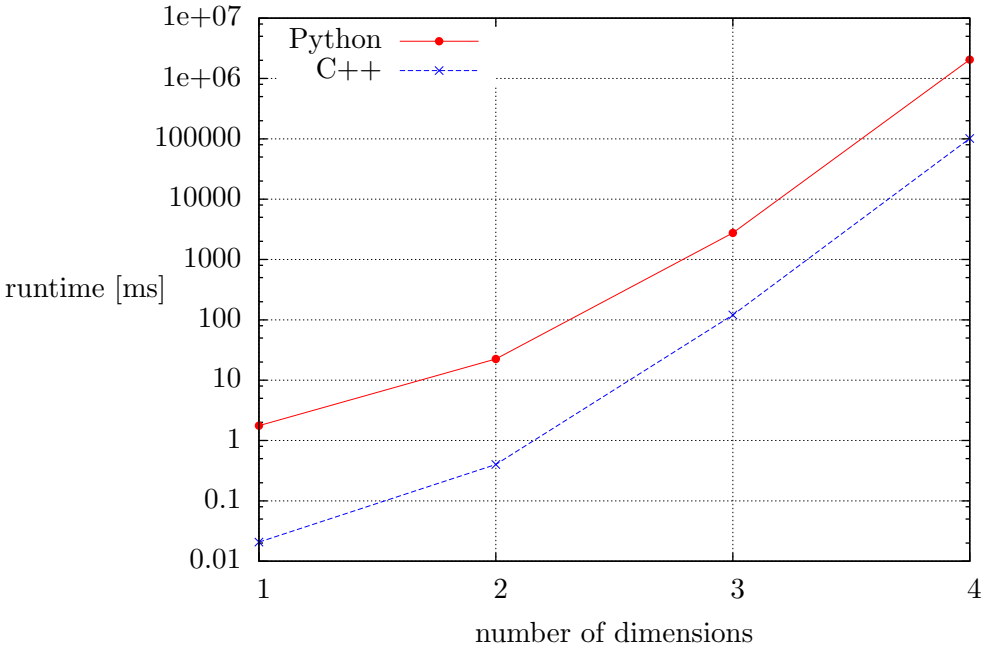

Figure 4: ND

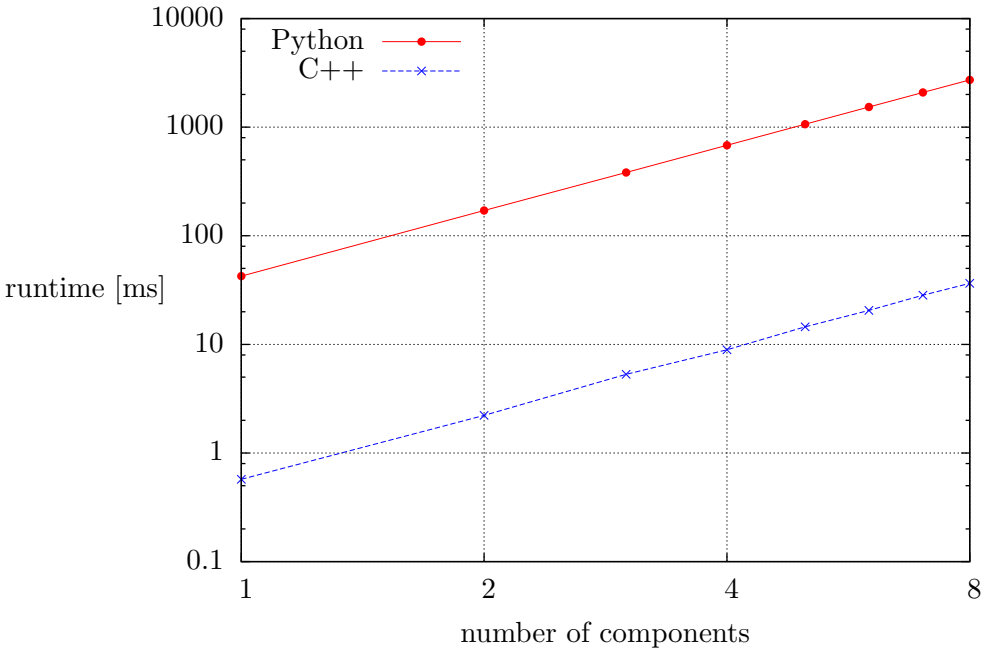

Figure 5: Multiple components

References

[1] R. Bourquin and V. Gradinaru. WaveBlocks: Reusable building blocks for simulations with semiclassical wavepackets. `https://github.com/raoulbq/WaveBlocksND`, 2010 – 2015.

[2] Raoul Bourquin. Wavepacket propagation in d-dimensional non-adiabatic crossings. 2012. `http://www.sam.math.ethz.ch/~raoulb/research/master_thesis/tex/main.pdf`.