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 [2] from Python to C++, using the Eigen template library [4] 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 [3].

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 reasons of porting the code to C++ was to speed up the programs, run-time measurements will also show the achieved improvements.

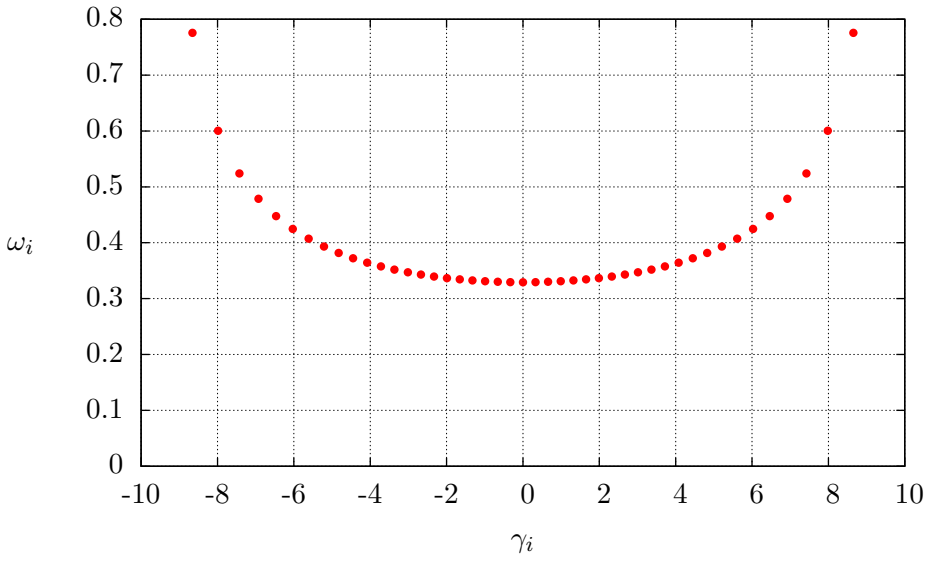

Figure 1: 45-point Gauss-Hermite nodes and weights.

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 [3]. The result is a rule in the form of (1). Figure 1 shows an example of the node distribution of a Gauss-Hermite rule.

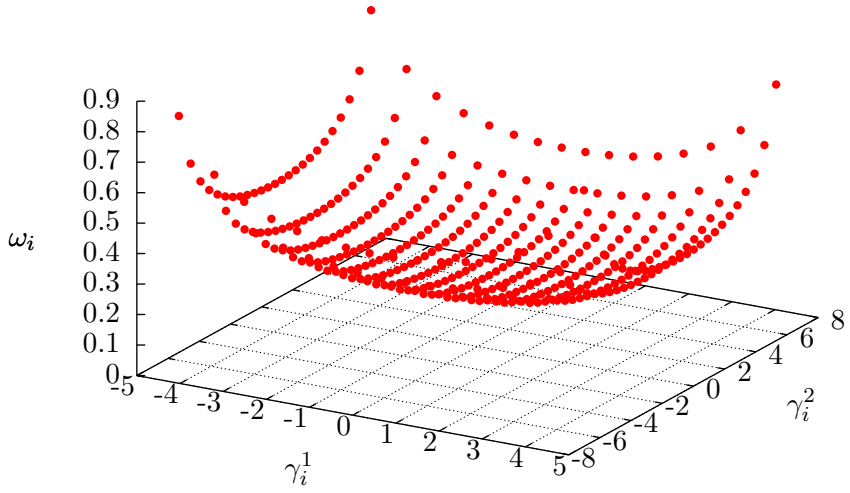

Figure 2: Tensor-product built from 16- and 25-point Gauss-Hermite rules.

For 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 $\underline{j} = (j_1, \ldots, j_d)$ for the resulting tensor quadrature rule is:

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

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

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

2.3 Genz-Keister Quadrature

If one wants to investigate higher dimensions, the tensor rule quickly becomes unfeasible to compute in its full form. As the results in section 4 show, the run-times grow faster than exponentially with increasing dimensionality, single quadratures taking minutes already in the fourth dimension.

For this reason, specialized multi-dimensional schemes, such as the Genz-Keister quadrature implemented here, operate on a reduced set of nodes. The details of the procedure are outlined in [1]. Similarly to the Gauss-Hermite quadrature rule, many of the intermediary constants were tabulated beforehand.

Comparing figures 4 and 5, one can see a difference of an order of magnitude between the computational complexity of tensor and Genz-Keister quadrature. However, the latter can yield less exact results, so the Genz-Keister level has to be chosen suitably.

2.4 Implementation Details

The three implemented quadrature rules share a similar structure in terms of their function interface, but because the code makes heavy use of template programming, no explicit inheritance hierarchy has been defined.

They provide the following methods:

- `number_nodes` for querying the number of nodes (non-trivial for tensor-products and Genz-Keister)
- `nodes` returning the the quadrature node positions,
- `weights` giving the quadrature node weights,
- `nodes_and_weights` as a utility method giving both of the above at once as a `std::pair`, and
- `clear_cache`, described in section (2.5).

Every method of the quadrature rule classes has been made static, so they need not be instantiated. Instead of quadrature rule objects holding a state at run-time, we utilize the type-system and give all the necessary parameters as template arguments. These are the Gauss-Hermite order, tensor-product base rules, and Genz-Keister dimension and level.

This avoids the overhead of constructing objects at run-time just to pass on parameters and makes some compile-time calculations possible. On the other hand, these parameters must be hardcoded, making it impossible to dynamically change the quadrature order, for instance.

Designing `TensorProductQR` to work this way proved to be challenging, since it needs to receive an arbitrary number of other quadrature rule classes,

one for each dimension, as template parameters and call their methods. This was accomplished using template parameter packs[1], introduced in C++11.

2.5 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 (2.2) and Genz-Keister (2.3) quadrature, some calculations must be performed to yield these values. We wish to avoid re-calculating node and weight vectors when the same quadrature rule is used many times, as is done in the case of wavepacket propagation over many timesteps.

For this reason, the results are only calculated once for a given complex rule, stored and re-used automatically on subsequent queries. Since the qudrature rule classes are not instantiated, 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 the method `clear_cache` must be called manually to free the memory used by the cache.

[1] `http://en.cppreference.com/w/cpp/language/parameter_pack`

3 Inner Products

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

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

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

For the mathematical derivation of the braket evaluation algorithms, again refer to [3].

3.1 Scalar Inner Products

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

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

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

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

mapping nodal points $\underline{x} \in \mathbb{R}^D$ and position $\underline{q} \in \mathbb{R}^D$ to the operator's values.

Here, R is the order of the quadrature. Instead of calling the operator function for the different nodes $\underline{x}$ separately, all points are given to the function at once as a matrix $\mathbf{x} \in \mathbb{R}^{D \times R}$ to make efficient vectorized 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 $\underline{c}$ and $\underline{c}'$ of Φ and Φ', returning $\underline{c}^H \mathbf{M} \underline{c}'$.

The "inhomogeneous" evaluation code for different wavepackets $\Phi[\Pi]$ and $\Phi'[\Pi']$ can be found in `waveblocks/inhomogeneous_inner_product.hpp`. If the two wavepackets are equivalent ($\Phi[\Pi] = \Phi'[\Pi']$), a more efficient "homogeneous" algorithm can be used instead 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:

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

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

The calculation of these brakets is built 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`.

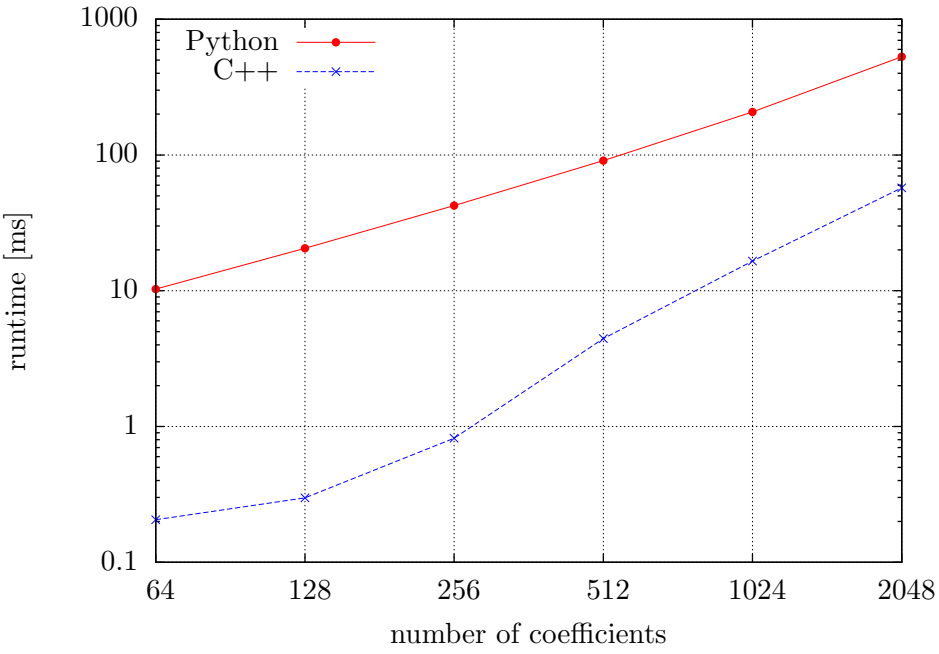

Figure 3: Run-times for 1-D homogeneous quadrature of order 8.

4 Results

Since one of the main reasons of porting WaveBlocksND from Python to C++ was to increase its performance, run-times for various use-cases must be measured and compared to evaluate the improvements. All tests were done using homogeneous quadrature with an identity operator; they can be found in the file `test/test_innerproduct_runtimes.cpp` for the C++ version and `test/RuntimeTest.py` for the Python version. The computer used for benchmarking had a quad-core Intel i5-2500 processor.

Different parameters were varied to compare the scaling behavior of the two implementations. Because the algorithms used in both cases were essentially the same, the run-times grow with the parameters roughly at the same rate, however there is a big difference in the absolute durations.

In the simplest test, the number of coefficients of a wavepacket was changed for a one-dimensional, single-component quadrature. The results in figure 3 show a huge improvement in the C++ version with a speed-up of factor 50 for smaller problems, converging to a factor of about 9 for larger ones.

The next test inspects the impact of dimensionality using tensor-product quadrature. As can be seen from figure 4, higher-dimensional problems can have a 20-fold speed-up.

Figure 5 results from the same type of test as above, but with Genz-

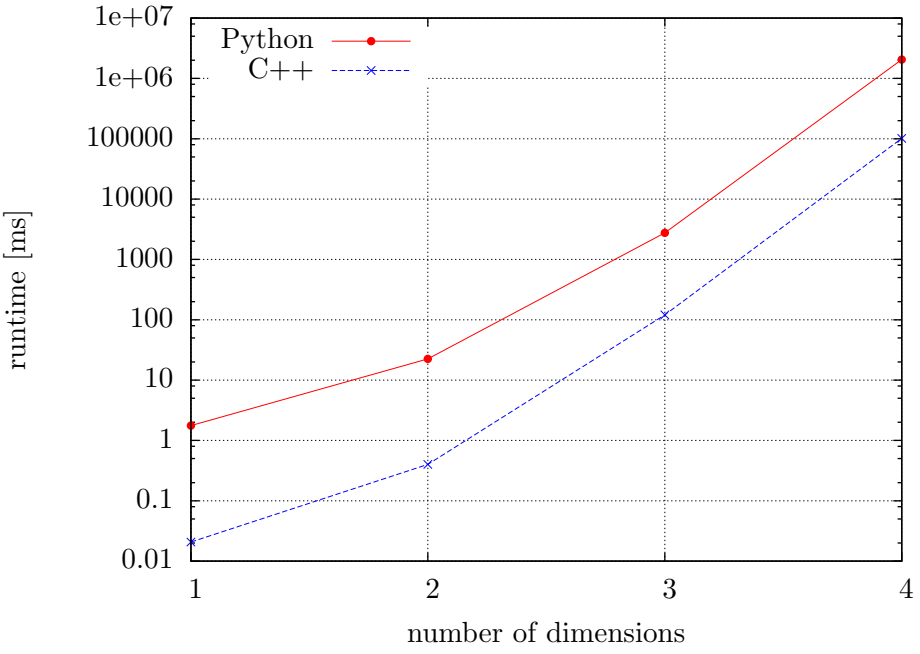

Figure 4: Run-times for multi-dimensional homogeneous quadrature of order 8 with 10 coefficients per dimension.

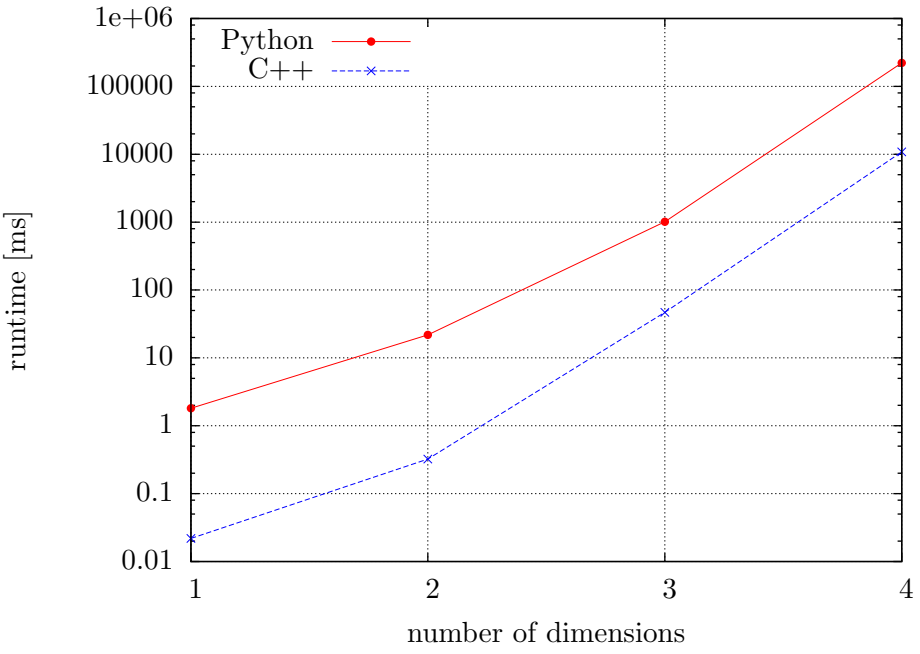

Figure 5: Run-times for multi-dimensional homogeneous Genz-Keister quadrature of level 6 with 10 coefficients per dimension.

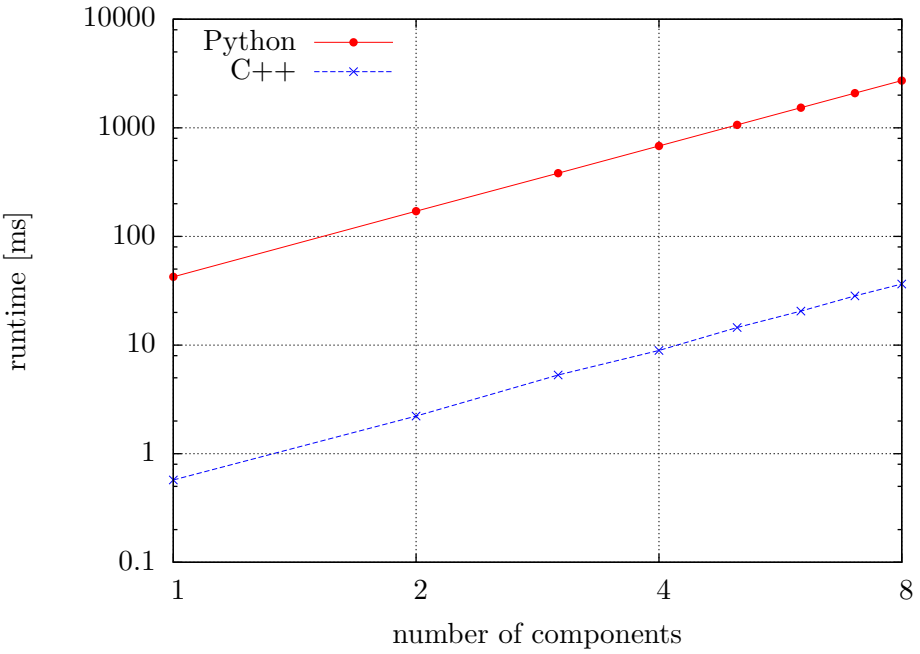

Figure 6: Run-times for multi-component 2-D homogeneous quadrature of order 8 with 10 coefficients per dimension.

Keister quadrature of level 6 instead of dense tensor-products. The speed-up is in a similar range.

Finally, figure 6 shows the behavior for varying sizes of multi-component wavepackets in two dimensions. It is apparent that the speed-up is very consistently a factor of 75.

In conclusion, there is no doubt that replacing the quadrature code with a C++ version using the highly-optimized Eigen library brings great improvements. However, it must be noted that these are isolated measurements and the speed-up is only significant if a large part of the execution time is spent in the quadrature. This means that for smaller problems, even though the measured speed-ups are huge, there are probably bottlenecks in other parts of the code, which diminishes the total improvement. Additionally, a noticeable trade-off is a much longer compilation time, which can hinder doing many test with small problems.

References

[1] R. Bourquin. Exhaustive search for higher-order kronrod-patterson extensions. Technical Report 2015-11, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2015.

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

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

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