A generic lattice sequences library for numerical integration

#include <mcint.hpp> /* The McInt C++ library */

Dirk Nuyens$^a,b$

$^a$School of Mathematics & Statistics, University of NSW, Australia
$^b$Department of Computer Science, K.U.Leuven, Belgium

Biennial Computational Techniques and Applications Conference, CTAC'08
Sunday 13th July – Wednesday 16th July, 2008
ANU, Canberra, Australia
Outline

1. Introduction
2. Integration by lattice rules and lattice sequences
3. The McInt library
4. The lattice sequence generator
5. Conclusion
This talk is about scientific software design:

- a C++ software library for multivariate integration (s ranging from 1 to thousands) called McInt;
- how we use *concepts* and *templates* to build an efficient and generic library.

This talk is not about:

- the underlying lattice rules and sequences, but we will touch this lightly;
- a discussion about the correct programming language or techniques, nor on style.
We consider $s$-dimensional integrals over the unit cube $[0, 1)^s$

$$\int_{[0,1)^s} f(x) \, dx.$$ 

Often a useful approximation is given by lattice rules

$$\frac{1}{n} \sum_{k=0}^{n-1} f\left(\frac{kz}{n}\right).$$

With $z \in \mathbb{Z}_n^s$ an $s$-dimensional integer vector.

Note: product rules don't work in high dimensions since, e.g., taking the minimum of 2 points per dimension in 100 dimensions gives $2^{100} = 1267650600228229401496703205376$ points...
Lattice rules (a traditional view)

A (rank-1) lattice rule is defined by its generating vector, $\mathbf{z}$, and its modulus $n$

$$\frac{1}{n} \sum_{k=0}^{n-1} f \left( \left\{ \frac{k \mathbf{z}}{n} \right\} \right).$$

The traditional notation $\{ \cdot \}$ means to take the fractional part.

→ A better way to look at this is to consider the multiplication to be modulo $n$. Then $k\mathbf{z}$ cycles through the elements of the additive cyclic group $\langle \mathbb{Z}_n^s, + \rangle$.

In fact we have a lattice rule over the quotient ring $\langle \mathbb{Z}_n, +, \cdot \rangle$. 
The choice of the generating vector \( z = (z_1, z_2, \ldots, z_s) \) determines the quality of the lattice rule. Example for \( \mathbb{Z}_{17} \):

A bad choice: \( z = (1, 2) \).  
A good choice: \( z = (1, 5) \).

In previous work we developed fast algorithms to construct good generating vectors \( z \) for various kinds of lattice rules.

See Sloan, Kuo, Joe, … on CBC constructions and N. and Cools on fast CBC constructions.
Lattice sequences (a modern view)

By using an index permutation $\varphi : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$, the lattice rule can be used point by point:

$$x_k = \frac{\varphi(k) \cdot z \mod n}{n} \quad \text{for } k = 0, 1, \ldots, n - 1.$$ 

See Hickernell, Hong, L'Écuyer, Lemieux and Cools, Kuo, N.

A lattice sequence in base 3, i.e., $n_{\max} = 3^m$:

- $n = 3^3$
- $n = 2 \times 3^3 + 3^2 + 1$
- $n = 3^4$
Polynomial lattices (also a modern view)

A lattice point in the normal (scalar) case has lattice points (modulo 1):

\[ x_k = \frac{k \mathbf{z} \mod n}{n} \in [0, 1)^s. \]

With the same mathematical theory we can also define "polynomial" lattice rules. These have polynomial lattice points in a formal Laurent series ring (modulo 1(X)):

\[ x_k(X) = \frac{k(X) \mathbf{z}(X) \mod f(X)}{f(X)} \in \left( \mathbb{F}_b((X^{-1})) \right)^s. \]

These "fractional" polynomials need to be converted to the integration domain \([0, 1)^s\) via a mapping function:

\[ x_k = \nu_{bm}(p_k(X)). \]

See Dick, Pillichshammer.
In its most general form we consider lattice rules over a quotient ring $R$ with point sets generated as

$$x_k = \tau(\varphi(k) z)$$

for $k = 0, 1, \ldots, n - 1$,

with

- $z \in R^s$, with $\langle R, +, \cdot \rangle$ a quotient ring with $|R| = n$ and $\varphi(k) \in \mathbb{Z}_n$ is mapped to elements in $R$;
- $\varphi : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$, an index permutation; and
- $\tau : R \rightarrow [0, 1)$, a mapping to the (standard) integration domain $[0, 1)^s$, applied elementwise.
The McInt library

We now try to mold these mathematical concepts in software.

Design goals:

- The idea is to have a modern C++ library which blends in with current C++ practice.
- This means heavy use of *concepts* and *templates*.
- The starting reference was the Random Number Generator concept (and implementation) from Boost and TR1 for the new C++ standard library (both by Jens Maurer).

Note: TR1 contains lots of *models* of the Random Number Generator concept. E.g., simple linear congruential generators `linear_congruential<Int, a, c, m>` or the Mersenne Twister `mersenne_twister<...>` with a typedef for `mt19937`. 
The McInt library

A basic McInt usage example

```cpp
#include <iostream>
#include <mcint/mcint.hpp>

using namespace std;
using namespace mcint;

inline double f(const double *x) { ... }

int main(void) {
    const size_t s = 2, n = 89;
    unsigned int z[s] = {1, 55};
    rank1_lattice< s, Zn<n>, radical_inverse_order<2, n> > P(z);

    double Qs = 0;
    mc_integrate(f, P, n/2, Qs);
    cout << "half the number of points: Q(f) = " << Qs/(n/2) << endl;
    mc_integrate(f, P, n-n/2, Qs);
    cout << " and now using all points: Q(f) = " << Qs/n << endl;

    return EXIT_SUCCESS;
}
```
As with the random engines from Boost and TR1 the parameters for the generator appear as compile time constants. (Except for the generating vector.)

```cpp
const size_t s = 2, n = 89;
unsigned int z[s] = {1, 55};
rank1_lattice< s, Zn<n>, radical_inverse_order<2, n> > P(z);
```

The calling sequence for (quasi-)Monte Carlo integration is extremely simple (and flexible due to templatization).

```cpp
double Qs = 0;
mc_integrate(f, P, n/2, Qs);
cout << "half the number of points: Q(f) = " << Qs/(n/2) << endl;
mc_integrate(f, P, n-n/2, Qs);
cout << " and now using all points: Q(f) = " << Qs/n << endl;
```
Both from a user perspective and a designer perspective the `mc_integrate` routine plays a crucial rule.

```
// mc_integrate(f, gen, max_n, result) {{
template <typename PointGen, typename Function, typename Result
    typename WeightGen = infinite_ones>
inline size_t
mc_integrate(const Function& f, PointGen& gen, size_t max_n,
    Result& result, WeightGen wgen = 1)
{
    size_t n;
    for(n = 1; (n <= max_n) && !converged(result); ++n) {
        result += f(gen()) * wgen();
    }
    return n;
}
```
Both from a user perspective and a designer perspective the \texttt{mc_integrate} routine plays a crucial rule.

```cpp
1 // mc_integrate(f, gen, max_n, result) {{
2 template <typename PointGen, typename Function, typename Result
3 typename WeightGen = infinite_ones>
4 inline size_t
5 mc_integrate(const Function& f, PointGen& gen, size_t max_n,
6 Result& result, WeightGen wgen = 1)
7 {
8     size_t n;
9     for(n = 1; (n <= max_n) && !converged(result); ++n) {
10         result += f(gen()) * wgen();
11     }
12     return n;
13 }
14 // }}
```

- The sample point generator \texttt{gen} delivers a vector on each call.
Both from a user perspective and a designer perspective the \texttt{mc\_integrate} routine plays a crucial rule.

```cpp
1 // mc\_integrate(f, gen, max\_n, result) {{
2 template <typename PointGen, typename Function, typename Result
3     typename WeightGen = infinite\_ones>
4 inline size\_t
5 mc\_integrate(const Function& f, PointGen& gen, size\_t max\_n,
6     Result& result, WeightGen wgen = 1)
7 {
8     size\_t n;
9     for(n = 1; (n <= max\_n) && !converged(result); ++n) {
10        result += f(gen()) * wgen();
11    }
12    return n;
13 }
14 // }}}
```

- The result accumulates the function evaluations.
Both from a user perspective and a designer perspective the `mc_integrate` routine plays a crucial rule.

```cpp
// mc_integrate(f, gen, max_n, result) {{
template <typename PointGen, typename Function, typename Result
    typename WeightGen = infinite_ones>
inline size_t
mc_integrate(const Function& f, PointGen& gen, size_t max_n,
    Result& result, WeightGen wgen = 1)
{
    size_t n;
    for(n = 1; (n <= max_n) && !converged(result); ++n) {
        result += f(gen()) * wgen();
    }
    return n;
} // }}
```

- The free standing function `converged` provides a generic convergence test.
mc_integrate(f, gen, max_n, result):

- Accepts any function which can be called as f(x).
  → standard STL Functor concept.

- The sample point generator gen delivers a vector on each call.
  → *Point Generator* concept.

- The result accumulates the function evaluations.
  → *Result Gatherer* concept.

- The free standing function converged provides a generic convergence test.
  → Typed on the Result Gatherer concept.

Because of the templatization you can plug in any model of these concepts that you like. E.g., the type `double` will do as a Result Gatherer and gets the default version of converged which is a “do nothing” (i.e., always `false`). Which gets optimized out...
A **Point Generator** is a refinement of
  - the STL Generator concept and
  - the STL Forward Input Iterator concept.

In short, if \( a \) is of type \( X \) and type \( X \) is a model of this concept then it behaves like:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++( a )</td>
<td>pre-increment, move to the next point in the sequence</td>
</tr>
<tr>
<td>( a++ )</td>
<td>post-increment, move to the next point in the sequence</td>
</tr>
<tr>
<td>( *a )</td>
<td>return the current point</td>
</tr>
<tr>
<td>( a() )</td>
<td>return the current point and move to next point</td>
</tr>
</tbody>
</table>

Also \( a == b \) and \( a != b \) have their trivial meaning.

The return type of \( *a \) and \( a() \) refers to an \( s \)-dimensional vector, \( 1 \leq s \). In its simplest form this could be a pointer to static memory since an Input Iterator is invalidated as soon as it moves to the next position.
The Result Gatherer concept just models an accumulator to store the quadrature sums. In its simplest form it could be a `double`.

A more complex model is also implemented:

- An adapter is used to transform the Point Generator into multiple streams by adding $M$ random shifts to each point.
- The Result Gatherer is now modeled as a list of $M$ accumulators.
- The free function `converged` now uses a statistical error estimator based on the standard error of the $M$ results.

In other words this is a randomized rule:

$$Q(f) = \frac{1}{m} \sum_{\ell=1}^{m} \frac{1}{n} \sum_{k=1}^{n} f(\{x_k + \Delta_\ell\})$$

using the Chebyshev inequality as an error estimator.
We defined the point set of a rank-1 lattice rule as

\[ \mathbf{x}_k = \tau(\varphi(k) \mathbf{z}) \quad \text{for } k = 0, 1, \ldots, n - 1. \]

Now remember the basic usage example:

```cpp
const size_t s = 2, n = 89;
unsigned int z[s] = {1, 55};
rank1_lattice< s, Zn<n>, radical_inverse_order<2, n> > P(z);
```

This instantiates the `rank1_lattice` template:

- in \( s \) dimensions;
- over a quotient ring \( R \); and
- using an index permutation \( \varphi \) (the `IndexGen` type).

Obviously `rank1_lattice` is a model of Point Generator. The abstraction of the quotient ring gives us easy polynomial lattice rules.
An *Index Generator* is the modeling of a permutation. This is similar to a one-dimensional Point Generator.

Three models are provided:

- `natural_order<>`: just 0, 1, ..., all decent compilers should be able to optimize this away;
- `radical_inverse_order<b, n>`: generates the integer version of the radical inverse function in base $b$;
- `gray_order<b, n>`: returns the radical inversion of the Gray code sequence in base $b$.

Both the radical inverse order and the Gray code order are well known in the low-discrepancy literature.

If $n$ is not an integral power of $b$ then an acceptance rejectance strategy is followed to generate indices in $[0, n)$.

See Cools and N.
The acceptance rejectance strategy would be in the way for a generator which does not need it. In that case it is excluded at compile time:

```cpp
radical_inverse_order& operator++() // pre-increment
{
    accept_reject_helper< radical_inverse_order,
        n != mp_power<b, m>::value >()(*this);
    return *this;
}
```

- Each time the generator has to move to the next point operator++ is called.
- There the acceptance rejectance strategy is followed if \( n \neq b^m \). Hereby \( mp_power<b, m> \) calculates \( b^m \) at compile time via template meta programming. Likewise \( m \) was already found via template meta programming for other purposes.
An example of implementation: radical inverse order

```cpp
// accept_reject_helper {{
template <typename Mapper, bool ar>
struct accept_reject_helper
{

    void operator()(Mapper& m) const {
        typedef typename
            integer_traits<typename Mapper::value_type>::associated_uint_t uint_t;
        if((uint_t)(m.i) < m.maxn - 1) {
            m._next_();
            while((uint_t)(m.phik) >= m.maxn) m._next_();
            ++(m.i);
        }
    }
}

};

template <typename Mapper>
struct accept_reject_helper<Mapper, false>
{
    void operator()(Mapper& m) const {
        m._next_();
        ++(m.i);
    }
};

// }}
```
Template specialization is used to implement efficient versions for base 2 in case of radical_inverse_order and gray_order.

E.g., calculating the radical inverse in base 2 can be done via templates in \(\lg(m)\) steps for an \(m\) bit number as

\[
\begin{align*}
\text{phik} &= \text{reverse_step}\langle 1, m, \text{UInt}\rangle(\text{phik}); \\
\text{phik} &= \text{reverse_step}\langle 2, m, \text{UInt}\rangle(\text{phik}); \\
\text{phik} &= \text{reverse_step}\langle 4, m, \text{UInt}\rangle(\text{phik}); \\
\text{phik} &= \text{reverse_step}\langle 8, m, \text{UInt}\rangle(\text{phik}); \\
\text{phik} &= \text{reverse_step}\langle 16, m, \text{UInt}\rangle(\text{phik}); \\
\text{phik} &= \text{reverse_step}\langle 32, m, \text{UInt}\rangle(\text{phik});
\end{align*}
\]

where the reverse_step template exchanges blocks of consecutive bits, doing something like

\[
((\text{phik} \& \text{even_mask}) \ll s) | ((\text{phik} \& \text{odd_mask}) \gg s)
\]

but only when necessary for the data type.
We have shown a software library which is built out of small building blocks.

These building blocks are recombined to build, e.g., a specific rank-1 lattice point generator or Monte Carlo type integrator.

Optimal code is generated by using template specialization and compile time meta programming.

The library provides:

1. classical lattice rules;
2. lattice sequences in any base for any number of points, using either radical inverse ordering or Gray ordering;
3. polynomial lattice rules with the same flexibility as above; and
4. an implementation of the Sobol’ sequence as a Point Generator.

The McInt library is still being worked on, but nearly finished. Email me for the code.