Implementation of k-kuplesieve

ENS Lyon

18.12.2017

Outline

Outline

Loose coupling

Loose coupling

- Make individual components (classes) of the code independent, coupled only by their interface.
- Make it possible to replace individual classes by other classes that provide the same functionality / interface by changing at most a few lines of code.
- Individual parts should not need to know about each other.
- Various ways to achieve this:
  - interface classes from which concrete classes are derived.
  - (either virtual or not)
  - pass the used classes as template arguments.

Example: Z_NR

If we (re-)did Z_NR with our design patterns

Z_NR provides a common interface for **double**, **long** and **mpz_t**

- Specify the interface (in documentation, not in code)
- Write classes / class templates that satisfy this interface:
  - **class** Z_NRForMPZ ...
  - **template** <**class** Integer> **class** Z_NRForIntegralType ...
  - **template** <**class** FloatType> **class** Z_NRForFloatType ...
- use it as a template parameter
  template <**class** ZNRType> void foo(ZNRType &x)
  ... (Assumes that ZNRType satisfies the interface)
- call as Z_NRForMPZ x; foo(x);
- NOT: **template**<**class** T> void foo(Z_NR<T> &x)

Use full set of features of C++11

Use modern C++11 feature

- Lots of template code (and typedefs)
- Use rvalues and move semantics to avoid copying
  e.g. std::list<T> has members
  void push_back(**const** T &value);
  void push_back(T &&value);
  $\longrightarrow$ main_list.push_back(std::move(new_point));
- Use of **constexpr** for compile-time computations.
- Namespaces: Everyhing is in namespace GaussSieve.

Limitations

Limitations

- Everything is a template, so this is a single translation unit. $\longrightarrow$ header-only.
- We use global variables.

Outline

Usage

Usage of class

#include "SieveGauss_main.h"
#include "fplll.h"
...
using Traits = GaussSieve::DefaultSieveTraits<int32_t, false, -1, fplll::ZZ_mat<mpz_t>>;
GaussSieve::Sieve<Traits,false> sieve(basis, ...); \\optional args
sieve.run();
sieve.print_status();

Standalone executables

Files sieve_main.cpp → newlatsieve executable
test_newsieve.cpp → testsieve executable
Follow the same syntax as the old sieving code.

Outline

TODO

Graphic

C++ versioning

Compat.h

This file contains macros that detect availability of features from C++14 / C++17 or compiler specific features and selectively enables them.

We use some library extensions from beyond C++11, collected in **namespace** mystd.
e.g. mystd::decay_t, mystd::bool_constant.

Genrally, mystd:: features are identical to the std:: features.
Often-used Macros: CPP14CONSTREXPR, TEMPL_RESTRICT_*

Utility functions

SieveUtility.h

Collects some general utility functions and classes.

- convert_to_double(Source **const** &arg);
- convert_to_inttype<TargetType>(Source **const** &arg);

Works with arbitary integral, floating types, mpz_t and mpz_class.

- template<**int** nfixed, **class** UIntType = **unsigned int**> **class** MaybeFixed;

MaybeFixed<-1> encapsulates a run-time integer.
MaybeFixed<n> encapsulates a compile-time integer.

DefaultIncludes.h, DebugAll.h

DebugAll.h

Defines some DEBUG_SIEVE_$*$ macros

DefaultIncludes.h

Since we only have 1 translation unit, we collect all standard library includes in one file:

```
#include <array>
#include <atomic>
#include <bitset>
#include <cmath>
#include <cstdint>
#include <exception>
...
```

Typedefs.h

Typedefs.h

Defines a traits class that collects “global” typedefs, e.g.

```
template<[...]> class DefaultSieveTraits
[...]
using GaussList_ReturnType = FastAccess_Point;
using GaussQueue_ReturnType = GaussSampler_ReturnType;
using GaussQueue_DataType = GaussQueue_ReturnType;
using InputBasisType = InputBT;
using PlainPoint = PlainLatticePoint<ET,nfixed>;
using DimensionType = MaybeFixed<nfixed>;
[...]
```

Template parameter to most replaceable modules. Think of this as a “config” file.

Sieve

SieveJoint.h, SieveJoint_impl.h

Here, the Sieve class (which the user uses) is declared.

SieveST.cpp, SieveST2.cpp, SieveST3.cpp

Definitions for Sieve member functions specific to the single-threaded case.

SieveGauss.h, SieveGauss_main.h

SieveGauss_main.h is the main include file for users.
These file perform evil macro and include shenanigans.

Note that the naming / structure here is subject to change.

LatticeBases.h

LatticeBases.h

This file is responsible for reading the input lattice base, storing appropriate data and converting to our internal data structures.

- We support fplll's Z_NR input-types.
- Internally, we use normal arithmetic types or mpz_class.

Note: Z_NR is restricted to e.g. long. We want to be able to use shorter data types.

GaussQueue

GaussQueue.h, GaussQueue_impl.h

These files define the class used to store the queue of points that are yet to be processed by the queue. If the queue is empty, trying to retrieve a vector, it automatically samples a fresh one.

Sampler

Sampler.h, Sampler_impl.h

Sampler.h defines an abstract class (template) Sampler for sampling lattice points.

- Abstract class with pure virtual members.
- All samplers that are used need to be derived from this class

UniformSampler.h, GPVSampler.h, GPVSamplerExtended.h

These are concrete available samplers, derived from Sampler.

We use run-time polynomorphism here. The constructor of the sieve can take a user-prodvided sampler that was derived from Sampler.

Termination Conditions

TerminationConditions.h

TerminationConditions.h defines an abstract class (template) TerminationCondition that is queried to decided whether to stop the sieve.

- Abstract class with pure virtual members.
- All termination conditions need to be derived from this.

DefaultTermConds.h, DefaultTermConds_impl.h

Here, we provide some meaningful default termination conditions.

We use run-time polynomorphism. The user can define and give us his own termination condition class.

Static Data

GlobalStaticData.h

We use global variables. These are initalized and controlled by class templates StaticInitializer<Type>.

- Initialization status of global variables is tied to the lifetime of RAII - style wrappers.
- See GlobalStaticData.h for details.

Lattice Points

LatticePointConcept.h

This class declares the generic interface (concept) for classes that store lattice points. Note that concrete lattice points need to derive from the class declare here.

LatticePointGeneric.h

Here, we provide generic functions for lattice points (like addition, scalar products etc.)

ExactLatticePoint.h, PlainLatticePoint.h

Concrete classes that store lattice points (ExactLatticePoint stores a pre-computed norm-square, PlainLatticePoint does not)

SimHashes

SimHash.h

Defines functions and global variables required to use SimHashes. A SimHash of a point is a small sketch of a point that can be used to give a prediction of whether a lattice point is likely to lead to a reduction. Operating on sim hashes is much faster than a full scalar product.

BlockOrthogonalSimHash.h

Here, we define the concrete SimHash we are using.

PointWithBitapprox.h

Provides functionality to incorporate a simhash into a lattice point.

GaussListBitapprox.h

GaussListBitapprox.h

This defines the class used to store the main list of lattice points that we amass during the algorithm. Note that, for various efficiency reasons, the use of sim hashes are incorporated into the list class.

Statistics.h

Statistics.h

Used to manage and collect statistics during the run of the sieve. Useful for debugging.

Filtered Points

FilteredPoint.h

For the k-sieve with $k \geq 3$, the algorithm needs to construct temporary sublists of the main list. This file is responsible for these sublists. Note that we may want to store more data than just a pointer to the lattice point, for reasons of efficiency.