CLGRP 1.3
Unconditional Class Group Tabulation

Anton S. Mosunov

October 30, 2016

Contents

1 Introduction

The CLGRP library contains the implementation of various subroutines utilized for the tabulation of class groups of imaginary quadratic fields, as well as the unconditional verification of this tabulation. To parallelize the computations of the tabulation algorithm, the OpenMPI library is utilized.

CLGRP is maintained by Anton S. Mosunov, University of Waterloo, and is an appendix to the Master's thesis [Mos14], written under the supervision of Michael J. Jacobson, Jr. It is highly recommended to read the thesis (Section 4 in particular) before utilizing the library, as certain notions, such as the bundling parameter or the bit size parameter, are not defined in this manual.

The manual for CLGRP is based on the LaTeX template for the manual of FLINT 1.0:
http://web.mit.edu/sage/export/tmp/flint-1.1/doc/flint-roadmap.tex.

2 Changes since the previous version

October 29th, 2016. Since the version 1.2, the following changes had been made:

- The `configure` file is now a part of CLGRP. In contrast, the previous version contained only the Makefile and required the user to edit it manually;
- All the files are now supplied with headers, providing information on the GNU General Public License.

3 Dependencies in CLGRP

CLGRP depends on several libraries and specifications that need to be present on your system prior to the installation. These libraries are:

1. `GMP`, gmplib.org. The GNU multiple precision arithmetic library;
2. `OpenMPI`, open-mpi.org. A high performance message passing library;
3. `optarith`, github.com/maxwellsayles. Optimized arithmetic operations for 32, 64, and 128bit integers. Includes optimized implementations of many different extended GCD algorithms;
4. `PARI/GP`, http://pari.math.u-bordeaux.fr/. Computer algebra system designed for fast computations in number theory. In CLGRP, it is used solely for debugging purposes;
5. `qform`, github.com/maxwellsayles. Ideal class group arithmetic in imaginary quadratic fields.

Before installing, make sure that each of those libraries is installed. The only exception can be made regarding the `PARI/GP` library utilized for debugging purposes. In this case, when utilizing the CLGRP library make sure you don't use `-DWITH_PARI` when compiling your program, as the code will not compile.

4 Building and using CLGRP

The easiest way to use CLGRP is to build it using `make`. The `make` and the `make verify` command create executables, while the `make lib` command creates a static library.

5 Reporting issues

The maintainers wish to be made aware of any bugs in the library or typos in this manual. Please send an email with your bug report to `amosunov@uwaterloo.ca`.

If possible please include details of your system, version of gcc, version of GMP and precise details of how to replicate the bug.

Note that CLGRP needs to be linked against version 4.2.1 or later of GMP and must be compiled with gcc version 4.2 or later.

6 Files

The CLGRP library contains the following files:

1. The `functions.c` file contains miscellaneous functions utilized by the class group computation and verification algorithms. It also contains basic implementations of an indexed hash table type `htab_t`, vector type `vec_t` and a matrix type `mat_t`. See Subsection 9 for more details;
2. The `sieve.c` file contains implementations of various prime sieves. See the Subsection 10 for more details;
3. The `clgrp.c` file contains the implementation of the Buchmann-Jacobson-Teske Algorithm, or BJT, applied to the computation of a class group of some imaginary quadratic field. It also contains the subroutine for the tabulation of class groups. See Subsection 11 for more details;
4. The `verify.c` file contains four subroutines utilized for the computation of the Eichler-Selberg trace formula, used for the unconditional verification of tabulated class groups. See Subsection 12 for more details;
5. The files entitled `functions.h`, `sieve.h`, `clgrp.h` and `verify.h` contain declarations of all the subroutines implemented in files listed previously;
6. The `clgrp_main.c` file contains the implementation of a command line program for the class group tabulation. This is the only file where the OpenMPI library is used for the parallelization. See Subsection 13 for more details;
7. The `verify_main.c` file contains the implementation of the command line program for unconditional verification of tabulated class groups. See Subsection 14 for more details.

7 Macros

The `MAX_RANK` macro defined in `clgrp.h` denotes the maximum rank among all the class groups that get tabulated. The standard value is 10, so please adjust it if you know that there is a possibility that ranks of class groups you'll be considering may exceed this quantity;

The `FAC_TOTAL` macro defined in `sieve.h` is utilized by the `tabulate_bjt` routine, and determines how many discriminants get factored at the same time by the sieve;

The `MAX_DIVISORS` macro defined in `verify.h` is utilized during the verification and corresponds to the largest number of divisors that an integer can possess. By default, it corresponds to the integer 963761198400 with `MAX_DIVISORS` = 12000 divisors. No integer up to 2^{40} has more than 12000 prime divisors;

The `WITH_INDICES` macro defined in `functions.h` is utilized by certain sieves. When the `flags` parameter of a particular routine is set to `WITH_INDICES`, instead of determining the actual prime factors of an integer it determines only the *indices* of prime factors (say, if the 6th prime $p_6 = 13$ divides n, the function will record 6, *not* 13). This allows to simplify certain parts of the verification procedure.

The following macros can be defined during the compilation:

- `KEEP_FILES`. This macro is utilized solely by the `tabulate_bjt` routine, defined in `clgrp.c`. The prefixes of binary files with precomputed class numbers of imaginary quadratic fields can be provided to this function. This is done in order to speedup the tabulation, as the knowledge of a class number allows to discard certain p-subgroups of each class group from consideration. When the class groups get computed, these files get removed for the sake of cleaning up the space on the hard disk. When defined, the `KEEP_FILES` macro prevents the deletion of binary files. It is highly recommended to define this macro when compiling the program.
- `DEBUG`. When defined, this macro allows to trace the process of computation of the `compute_group_bjt` routine, defined in `clgrp.c`. It prints out a detailed information on every iteration of the BJT algorithm.
- `WITH_PARI`. When defined, this macro allows to verify whether the tabulation/verification of class groups was performed correctly.

8 Setup

Before installing the CLGRP library, make sure that all the libraries that CLGRP depends on are installed (see the full list in Section 3). **Pay a particular attention to the `optarith` library!** Before installing it, ensure that there are enough primes defined in files `primes.c`, `primes.h`, and their quadratic residues are precomputed in `sqrtmodp_list.c`, `sqrtmodp_list.h`. Several functions of the CLGRP library, namely `next_prime` in `functions.c` and `next` in `clgrp.c`, heavily rely on these precomputed values. The reason is that as the discriminant gets bigger, more prime ideals may need to be considered to generate the whole group. To derive how much primes you need you may utilize the *conditional* upper bound on the number of prime generators due to Bach [Bac90], which is $6\log^2|\Delta|$. If you observed that there are not enough primes suitable for your needs, please compile the program `gen_sqrtmodp.cc` located in the folder `code_gen`. Run this program by providing to it the total number of primes you wish to generate as a parameter. It will generate two new files, `sqrtmodp_list.c` and `sqrtmodp_list.h`. Replace the old files with this name by the new ones, and then build the `optarith` library again with these new files.

In order to prepare the CLGRP library for compilation, please edit the `Makefile`. The `Makefile` defines the following four commands:

- The `make` command builds an executable `clgrp` which allows to tabulate class groups. See Subsection 13 on how to use it;
- The `make verify` command builds an executable `verify` which allows to verify tabulated data unconditionally. See Subsection 14 on how to use it;
- The `make lib` command builds a static library `libclgrp.a`, which incorporates four object files: `clgrp.o`, `functions.o`, `sieve.o` and `verify.o`, produced from `clgrp.c`, `functions.c`, `sieve.c` and `verify.c`, respectively. The file `libclgrp.a` should be placed in your library path. See Subsection 17 on how to link this library to your program;
- The `make clean` command removes all the files which have the extension `.o`. Use this command right after running `make lib` to remove all the object files.

9 Supplementary types and functions

The `functions.c` file contains implementation of three types, utilized by the class group tabulation program:

- `htab_t`: a generic indexed hash table with separate chaining. Utilizes the hash function $f_p \colon n \mapsto n \pmod{p}$, where p is some predefined prime. Basic functionality is implemented, including insertion/deletion of an element, and deletion of an element defined by a specific index;
- `vec_t`: a dynamic array with entries of type `int`;
- `mat_t`: a matrix with entries of type `int`. The only functionality provided is initializing, printing and clearing. The type is used by the `smith_normal_form` subroutine, which computes the Smith Normal Form (SNF) of a matrix. The SNF computation is an essential part of the BJT algorithm.

```
int next_prime(const int n)
```

Computes the prime which immediately follows `n`. Utilizes `prime_list` defined in `primes.c` of the `optarith` library.

```
long crt(const int a, const int m, const int  b, const int n, const long min)
```

Computes the smallest number $x \geq$ `min` such that $x \equiv$ `a` (mod `m`) and $x \equiv$ `b` (mod `n`). This is an extra function utilized by the `mod_sieve` routine defined in `sieve.h`.

```
char kronecker_symbol(long a, long p)
```

Computes the Kronecker symbol $\left(\frac{a}{p}\right)$.

```
long divisors_list(long * result, long D, const int * pfactors,
                   const int * primes, const int flags)
```

Computes the divisors of `D` and saves them into `result`. The `pfactors` array contains either prime divisors of `D` or their indices, depending on whether the `flags` parameter is = 0 or `WITH_INDICES` (note that `pfactors[0]` contains the total number of prime factors). In the latter case, the `primes` array with precomputed list of primes is essential to find the actual `i`-th prime divisor by its index via `primes[pfactors[i]]`.

10 Sieves

The class group tabulation procedure heavily relies on the integer factorization. There are two places where it occurs during the work of the `tabulate_bjt` routine defined in `clgrp.c`. First of all, as we are interested only in fundamental discriminants (i.e. those Δ that are not divisible by an odd square and satisfy the congruence $\Delta \equiv 1, 5, 8, 9, 12, 13 \pmod{16}$), it is important to factor each Δ to see whether it is fundamental or not. Further, if the class number $h(\Delta)$ is known, the factorization of $h(\Delta)$ can tell us which p-subgroups can be ignored during the class group structure computation. As the factorization of each individual discriminant takes a lot of time, the sieving procedure is utilized to factor all the discriminants and class numbers in bulk.

The verification procedure requires sieving when computing the right hand side of the Eichler-Selberg trace formula (see [Mos14, Section 5.3]).

Warning: before using the functions `regular_sieve`, `segmented_sieve` and `mod_sieve`, make sure that enough memory is allocated for the two-dimensional array `factors` by precomputing the maximum number of prime factors that your integers can have. This can be done by determining the number k such that $p_1 \cdot p_2 \cdot \ldots \cdot p_k \leq \texttt{blocksize} < p_1 \cdot p_2 \cdot \ldots \cdot p_k \cdot p_{k+1}$, where p_i denotes the i-th prime. Every n-th array `factors[`n`]` is an array of $k+1$ elements, as the 0-th entry contains the total number of prime factors. To find the j-th prime factor of an integer n, write `factors[`n`][`$j+1$`]`. To determine the total number of prime factors less than `max_prime`, write `factors[`n`][0]`.

The following functions are implemented in `sieve.c`:

```
void prime_sieve(const int max_prime, int * primes)
```

Computes all primes less than `max_prime` using the sieve of Eratosthenes and saves them to `primes`.

```
void regular_sieve(const int max_prime, const long blocksize, int ** factors,
                   const int * primes, const int flags)
```

This subroutine computes all prime factors less than `max_prime` of every integer less than `blocksize`. The precomputed primes are contained in `primes`. The result is saved into a two-dimensional array `factors`. When `flags=WITH_INDICES`, instead of prime factors the prime indices get stored.

```
void segmented_sieve(const int max_prime, const long blocksize, const long l,
                     int ** factors, const int * primes, const int flags)
```

This subroutine computes all prime factors less than `max_prime` of every integer between `l` and `l` + `blocksize` (exclusive). The precomputed primes are contained in `primes`. The result is saved into a two-dimensional array `factors`. When `flags=WITH_INDICES`, instead of prime factors the prime indices get stored.

```
void mod_sieve(const long blocksize, const long l, int ** factors,
               const int * primes, const int a, const int m)
```

This subroutine computes all prime factors of every integer congruent to `a` (mod `m`) between `l` and `l` + `blocksize` $\cdot$ `m` (exclusive). The precomputed primes are contained in `primes`. The result is saved into a two-dimensional array `factors`.

11 Class group computation and tabulation

The following functions are implemented in `clgrp.c`:

```
void pari_verify(int * result, const long D)
```

This function gets declared whenever `WITH_PARI` macro is defined during compilation. Given the discriminant `D` (positive or negative) and the class group structure saved to `result`, it verifies the correctness of the data in `result` by computing the class group corresponding to `D` using PARI/GP. In case if the results do not match, the subroutine prints out the error message and terminates the program. This function is utilized by the `tabulate_bjt` function.

```
int next(group_pow_t * gp, form_t * R, const int init_pow, int prime_index)
```

An extra function utilized by the `tabulate_bjt` subroutine. The `gp` parameter defines the class group; in particular, it contains the discriminant Δ. The subroutine iterates through primes $p_1 = 2, p_2 = 3, \ldots$ starting from index `prime_index` until the index k is reached such that $b^2 \equiv \Delta \pmod{p_k}$ for some $b \in \mathbb{Z}$. As a result, the binary quadratic form $\big(p_k, b, (b^2 - \Delta)/4a\big)^{\texttt{init\_pow}}$ gets saved into `R`, and k gets returned.

```
int h_upper_bound(const long D)
```

Computes the upper bound on the class number $h(\mathtt{D})$ using Dirichlet's class number formula and Ramaré's bounds on $L(1, \chi_\Delta)$ (see [Mos14, Section 4.3]).

```
int h_lower_bound(const long D)
```

Computes the *conditional* lower bound h^* of the class number $h(\mathtt{D})$ satisfying $h^* \le h(\Delta) \le 2h^*$ using Bach's bound [Bac90]. Utilized by the `tabulate_bjt` subroutine.

```
int compute_group_bjt(int * result, const long D, const int init_pow,
                      const int h_star, htab_t * R, htab_t * Q)
```

Computes the subgroup of order $h(\mathtt{D})/\texttt{init\_pow}$ of a class group corresponding to the negative discriminant `D`. Here, `h_star` is the approximation of a class number $h(\mathtt{D})$, satisfying $\texttt{h\_star} \le h(\mathtt{D}) \le 2\texttt{h\_star}$. The `init_pow` parameter must divide $h(\mathtt{D})$. If the class number is unknown, set to 1. If the prime factorization $h(\mathtt{D}) = p_1^{e_1} \cdot \ldots \cdot p_k^{e_k}$ is known, set $\texttt{init\_pow} = \prod_{\substack{1 \le i \le k \\ e_i = 1}} p_i$. The resulting structure of a subgroup gets saved into `result`. The hash tables `R` and `Q` are utilized by the BJT algorithm.

```
void tabulate_bjt(const int index, const long D_total, const char * file,
                  const char * folder, const int a, const int m,
                  const int * primes, int ** h_factors,
                  int ** D_factors, int * h_list)
```

Tabulates all class groups with *fundamental* negative discriminants Δ, satisfying $|\Delta| \equiv \mathtt{a} \pmod{\mathtt{m}}$ and $\texttt{index}\cdot\texttt{D\_total}\cdot\texttt{m} \le |\Delta| < (\texttt{index}+1)\cdot\texttt{D\_total}\cdot\texttt{m}$ (exclusive). The resulting text file `cl[a]mod[m].[index]` get saved into a folder `cl[a]mod[m]` created by the program inside the folder `folder`; here, `[a]`, `[m]` and `[index]` should be replaced by the values of `a`, `m` and `index`, respectively. For the format of the text file see Subsection 16. The set of primes in `primes` is used by the `mod_sieve` routine when factoring the discriminants. The two-dimensional array `h_factors` contains the list of precomputed prime divisors of *all* potential class numbers. The prime divisors of each discriminant get saved into the array `D_factors`. The binary file located in `/folder/file[index]`, where `[index]` should be replaced by the value of `index`, contains all the class numbers for a given congruence class of $|\Delta|$. If the `KEEP_FILES` macro is undefined, this file gets deleted. If `file=NULL`, i.e. the class numbers are unknown, the algorithm utilizes the `h_lower_bound` routine and computes the class group conditionally.

12 Verification

The following subroutines are implemented in `verify.c`:

```
void partial_left_hand_side(mpz_t sum, const char * file, const char * folder,
                            const int index, const long blocksize, const long x,
                            const int a, const int m, const int * primes,
                            int ** factors)
```

Partially computes the left hand side of the Eichler-Selberg trace formula (multiple of 6, to balance out fractional $H(\Delta)$), corresponding to $|\Delta| \equiv a \pmod{m}$ for $\texttt{index} \cdot \texttt{blocksize} \le |\Delta| \le (\texttt{index}+1) \cdot \texttt{blocksize}$, where $(a, m) \in \{(8,16),(4,16),(3,8),(7,8)\}$. If we set $i = \texttt{index}$ and $B = \texttt{blocksize}$, then this subroutine computes

$$S_{a,m,i,B}(X) = 6 \sum_{\substack{|\Delta| \equiv 0 \pmod{8} \\ |\Delta| \equiv a \pmod{m} \\ iB \le |\Delta| \le (i+1)B}} H(\Delta) + 12 \sum_{\substack{|\Delta| \equiv a \pmod{m} \\ iB \le |\Delta| \le (i+1)B}} r(\Delta, X) H(\Delta)$$

where

$$r(\Delta, X) = \begin{cases} 0, & \text{if } |\Delta| \equiv 3 \pmod{8}; \\ \left\lfloor \frac{Y+1}{2} \right\rfloor & \text{if } |\Delta| \equiv 7 \pmod{8}; \\ \left\lfloor \frac{Y+2}{4} \right\rfloor & \text{if } |\Delta| \equiv 4 \pmod{8}; \\ \left\lfloor \frac{Y}{4} \right\rfloor & \text{if } |\Delta| \equiv 0 \pmod{8} \end{cases}$$

and $Y = \left\lfloor \sqrt{8X + \Delta} \right\rfloor$. The result of summation gets saved into `LHS`. All Hurwitz class numbers for fundamental Δ are contained in a file `/[folder]/[file].[index]` of the format described in Subsection 15. In order to compute the Hurwitz class numbers for non-fundamental Δ, the computation of Kronecker symbols modulo various primes is required. For this purpose, the array of primes `primes` and the two-dimensional array `factors` are supplied.

```
void left_hand_side(mpz_t LHS, const long D_max, const long files,
                    const int * primes, const char * folder)
```

Computes the left hand side of the Eichler-Selberg trace formula, multiplied by 6:

$$LHS = \sum_{i=0}^{\texttt{files}-1} S_{8,16,i,B}(X) + S_{4,16,i,B}(X) + S_{3,8,i,B}(X) + S_{7,8,i,B}(X).$$

where $B = \texttt{D\_max}/\texttt{files}$. The result is saved into `LHS`. The files are located in `folder`. The tabulation upper bound is `D_max` and the total number of files for each congruence class is `files`. The array `primes` contains precomputed primes required for the computation of Kronecker symbols.

```
void partial_right_hand_side(mpz_t sum, const long blocksize, const long l,
                             const int * primes, int ** factors)
```

Partially computes the right hand side of the Eichler-Selber trace formula multiplied by 6 (to balance out the fraction occurring in $1/6\chi(2n)$):

$$R_l = 6 \sum_{n=l}^{\texttt{l}+\texttt{blocksize}-1} \left(2 \left(\sum_{\substack{d|2n \\ d \ge \sqrt{2n}}} d \right) - \chi(2n)\sqrt{2n} + \frac{1}{6}\chi(2n) \right).$$

The result is saved into `sum`. The array `primes` contains precomputed primes required for the computation of divisors of each even $2\texttt{l} \le n < 2(\texttt{l} + \texttt{blocksize} - 1)$. The two-dimensional array `factors` is utilized to store prime factors of each n.

```
void right_hand_side(mpz_t RHS, const long n_max, const long blocksize,
                     const int * primes)
```

Computes the right hand side of the Eichler-Selberg trace formula, multiplied by 6:

$$6RHS = \sum_{l=0}^{\texttt{n\_max/blocksize}-1} R_l.$$

The result is saved into RHS. The upper bound n_max should be evenly divisible by blocksize. The summation is performed block-by-block (n_max/blocksize blocks in total). The array primes contains precomputed primes, required for the computation of divisors of each even $n < 2\texttt{n\_max}$.

13 Utilizing the executable for tabulation

To run the program, use the command mpirun and through -np specify how many processors you would like to use for parallelization. The executable accepts six parameters:

- [D_max]: the tabulation upper bound;
- [files]: total number of files where the data gets saved. Must divide D_max;
- [a]: the congruence class of $|\Delta|$ modulo m. Has to be either 4, 8 when m $= 16$ or 3, 7 when m $= 8$;
- [m]: the modulus, either 8 or 16. Must evenly divide D_max/files;
- [h_prefix]: the prefix of the binary files containing class numbers for a specific congruence class. Set to null if the class numbers are unknown;
- [folder]: the folder where the data gets saved.

The parameters should be supplied in the following order:

```
mpirun -np [procs] ./clgrp [D_max] [files] [a] [m] [h_prefix] [folder]
```

14 Utilizing the executable for verification

To produce the executable, please use the make verify command. The verify program accepts three command line parameters:

- [D_max]: the tabulation upper bound;
- [files]: total number of files to which the data gets saved. Must divide D_max;
- [folder]: the folder where the data gets saved.

The parameters should be supplied in the following order:

```
./verify [D_max] [files] [folder]
```

Note that in order for the program to work correctly, the folder folder should contain four folders cl8mod16, cl4mod16, cl3mod8 and cl7mod8, which in turn contain the result of tabulation for a specific congruence class, distributed over files files.

15 File format

All tabulated class groups get saved into a text file, compressed with `gzip`. The name of each file is of the form `cl[a]mod[m].[index]`, where `[a]` is the congruence class of $|\Delta|$ modulo `[m]`, `[m]` is the modulus, and `[index]` is the index of a file. For each file, it is important to determine the starting discriminant `D_start`. To do this, you need to know the value `D_total`, which satisfies $\texttt{index} \cdot \texttt{D\_total} \leq |\Delta| < (\texttt{index} + 1) \cdot \texttt{D\_total}$. Note that $\texttt{D\_total} = \texttt{D\_max}/\texttt{files}$, where $\texttt{D\_max} = |\Delta_{\max}|$ is the tabulation upper bound, and `files` is the total number of files among which the tabulated class groups get distributed (so $0 \leq \texttt{index} < \texttt{files}$). In the end, we have $\texttt{D\_start} = \texttt{a} + \texttt{index} \cdot \texttt{D\_total}$.

The output text file has the following format (description taken from [LMFDB]):

- There is one line per field;
- Fundamental discriminants for a given file are listed in order (in absolute value);
- If $\Delta_i = -d_i$ is the i-th discriminant of a file, line $i + 1$ has the form

  $$a \quad b \quad c_1 c_2 \ldots c_t$$

  to signify that

  - $d_{i+1} = d_i + a \cdot m$ (m is the modulus for the file);
  - $h(-d_{i+1}) = b$;
  - invariant factors for the class group are $[c_1, c_2, \ldots, c_t]$.

  In particular, $b = \prod_{j=1}^{t} c_j$.

For example, the first 10 lines of the file `cl8mod16.0` ($\texttt{a} = 8$, $\texttt{m} = 16$, $\texttt{index} = 0$), downloaded from [LMFDB], can be translated as follows:

			$\|\Delta\|$	$h(\Delta)$	$Cl(\Delta)$
0	1	1	$8 = 8 + 0 \cdot 16 + 0 \cdot 2^{28}$	1	C_1
1	2	2	$24 = 8 + 1 \cdot 16$	2	C_2
1	2	2	$40 = 24 + 1 \cdot 16$	2	C_2
1	4	4	$56 = 40 + 1 \cdot 16$	4	C_4
2	2	2	$88 = 56 + 2 \cdot 16$	2	C_2
1	6	6	$104 = 88 + 1 \cdot 16$	6	C_6
1	4	2 2	$120 = 104 + 1 \cdot 16$	4	$C_2 \times C_2$
1	4	4	$136 = 120 + 1 \cdot 16$	4	C_4
1	6	6	$152 = 136 + 1 \cdot 16$	6	C_6
1	4	2 2	$168 = 152 + 1 \cdot 16$	4	$C_2 \times C_2$
	...			...	

In the first line of the table on the right, we have $\texttt{D\_total} = \texttt{D\_max}/\texttt{files} = 2^{40}/2^{12} = 2^{28}$.

16 Examples

Example 1. The following command tabulates all class groups with $|\Delta| \equiv 8 \pmod{16}$ and $|\Delta| < 2^{40} = 1099511627776$. The result gets saved into $2^{12} = 4096$ files `/home/cl8mod16/cl8mod16.0`, ..., `/home/cl8mod16/cl8mod16.4095`. The files `/home/h8mod16/h8mod16.0`, ..., `/home/h8mod16/h8mod16.4095` contain $h(\Delta)$ for $|\Delta| \equiv 8 \pmod{16}$. There are 256 processors utilized by OpenMPI for parallelization.

```
mpirun -np 256 ./clgrp 1099511627776 4096 8 16 h8mod16/h8mod16. /home
```

Example 2. The following command tabulates all class groups with $|\Delta| \equiv 4 \pmod{16}$ and $|\Delta| < 2^{40} = 1099511627776$. The result gets saved into $2^{12} = 4096$ files `/home/cl4mod16/cl4mod16.0`, ..., `/home/cl4mod16/cl4mod16.4095`. The files `/home/h4mod16/h4mod16.0`, ..., `/home/h4mod16/h4mod16.4095` contain multiples of 2 of $h(\Delta)$ for $|\Delta| \equiv 4 \pmod{16}$. There are 256 processors utilized by OpenMPI for parallelization.

```
mpirun -np 256 ./clgrp 1099511627776 4096 4 16 h4mod16/h4mod16. /home
```

Example 3. The following command tabulates all class groups with $|\Delta| \equiv 3 \pmod{8}$ and $|\Delta| < 2^{40} = 1099511627776$. The result gets saved into $2^{12} = 4096$ files `/home/cl3mod8/cl3mod8.0`, ..., `/home/cl3mod8/cl3mod8.409` The files `/home/h3mod8/h3mod8.0`, ..., `/home/h3mod8/h3mod8.4095` contain multiples of 3 of $h(\Delta)$ for $|\Delta| \equiv 4 \pmod{16}$. There are 256 processors utilized by OpenMPI for parallelization.

```
mpirun -np 256 ./clgrp 1099511627776 4096 3 8 h3mod8/h3mod8. /home
```

Example 4. The following command *conditionally* tabulates all class groups with $|\Delta| \equiv 7 \pmod{8}$ and $|\Delta| < 2^{40} = 1099511627776$. The result gets saved into $2^{12} = 4096$ files `/home/cl7mod8/cl7mod8.0`, ..., `/home/cl7mod8/cl7mod8.4095`. The `null` parameter indicates that the class numbers for this congruence class are unknown. There are 256 processors utilized by OpenMPI for parallelization.

```
mpirun -np 256 ./clgrp 1099511627776 4096 7 8 null /home
```

Example 5. The following command verifies the class group tabulation data up to $|\Delta| < 2^{40} = 1099511627776$. The data for each congruence class is distributed over $2^{12} = 4096$ files.

```
./verify 1099511627776 4096 /home
```

17 Utilizing the library

In order to utilize the library, place the files `clgrp.h`, `functions.h`, `sieve.h` and `verify.h` into your include path, and copy the library `libclgrp.a` into your library path. By writing

```
#include <clgrp.h>
```

among other inclusions in your file you will gain access to all the subroutines defined in `clgrp.h`. Same applies to other files mentioned above.

When compiling, link the library to your program by writing `-lclgrp`. Don't forget to link all the other libraries that CLGRP depends on (see Subsection 3 for the complete list).

References

[Bac90] E. Bach, *Explicit bounds for primality testing and related problems*, Computation 55, pp. 355 – 380, 1990.

[BJT97] J. Buchmann, M. J. Jacobson, Jr., E. Teske, *On some computational problems in finite abelian groups*, Mathematics of Computation 66 (220), pp. 1663 – 1687, 1997.

[GG03] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 2nd edition, 2003.

[HTW10] W. B. Hart, G. Tornaría, M. Watkins, *Congruent number theta coefficients to* 10^{12}, Algorithmic Number Theory — ANTS-IX (Nancy, France), Lecture Notes in Computer Science 6197, Springer-Verlag, Berlin, pp. 186 – 200, 2010.

[JRW06] M. J. Jacobson, Jr., S. Ramachandran, H. C. Williams, *Numerical results on class groups of imaginary quadratic fields*, Algorithmic Number Theory — ANTS-VII (Berlin, Germany), Lecture Notes in Computer Science 4076, Springer-Verlag, Berlin, pp. 87 – 101, 2006.

[LMFDB] Linear and Modular Forms Database, *Class Groups of Quadratic Imaginary Fields.* http://beta.lmfdb.org/NumberField/QuadraticImaginaryClassGroups, 2015.

[Mos14] A. S. Mosunov, *Unconditional Class Group Tabulation to* 2^{40}, Master's thesis, University of Calgary, Calgary, Alberta, 2014.

[Say13a] M. Sayles, *Improved arithmetic in the ideal class group of imaginary quadratic fields with an application to integer factoring*, Master's thesis, University of Calgary, Calgary, Alberta, 2013.

[SvdV91] R. Schoof, M. van der Vlugt, *Hecke operators and the weight distributions of certain codes*, Journal of Combinatorial Theory 57, pp. 163 – 186, 1991.

[Wes14] Hungabee specification, https://www.westgrid.ca/support/systems/Hungabee, 2014.