# Table of Contents

1 Introduction ................................................................. 1

2 Installation ................................................................. 2
   2.1 Instructions ............................................................ 2
   2.2 Documentation ........................................................ 2
   2.3 Data ................................................................. 3

3 Library ................................................................. 4
   3.1 CM parameters ......................................................... 4
   3.2 Computing class polynomials ........................................... 6
   3.3 Computing CM elliptic curves .......................................... 7
   3.4 Elliptic Curve Primality Proofs (ECPP) .................................. 8

4 Applications ................................................................. 9
   4.1 classpol ............................................................ 9
   4.2 cm .......................................................... 10
   4.3 eccp .......................................................... 11

References ............................................................ 12

Index ........................................................... 13
1 Introduction

The CM software implements the construction of ring class fields of imaginary quadratic number fields and of elliptic curves with complex multiplication via floating point approximations. It consists of a library that can be called from within a C program and of executable command line applications. For the implemented algorithms, see [Enge09], page 12.

Given an imaginary quadratic discriminant $D < 0$, the associated ring class field is generated by the values of modular functions in special arguments taken from the quadratic field $\mathbb{Q}(\sqrt{D})$; these values are called singular values or class invariants. Depending on $D$, different modular functions need to be chosen; we call the suitable ones class functions. CM implements (to a greater or lesser extent) all major class invariants described in the literature.

Licence

CM is free software; you can redistribute it and/or modify it under the terms of the GNU General Public licence as published by the Free Software Foundation; either version 3 of the licence, or (at your option) any later version.

CM is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public licence for more details.

You should have received a copy of the GNU General Public licence along with CM; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
2 Installation

2.1 Instructions

CM relies on a number of external libraries, which need to be installed before compiling CM: GNU MP (see [Gretal20], page 12, version 4.3.2 or higher), GNU MPFR (see [HaLePeZi20], page 12, version 3.0.0 or higher), GNU MPC (see [EnGaThZi20], page 12, version 1.0.0 or higher), MPFRCX (see [Enge21], page 12, version 0.6.3 or higher) and PARI/GP (see [Pari21], page 12, version 2.11.0 or higher, compiled with GMP as the arithmetic kernel). Compilation of CM needs a standards compliant C compiler (preferably GCC).

These are the steps needed to install CM, provided that the required libraries are installed in standard locations:

1. 'tar xzf cm-0.4.0.tar.gz'
2. 'cd cm-0.4.0'
3. './configure'
4. 'make'
   This compiles CM.
5. 'make check'
   This performs a few tests to check that CM has been built correctly.
   If you get error messages, please report them to the author.
6. 'make install'
   This copies the executable applications into the directory /usr/local/bin, the header files into /usr/local/include, the library files into /usr/local/lib, the data files into subdirectories of /usr/local/share/cm (see Section 2.3 [Data], page 3) and the file cm.info into /usr/local/share/info.

   It is possible to pass the option ' --prefix=/my/install/directory' to the './configure' step above, so that all files go into subdirectories of /my/install/directory instead of /usr/local.

If auxiliary libraries are to be found in non-standard locations, these need to be passed in the './configure' step above by adding parameters

- ' --with-gmp=<gmp_install_dir>',
- ' --with-mpfr=<mpfr_install_dir>',
- ' --with-mpc=<mpc_install_dir>',
- ' --with-mpfrcx=<mpfrcx_install_dir>' and
- ' --with-pari=<pari_install_dir>'.

If you wish to compile the parallel, MPI version ecpp-mpi of the ecpp binary for elliptic curve primality proofs, you need to pass the option ' --enable-mpi' (and, of course, have an MPI library installed against which the binary will be compiled and linked).

For an exhaustive list of configuration parameters, execute './configure --help'.

2.2 Documentation

Besides the texinfo documentation obtained by a simple invocation of 'make', the commands 'make dvi', 'make ps', 'make pdf' and 'make html' create the documentation in the corresponding formats.
2.3 Data

Some parameterised families of class functions need additional data (namely, modular polynomials), which depend on the parameter value, to deduce the equation of an elliptic curve from the class polynomial. A few modular polynomials are provided and stored in subdirectories of /usr/local/share/cm (or of the subdirectory share/cm of the installation directory provided with the '--prefix' configuration option, respectively). More precisely, these bivariate polynomials relate the class function with the modular function \( j \); instantiating in a class invariant makes it possible to compute the \( j \)-invariant of a corresponding elliptic curve as a root of the modular polynomial.

An infinite amount of data is needed to handle all possible discriminants with a given family of class functions, and already covering all moderately sized discriminants would require gigabytes of data. So only a very small sample of modular polynomials is currently distributed; if you need more, please write to the author.

More precisely, the subdirectory /usr/local/share/cm/df contains all modular polynomials for double \( \eta \)-quotients that are of the minimally possible degree 2 in \( j \); the subdirectory /usr/local/share/cm/af contains all modular polynomials for Atkin functions of degree 2 in \( j \). All these are functions of some level \( N \), invariant under the Fricke-Atkin-Lehner involution, for which the modular curve \( X^{+}_0(N) \) is of genus 0. As a consequence, both roots in \( j \) of the instantiated modular polynomial yield a suitable CM elliptic curve. Finally the subdirectory /usr/local/share/cm/mf contains all modular polynomials for triple \( \eta \)-quotients of the minimal degree 4 or of degree 8 in \( j \).
3 Library

The code of CM comes first and foremost as a C library making its functionalities accessible to external applications. The names of all publicly accessible, non-static functions and types start by cm_ to create a name space proper to CM.

If you are not interested in programming in library mode, the project also comes with a few sample applications described in more detail in Chapter 4 [Applications], page 9; all of them are implemented with just a few library calls, so describing the library first makes it easier to give the parameter choices for the applications, and enables you to easily create small modifications.

Public constants, types and functions are defined in the file cm.h; the exact composition of the types is not important to document here, since they are usually initialised by calls to dedicated functions and then passed to further functions needing them. In general, the usual trick known from GMP is applied, that is, the types are one-dimensional arrays of structs, so that the difference between passing arguments by value or by reference disappears and it is usually not necessary to use the & and * operators for referencing and dereferencing. Also in GMP style, parameters modified by functions are usually passed first.

3.1 CM parameters

The cm_param_t type holds the main parameters fixed before computing the class field of an imaginary-quadratic order. This is first and foremost the quadratic discriminant $D < 0$. For most applications, it will be enough to consider fundamental discriminants, but this is not a requirement of the code, so all quadratic discriminants are accepted; if a non-fundamental discriminant is provided, the corresponding ring class field is computed. The second main parameter provides the type of class function used, either a single function or one out of a parameterised family; in the latter case also these parameter values are stored.

int_cl_t

This signed 64 bit integer type is used for discriminants.

cm_param_t

This type, used for holding the main parameters of a CM setting, is defined using the GMP trick of a one-dimensional array of a struct. There should be no need to manipulate its fields directly.

bool cm_param_init (cm_param_t param, int_cl_t d, char invariant, int maxdeg, int subfield, bool verbose)

This function initialises the param object depending on the main input d, a negative quadratic discriminant, and invariant, one of the following constants describing a class function or a parameterised family of class functions. Every class function has a height factor associated to it, which indicates by how much (asymptotically for $|D| \to \infty$) the number of digits of the largest coefficient, or equivalently the precision required for the floating point approximations, is smaller than for the $j$-function; the latter works for every discriminant, but leads to the largest class polynomials. On the other hand, alternative class functions work only for a restricted class of discriminants each.

- **CM_INVARIANT_J**: The modular function $j$ with height factor 1 by definition.
- **CM_INVARIANT_GAMMA3**: The modular function $\sqrt{d} \gamma_3$, where $\gamma_3$ is a square root of $j - 1728$. The additional factor of $\sqrt{d}$ is needed only to make the class polynomial real. Its height factor is 2, and it works whenever $D$ is odd.
- **CM_INVARIANT_GAMMA2**: The modular function $\gamma_2$, a cube root of $j$. Its height factor is 3, and it works whenever $D$ is not divisible by 3.
The level of the function is essentially the discriminant (or the simple \( \eta \)-quotient is a hauptmodul), so that a CM elliptic curve may be deduced without the use of a modular polynomial; that is, \( N \in \{3, 5, 7, 13, 4, 9, 25\} \). The height factor is between 2 and 24.

CM_INVARIANT_DOUBLEETA: Double \( \eta \)-quotients of the form \( \eta(z)/\eta(z/N) \) for an integer \( N \) and their powers (to an exponent that divides 24), see [EnSc04], page 12. The code implements only prime power levels \( N \) for which \( X_0(N) \) is of genus 0 (and the simple \( \eta \)-quotient is a hauptmodul), so that a CM elliptic curve may be deduced without the use of a modular polynomial; that is, \( N \in \{3, 5, 7, 13, 4, 9, 25\} \). The height factor is between 2 and 24.

CM_INVARIANT_MULTIETA: Multiple \( \eta \)-quotients for \( k \) primes \( p_1, \ldots, p_k \) of level \( N = p_1 \cdots p_k \); this is the quotient of two products of \( \eta(z/n) \), where \( n \) varies over the \( 2^k \) divisors of \( n \), and the \( n \) with an odd number of primes appear in the numerator, those with an even number of primes in the denominator, see [EnSc13], page 12. The code is implemented generically, but currently only triple \( \eta \)-quotients (with \( k = 3 \)) are actually used.

CM_INVARIANT_ATKIN: Functions for \( X_0^+(N) \) of genus 0 for a prime level \( N \in \{47, 59, 71, 131\} \); the functions are optimal in the sense that they have a pole of lowest degree at the cusp for a given family. The height factor is between 24 and 36. On the other hand, this finite family of class functions is obtained by applying certain Hecke operators to \( \eta(z)\eta(Nz) \), and the numerical evaluation of these Hecke operators is costly.

For families of class functions, the function selects the admissible parameter combination yielding the smallest class polynomial. Admissibility depends mainly on the discriminant (or more precisely, on the splitting behaviour of the primes dividing the level \( N \) in \( Q(\sqrt{D}) \)), but also on the values of the further arguments to the function.

The parameter \texttt{maxdeg} sets an upper limit on the degree in \( j \) of the modular polynomial; it has an effect only for infinite families of class functions, that is, when \texttt{variant} is \texttt{CM_INVARIANT_DOUBLEETA} or \texttt{CM_INVARIANT_MULTIETA}. When set to 0, it is disabled; when set to -1, it is internally set to the degree for which modular polynomials are distributed, which makes it possible to derive a CM elliptic curve from the class polynomial.

Double and multiple \( \eta \)-quotients are known to generate strict subfields of the class field in some cases, see [EnSc13], page 12. Whether this is admitted depends on the value of \texttt{subfield}, which can take the following constants:

- **CM_SUBFIELD_NEVER**: Do not choose parameters known to generate subfields. This may still happen by chance (and break the computation of a Galois tower decomposition). This should be chosen to obtain a generator of the class field.
- **CM_SUBFIELD_PREFERRED**: Whenever possible, compute a subfield of the class field, and if several choices are possible, prefer the one with the biggest index. This speeds up the computation of elliptic curve primality proofs (ECPP), where finding a root of the class polynomial becomes one of the bottlenecks for large input.
- **CM_SUBFIELD_OPTIMAL**: Compute the field with the smallest class polynomial, regardless of its degree. This will often be a subfield, if available, since the index of the subfield
lowers the size of the class polynomial by a quadratic factor, acting on the degree of the polynomial and on the size of its coefficients. This is intended to yield optimal speed.

If \texttt{verbose} is set to \texttt{true}, some information is printed on screen during execution.

If an admissible parameter combination is found, the function stores it in \texttt{param} and returns \texttt{true}; otherwise it returns \texttt{false}.

Since the function does not allocate any memory, there is no corresponding function \texttt{cm_param_clear}.

### 3.2 Computing class polynomials

\texttt{cm_class_t} \hspace{1cm} [Type]

This type is also implemented as a one-dimensional array of a struct, but there should not be any need to access its fields. It stores information about the ring class group and, once computed, the class polynomial or its decomposition as a tower of Galois fields.

The code for computing class polynomials relies on the PARI library, which needs to be initialised before calling any of its functions. While it would be possible to hide this initialisation from the user (inside \texttt{cm_class_init}, for instance), this would make it more difficult to mix CM code with code that uses the PARI library for other purposes. So there are special functions for initialising and clearing the PARI library.

\texttt{void cm_pari_init ()} \hspace{1cm} [Function]

This function must be called before any other function operating on class polynomials. Essentially it encapsulates a call to \texttt{pari_init}.

\texttt{void cm_pari_clear ()} \hspace{1cm} [Function]

This function should be called at the end of working with the CM library. Essentially it encapsulates a call to \texttt{pari_close}.

\texttt{void cm_class_init (cm_class_t c, cm_param_t param, bool verbose)} \hspace{1cm} [Function]

This function should be called after \texttt{cm_pari_init}. Given a CM parameter \texttt{param} initialised with a call to \texttt{cm_param_init}, it allocates memory and executes some fast precomputations (such as the class group), the results of which are stored in \texttt{c}.

If \texttt{verbose} is set to \texttt{true}, some information is printed on screen during execution.

\texttt{void cm_class_clear (cm_class_t *c)} \hspace{1cm} [Function]

This is the counterpart to \texttt{cm_class_init} and should be called once the class polynomial is not needed any more to free the allocated space.

\texttt{void cm_class_compute (cm_class_t c, cm_param_t param, bool classpol, bool tower, bool verbose)} \hspace{1cm} [Function]

Given a previously initialised \texttt{cm_class_t} object \texttt{c} and corresponding parameter object \texttt{param}, the function computes the class polynomial and stores it in \texttt{c}.

More precisely, if \texttt{classpol} is set to \texttt{true}, the function computes the class polynomial in one variable \( X \) defining the class field; if \texttt{tower} is set to \texttt{true}, it computes the same class field as a tower of relative extensions of prime degree using the asymptotically fast algorithms of see [EnMo03], page 12. Otherwise said, it computes a univariate polynomial \( f_1 \) in the variable \( X_1 \) defining an absolute extension \( K_1/Q \) (or \( K_1/Q(\sqrt{D}) \), see below), then a bivariate polynomial \( f_2 \) in \( X_1 \) and \( X_2 \) defining a relative extension \( K_2/K_1 \), and so on. For the function to have
any effect, at least one of classpol or tower needs to be set to true, and usually exactly one is enough.

The class polynomial (or the bivariate polynomials defining a tower of Galois extensions) are either real, that is, they have coefficients in \( \mathbb{Z} \); or they are complex, that is, they have coefficients in the ring of integers of the imaginary-quadratic field \( \mathbb{Q}(\sqrt{D}) \). We write the latter using the standard basis as \( \mathbb{Z} + \mathbb{Z} \omega \), where \( D \) is the fundamental discriminant attached to \( D \) and \( \omega = \sqrt{D}/2 \) if \( D \) is even and \( \omega = (1 + \sqrt{D})/2 \) if \( D \) is odd.

If verbose is set to true, some information is printed on screen during execution.

```c
void cm_class_print_pari (FILE* file, cm_class* c, char* fun, char* var)
```

Print the computed class polynomial or the relative polynomials defining a number field tower from \( c \) to \( file \) (which may be stdout, for instance) in a format that can be copy-pasted or loaded into PARI/GP. The arguments \( fun \) and \( var \) define the function and variable names used; if set to NULL, default values are given.

If computed, the class polynomial is printed with \( fun \) (default f) as the name of the polynomial in the variable \( var \) (default x); if it is complex, then the second basis element \( \omega \) is abbreviated to o.

If computed, the class polynomial tower is printed with \( fun \) as the name of the \( i \)-th polynomial in \( var \).

### 3.3 Computing CM elliptic curves

After computing a class field by a call to \texttt{cm_class_compute} (either through a class polynomial as an absolute extension or as a tower of relative extensions, or both), it can be used to derive a CM elliptic curve over a finite field. The CM library implements the computation of curves over prime fields, and also returns a point of prime order on such a curve.

```c
void cm_curve_and_point (mpz_t a, mpz_t b, mpz_t x, mpz_t y, cm_param_t param, cm_class_t c, mpz_t p, mpz_t l, mpz_t co, const char* modpoldir, bool print, bool verbose)
```

Let \( c \) contain a computed class polynomial or the relative polynomials defining a number field tower from \( c \) to \( file \) (which may be stdout, for instance) in a format that can be copy-pasted or loaded into PARI/GP. The arguments \( fun \) and \( var \) define the function and variable names used; if set to NULL, default values are given.

If computed, the class polynomial is printed with \( fun \) (default f) as the name of the polynomial in the variable \( var \) (default x); if it is complex, then the second basis element \( \omega \) is abbreviated to o.

If computed, the class polynomial tower is printed with \( fun \), as the name of the \( i \)-th polynomial in \( var \).

```c
void cm_curve_crypto_param (mpz_t p, mpz_t n, mpz_t l, mpz_t co, int cl_t d, int fieldsize, bool verbose)
```

This function computes field parameters for a CM curve over a finite field for the discriminant \( d \) that satisfies the following conditions for use in an elliptic curve cryptosystem; notice that due to their special nature, CM curves are not recommended for cryptography, and that further security considerations should be taken into account. As such, the function is mainly
intended to test \texttt{cm\_curve\_and\_point}. Besides the discriminant it takes \texttt{fieldsize} and outputs in \(p\) the characteristic of a finite prime field with \texttt{fieldsize} bits such that there is an elliptic curve over \(F_p\) with \(n=l\cdot co\) points such that \(l\) is a (pseudo-)prime and \(co\in \{1, 2, 4, 8\}\) is the minimal cofactor reachable with this discriminant.

If \texttt{verbose} is set to \texttt{true}, some information is printed on screen during execution.

### 3.4 Elliptic Curve Primality Proofs (ECPP)

Primality proving with elliptic curves (ECPP) is one of the main applications of CM elliptic curves over finite fields. The CM library implements the asymptotically fast version \texttt{fastECPP} of [FrKIMoWi04], page 12, and [Morain07], page 12.

Certificates are computed and printed in a format compatible with PARI/GP.

\begin{verbatim}
void cm_ecpp (mpz_t N, const char* modpoldir, bool pari, bool tower, bool print, bool check, bool verbose, bool debug)
\end{verbatim}

Given a prime number \(N\), the function computes an ECPP certificate and prints it to screen if \texttt{print} is set to \texttt{true}. If \texttt{check} is set to \texttt{true}, then the certificate is checked; the outcome is printed if \texttt{verbose} is also set to \texttt{true}. The directory \texttt{modpoldir}, usually \texttt{/usr/local/share/cm}, in which modular polynomials are stored, is required to be passed to \texttt{cm\_curve\_and\_point}.

If set to \texttt{true}, the argument \texttt{pari} indicates that the first step of the algorithm, instead of being executed with code of the CM library, is to be taken from a separate implementation copied from PARI/GP version 2.11. This is obsolete and only useful for comparison purposes; \texttt{false} should be used instead.

The argument \texttt{tower}, if set to the recommended value \texttt{true}, indicates that class fields should not be given as absolute extensions, but as a tower of relative extensions. This takes negligibly more time for the CM computations, but can save a lot of time during the construction of the elliptic curves in the certificate, since the computation of a root of one high degree polynomial over a finite field is replaced by a series of roots of smaller degree polynomials.

If \texttt{verbose} is set to \texttt{true}, some information is printed on screen during execution. In particular, if both \texttt{print} and \texttt{verbose} are set to \texttt{false}, the function has no visible effect.

If additionally to \texttt{verbose}, \texttt{debug} is set to \texttt{true}, more information, in particular on timing of different steps of the ECPP algorithm, which is useful only for debugging purposes, is printed on screen. This option should become obsolete once the ECPP code becomes less experimental.
4 Applications

CM comes with a few applications: \texttt{classpol} for computing class polynomials and \texttt{cm} for computing a CM elliptic curve that could be used for cryptography, and \texttt{ecpp} for performing elliptic curve primality proofs. The first two compute one class field or CM elliptic curve and share a certain number of command line arguments; the third one currently takes no command line arguments. All of them are implemented with only a few calls to library functions. The following sections present the functionality of each application and the command line arguments it takes, and also reproduce the essential part of its code to further illustrate the library interface.

4.1 \texttt{classpol}

The \texttt{classpol} application takes one mandatory argument, \texttt{-d} followed by the absolute value $|D|$ of the discriminant. It computes and prints on screen the class polynomial for $D$ obtained using the $j$-invariant. The additional parameter \texttt{-v} enables more verbose output for the different steps of the algorithm.

\begin{verbatim}
$ classpol -d 207
f = x^6+42653766018394018375*x^5-5002547112103664005187500*x^4
+1819343755841564610147379736328125*x^3
-21067210985246065248197114115955810546875*x^2
+12041028291910181818274355885092809398864746093750*x
-183426864580818496179793649372867188930511474609375

Class polynomials for alternative class invariants are selected using the \texttt{-i} argument followed by the type of class function; this is the same as the library constants given in Section 3.1 [CM parameters], page 4, with the prefix \texttt{CM_INVARIANT_} left out and transformed to lower case; so \texttt{CM_INVARIANT_WEBER} becomes \texttt{-iweber}, and so on.

$ classpol -d 207 -i doubleeta -v
Discriminant 207, fundamental discriminant 23
Invariant d, parameter 2_73_1_1
Class number 6

f = x^6-6*x^5+16*x^4-22*x^3+16*x^2-5*x+1
\end{verbatim}

The verbose output tells us that -207 is not a fundamental discriminant, but a multiple of the fundamental discriminant -23, and that the double $\eta$-quotient used is of level 2 $\cdot$ 73 raised to the power 1 (the second 1 is a technical parameter, indicating the maximal power that might be needed for this level).

After evaluating the command line parameters, the essential part of the code implementing this functionality takes less than ten lines:

\begin{verbatim}
cm_pari_init ();
if (!cm_param_init (param, d, invariant, 0, CM_SUBFIELD_NEVER, verbose))
    exit (1);
cm_class_init (c, param, verbose);
cm_class_compute (c, param, true, false, verbose);
cm_class_print_pari (stdout, c, NULL, NULL, NULL);
cm_class_clear (c);
cm_pari_clear ();
\end{verbatim}

The call to \texttt{cm_param_init} initialises the variable \texttt{param} by checking whether the invariant is suitable for the given discriminants and choosing a class function if this is the case (such as
the $\eta$-quotient of level 39 in the example above). Assuming that the goal is to compute a class field, parameter combinations known to lead to a strict subfield are excluded by the constant \texttt{CM\_SUBFIELD\_NEVER}. The 0 indicates that the degree of modular polynomials may be arbitrarily high, since they would only need to derive CM elliptic curves.

The calls to \texttt{cm\_pari\_init} and \texttt{cm\_class\_init} and their respective clear counterparts reserve and free memory and carry out relatively fast precomputations. The main activity is launched through the call to \texttt{cm\_class\_compute}, in which the boolean values indicate that an absolute class polynomial is to be computed and not a tower of relative extensions. The result is output by a call to \texttt{cm\_class\_print\_pari}.

### 4.2 cm

The \texttt{cm} application takes the same command line arguments \texttt{-d}, \texttt{-i} and \texttt{-v} as \texttt{classpol}. See Section 4.1 [\texttt{classpol}], page 9. It computes a CM curve for the discriminant $D$ over a prime field of 256 bits such that its cardinality is “as prime as possible”, that is prime up to possibly a factor of 2, 4 or 8 depending on $D$. The \texttt{-v} parameter is needed to print the computed parameters on screen.

\begin{verbatim}
cm -d 207 -i doubleeta -v
Invariant d, parameter 2_13_2_2
... p = 11579208923731619538462578067141184801329650642019283009460547375490535224057 n = 4 * 2894802209329048949615644516785296200162271477044351520651896284230459251901 a = 9816321418547049705083700609726438077908546912553098655936731136039029295843 b = 3228512507898029771281739302267197783607850091512599038419136678099671035526 x = 1033976445671971353096333781717451285890560566736753198896456566993433387644320 y = 72297811879224681619558031933509767890723605942283336960075126018884088267112
The curve has equation $E : Y^2 = X^3 + aX + b$ over the finite prime field $F_p$, and its cardinality $n$ is 4 times a prime (for a prime cardinality, the discriminant must satisfy $D \equiv 5 \pmod{8}$).

The point of prime order $n/4$ is given by $(x, y)$.

Besides the same calls to functions initialising and clearing data, the core of the implementation is as follows:

```c
if (!cm_param_init (param, d, invariant, -1, CM\_SUBFIELD\_OPTIMAL, verbose))
    exit (1);

cm_class_compute (c, param, false, true, verbose);

cm_curve_crypto_param (p, n, l, co, d, 256, verbose);

cm_curve_and_point (a, b, x, y, param, c, p, l, co, CM\_MODPOLDIR,
    true, verbose);
```

Parameter initialisation uses the arguments \texttt{CM\_SUBFIELD\_OPTIMAL} to indicate that the class invariant expected to be computed in the fastest time should be used, independently of it leading to a subfield or the full class field; and \texttt{-1} to limit the search for class invariants for which modular polynomials are available. (The verbose output shows that the $\eta$-quotient of level 2-73 is replaced by the quotient of level 2-13 raised to the power 2, which is expected to yield a larger class polynomial, but for which the modular polynomial is distributed with the code).

The boolean arguments to \texttt{cm\_class\_compute} lead to the computation of the tower decomposition of the class field instead of the full class polynomial. The call to \texttt{cm\_curve\_crypto\_param} determines a suitable 256 bit prime and curve cardinality $n$, and the call to \texttt{cm\_curve\_and\_point} obtains and prints the coordinates of a point of prime order $l$ on the curve.
4.3 ecpp

The ecpp application computes an ECPP certificate for proving the primality of the number passed with the -n command line argument. It is assumed that this number is a suitably tested pseudo-prime; in particular, if it is smaller than $2^{64}$ then it is assumed to be prime, and no certificate is computed. The number can simply be given in decimal notation, but also by an arbitrary GP expression.

The argument -p causes printing of the certificate on screen, -v enables printing of progress information, and -g enables even more verbose debug output.

```bash
$ ecpp -n 'nextprime(10^31)' -p
```

```text
[[10000000000000000000000000000033, -5882759018432034, 12103604, 25,
[38768651611416530802393519623, 7268598020338906447634857503614]],
[826200196239071096737717, 667597927066, 916, 0,
[37611257001838975439341, 115218092585553575608847]],
[901965279736248361147, 54401389280, 118118316, 0,
[28262866493744390372, 352399418333719760852]];
```

The resulting line defining the certificate can be copy-pasted into a PARI/GP session and checked using

```text
? primecertisvalid(c)
```

with expected result 1.

Alternatively, the argument -c checks the validity of the computed certificate; to see the result, the option -v needs to be enabled.

The argument -f filename stores the computed certificate in a file of the given name. Additionally, it enables checkpointing: During the first phase of ECPP (the downrun step determining cardinalities of elliptic curves leading to a primality proof), the file filename.cert1 is written, during the second ECPP phase of computing the elliptic curves by complex multiplication, the file filename.cert2 is written. Upon restart, the program picks up these files and continues where it has been interrupted. After a successful run, these checkpointing files may be deleted.

Checkpointing is particularly useful with the MPI based parallel version of the binary, called ecpp-mpi; this is created and installed alongside the sequential ecpp binary when the --enable-mpi configure option is given.

Thus in a suitably set up MPI environment,

```bash
$ mpirun ecpp-mpi -g -n '10^1000+453' -c -f cert-1000
```

computes and checks an ECPP certificate for the first prime with 1001 digits and stores it into the file cert-1000, while outputting debug information on screen.
References

Index

A
applications ........................................... 9

C
class function ........................................ 1
class invariant ....................................... 1
class polynomial ..................................... 6
classpol ............................................... 9
cm ..................................................... 10
cm_class_clear ...................................... 6
cm_class_compute .................................. 6
cm_class_init ........................................ 6
cm_class_print_pari ................................. 7
cm_class_t ........................................... 6
cm_curve_and_point ................................ 7
cm_curve_crypto_param ............................. 7
cm_ecpp .............................................. 8
cm_param_init ...................................... 4
cm_param_t .......................................... 4
cm_pari_clear ...................................... 6
cm_pari_init ........................................ 6
CM curve ............................................. 7
copying conditions .................................. 1

D
data ..................................................... 3
documentation ........................................ 2

E
ecpp ................................................... 11
ECPP .................................................. 8

H
height factor .......................................... 4

I
installation ........................................... 2
int_cl_t .............................................. 4
introduction ......................................... 1

L
library ................................................ 4
licence ............................................... 1

M
modular polynomial .................................. 3

P
parameters ........................................... 4
primality proof ....................................... 8

S
singular value ........................................ 1