CM

Complex Multiplication
Version 0.4.3
February 2024

Andreas Enge andreas.enge@inria.fr

Table of Contents

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 13.

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 $Q(\sqrt{D})$; 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

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 13, version 4.3.2 or higher), GNU MPFR (see [HaLePeZi20], page 13, version 3.0.0 or higher), GNU MPC (see [EnGaThZi20], page 13, version 1.0.0 or higher), MPFRCX (see [Enge21], page 13, version 0.6.3 or higher) and PARI/GP (see [Pari], page 13, version 2.11.0 or higher, compiled with GMP as the arithmetic kernel). Additionally, FLINT (see [Flint], page 13) can be used if it is available, but it is not mandatory. 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-0.4.3.tar.gz`'
2. '`cd cm-0.4.3`'
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>`'
- '`--with-pari=<pari_install_dir>`'. and
- '`--with-flint=<flint_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 η-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 η-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` [Type]
This signed 64 bit integer type is used for discriminants.

`cm_param_t` [Type]
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`) [Function]
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 γ_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 γ_2, a cube root of j. Its height factor is 3, and it works whenever D is not divisible by 3.

- `CM_INVARIANT_WEBER`: One of the Weber functions f, f_1 or f_2 or its powers. This is a finite family of class functions. One of them works when D is divisible by 4, but not 32, with a height factor between 6 and 72.
- `CM_INVARIANT_SIMPLEETA`: Simple η-quotients of the form $\eta(z)/\eta(z/N)$ for an integer N and their powers (to an exponent that divides 24), see [EnMo14], page 13. The code implements only prime power levels N for which $X_0(N)$ is of genus 0 (and the simple η-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 η-quotients of the form $(\eta(z/p_1)\eta(z/p_2))/(\eta(z)\eta(z/(p_1p_2)))$ for primes p_1 and p_2, and their powers, see [EnSc04], page 13, and [EnSc13], page 13. The level of the function is essentially $N = p_1p_2$ (or, to be more precise, a multiple of N and a divisor of $24N$, which depends on the congruences satisfied by the p_i modulo 24 and the exact power used). For $p_1, p_2 \to \infty$, these class functions approach a height factor of 12; the optimal height factor of 37 is reached for $p_1 = 2, p_2 = 73$.
- `CM_INVARIANT_MULTIETA`: Multiple η-quotients for k primes p_1, ..., 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 13. The code is implemented generically, but currently only triple η-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 *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 *invariant* is `CM_INVARIANT_DOUBLEETA` or `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 η-quotients are known to generate strict subfields of the class field in some cases, see [EnSc13], page 13. Whether this is admitted depends on the value of *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 *verbose* is set to `true`, some information is printed on screen during execution.

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

Since the function does not allocate any memory, there is no corresponding function `cm_param_clear`.

3.2 Computing class polynomials

`cm_class_t` [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 `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.

`void cm_pari_init ()` [Function]
This function must be called before any other function operating on class polynomials. Essentially it encapsulates a call to `pari_init`.

`void cm_pari_clear ()` [Function]
This function should be called at the end of working with the CM library. Essentially it encapsulates a call to `pari_close`.

`void cm_class_init (`*cm_class_t* `c`, *cm_param_t* `param`, *bool* `verbose``)` [Function]
This function should be called after `cm_pari_init`. Given a CM parameter *param* initialised with a call to `cm_param_init`, it allocates memory and executes some fast precomputations (such as the class group), the results of which are stored in *c*.

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

`void cm_class_clear (`*cm_class_t* `*c`)` [Function]
This is the counterpart to `cm_class_init` and should be called once the class polynomial is not needed any more to free the allocated space.

`void cm_class_compute (`*cm_class_t* `c`, *cm_param_t* `param`, *bool* `classpol`, *bool* `tower`, *bool* `verbose``)` [Function]
Given a previously initialised `cm_class_t` object *c* and corresponding parameter object *param*, the function computes the class polynomial and stores it in *c*.

More precisely, if *classpol* is set to `true`, the function computes the class polynomial in one variable X defining the class field; if *tower* is set to `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 13. 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 Z; or they are complex, that is, they have coefficients in the ring of integers of the imaginary-quadratic field $Q(\sqrt{D})$. We write the latter using the standard basis as $Z + Z\omega$, where D_0 is the fundamental discriminant attached to D and $\omega = \sqrt{D_0}/2$ if D_0 is even and $\omega = (1 + \sqrt{D_0})/2$ if D_0 is odd.

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

`void cm_class_print_pari` (*FILE** `file`, *cm_class_srcptr* `c`, *char** `fun`, *char** `var`) [Function]

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 ω is abbreviated to `o`.

If computed, the class polynomial tower is printed with fun_i 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 `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.

`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`) [Function]

Let *c* contain a computed class polynomial for the CM parameters *param*. Let $p \geq 5$ be a prime such that the elliptic curve with CM by the discriminant D in *param* has $l \cdot co$ points, where l is prime and co not divisible by l; otherwise said, there are integers t and v such that $p+1-t = l \cdot co$ and $4p = t^2 - v^2 D$. Then the function returns in *a* and *b* the parameters of an elliptic curve $E : Y^2 = X^3 + aX + b$ over the finite field F_p and in *x* and *y* a point $P = (x,y)$ of order *l*.

modpoldir is the name of the base directory containing subdirectories with modular polynomials for the different families of class functions; for a standard installation, this is `/usr/local/share/cm`.

If *print* is set to true, the computed curve and point coordinates are output on screen.

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

`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`) [Function]

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 `cm_curve_and_point`. Besides the discriminant it takes *fieldsize* and outputs in p the characteristic of a finite prime field with *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 *verbose* is set to `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 fastECPP of [FrKlMoWi04], page 13, and [Morain07], page 13.

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

`void cm_ecpp` (*mpz_t* `N`, *const char** `modpoldir`, *const char** `filename`, *const char** `tmpdir`, *bool* `print`, *bool* `trust`, *bool* `check`, *int* `phases`, *bool* `verbose`, *bool* `debug`) [Function]

Given a prime number N, the function computes an ECPP certificate and prints it to screen if *print* is set to `true`. If *filename* is different from `NULL`, the final ECPP certificate is output to the file in PARI/GP format, and to a file with additional suffix `.primo` in PRIMO format, and the phase 1 and phase 2 partial results are read from and written to temporary files with suffices `.cert1` and `.cert2`, respectively. The variable *phases* is usually set to 0; alternatively, it can be set to 1, in which case only the first, downrun phase of ECPP is executed; or it can be set to 2, in which case a (potentially partial) first phase result is read from a file and only the second, CM phase of ECPP is executed, resulting in a (potentially partial, not ending with a number below 2^{64}) certificate. Using values different from 0 makes sense only when *filename* is given at the same time.

If *check* is set to `true`, then the certificate is checked; the outcome is printed if *verbose* is also set to `true`. Notice that partial certificates are invalid by definition, even if their content is so far correct.

The directory *modpoldir*, usually `/usr/local/share/cm`, in which modular polynomials are stored, is required to be passed to `cm_curve_and_point`.

If *trust* is set to `true`, then the input number is trusted to be pseudoprime; otherwise a quick primality test is run.

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

If additionally to *verbose*, *debug* is set to `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.

The parameter *tmpdir*, if not `NULL`, indicates a directory where files can be written and read that are normally recomputed for every execution, but that contain data which does not actually depend on the number to be certified. These could theoretically be distributed with the code; since they can fill tens of gigabytes, this is not practical, however, and they are computed and written on the first run. For the MPI code, this directory needs to be accessible from all MPI processes. If the variable is `NULL`, the data is recomputed every time.

4 Applications

CM comes with a few applications: `classpol` for computing class polynomials and `cm` for computing a CM elliptic curve that could be used for cryptography, and `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 `classpol`

The `classpol` application takes one mandatory argument, `-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 `-v` enables more verbose output for the different steps of the algorithm.

```
$ classpol -d 207
f = x^6+4265376601839401837 5*x^5-5002547112103664005187500*x^4
+1819343755841562591564610147379736328125*x^3
-2106721098515824460652481971141159558105468 75*x^2
+1204102829191018181827435588509280939886474609375 0*x
-1834268645808184961797936493728671889305114746093 75
```

Class polynomials for alternative class invariants are selected using the `-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 `CM_INVARIANT_` left out and transformed to lower case; so `CM_INVARIANT_WEBER` becomes `-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
```

The verbose output tells us that -207 is not a fundamental discriminant, but a multiple of the fundamental discriminant -23, and that the double η-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:

```
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 ();
```

The call to `cm_param_init` initialises the variable *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 η-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 `CM_SUBFIELD_NEVER`. The 0 indicates that the degree of modular polynomials may be arbitrarily high, since they would only be needed to derive CM elliptic curves.

The calls to `cm_pari_init` and `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 `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 `cm_class_print_pari`.

4.2 `cm`

The `cm` application takes the same command line arguments `-d`, `-i` and `-v` as `classpol`, See Section 4.1 [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 `-v` parameter is needed to print the computed parameters on screen.

```
cm -d 207 -i doubleeta -v
Invariant d, parameter 2_13_2_2
...
p = 115792089237316195398462578067141184801329650642019283009460547375490535224057
n = 4 * 28948022309329048849615644516785296200162271477044351520651896284230459251901
a = 98163214185470497050837006097264380779085406912553098655936731136039029295843
b = 32285125078980297712817393022671977836076850091512599038419136678099671035526
x = 10339764456719713530963337817174512858905605667367531988964565699343338764432O
y = 72297811879224681619558031933509678790723605942283336960075126018884088267112
```

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:

```
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 `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 -1 to limit the search for class invariants for which modular polynomials are available. (The verbose output shows that the η-quotient of level $2{\cdot}73$ is replaced by the quotient of level $2{\cdot}13$ 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 `cm_class_compute` lead to the computation of the tower decomposition of the class field instead of the full class polynomial. The call to `cm_curve_crypto_param` determines a suitable 256 bit prime and curve cardinality n, and the call to `cm_curve_and_point` obtains and prints the curve and the coordinates of a point of prime order l on the curve.

4.3 eccp

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.

```
$ ecpp -n 'nextprime(10^31)' -p
c = [[10000000000000000000000000000033, -5882759018432034, 12103604, 25,
[387686851611416530808239351962 3, 726859802033890644763485750 3614]],
[826200196239071096737717, 667597927066, 916, 0,
[376111257001838975439341, 115218092585553575608847]],
[901965279736248361147, 54401389280, 118118316, 0,
[282628664937444390372, 3523994183337197608 52]]];
```

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

```
? 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 in PARI/GP format, and additionally in the file *filename*`.primo` in Primo (see [Primo], page 13) format (for checking it with Primo, one needs to rename the file to *filename*`.out`). 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,

```
$ 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.

By default, the code carries out a quick (as opposed to thorough) primality test, which is meant to catch typos and obvious errors. If the number is trusted to be pseudoprime since this has been tested independently, the command line switch `-t` can be added to drop this test.

The environment variable `CM_ECPP_TMPDIR`, if set in the shell from which the binary is invoked, indicates a directory, available from all MPI processes for the parallel version, in which data that is common over several invocations can be stored. The data is then computed on the first run and read from disk during subsequent runs. If the environment variable is not set,

the data is recomputed every time. This concerns files named `cm_h.dat` for class numbers and `cm_prim_xxxx.dat` for primorials.

Setting this variable also causes checkpoint files named `cm_factor_xxxxxxxxxxxxxxx.dat` to be written to this directory; they are useful when certifying very large numbers, for which the program execution may be interrupted during the second ECPP phase. They depend on the number to be certified and can be deleted after the certificate creation.

It is possible to run only the first, downrun phase or only the second, CM phase of ECPP using the command line options `-1` or `-2`, respectively. Both require that a filename be given with the `-f` option. Running only the first phase makes it possible to allocate a different (usually larger) number of MPI processes to this phase. Running only the second phase is possible also when the first phase has been completed only partially, so that the first elliptic curves, which often take a very long time, can be obtained on fewer cores in parallel with the ongoing first phase. Notice that in this case a certificate is written, but that it is considered invalid in particular by the option `-c`. It can be completed by subsequent runs specifying `-1`, `-2` or none of them.

4.4 `ecpp-check`

This is a small helper application to check the validity of ECPP certificates. It requires the `-f` parameter followed by the name of a file containing a certificate in the PARI/GP format, as created by the `ecpp` or `ecpp-mpi` binaries started with the `-f` parameter, or by PARI/GP with the `primecert` command. The ECPP certificates of CM and PARI/GP may differ slightly: CM creates points of prime orders on the elliptic curves, while the points created by PARI/GP have prime order only after multiplication by the smooth cofactor. So every valid CM certificate is a valid PARI/GP certificate, but not vice versa. The application checks both cases. It can also be used to verify partial certificates, that is, prefixes of certificates, in which the downrun has been stopped before the size of the number has dropped below 64 bits. This can be used to check whether partial computations are correct.

Verification uses functions from PARI/GP, to provide for a way to check independently the validity of certificates created by CM. The same internal machinery is used by the `ecpp` and `ecpp-mpi` binaries invoked with the `-c` flag. PARI/GP multithreading is automatically taken into account.

References

- [Enge09] Andreas Enge. The complexity of class polynomial computation via floating point approximations. *Mathematics of Computation* 78 (266), 2009, pp. 1089–1107
- [Enge21] Andreas Enge. `mpfrcx` – A library for the arithmetic of univariate polynomials over arbitrary precision real or complex numbers. INRIA. Version 0.6.3, 2021, `http://mpfrcx.multiprecision.org/`
- [EnGaThZi20] Andreas Enge, Mickaël Gastineau, Philippe Théveny and Paul Zimmermann. `mpc` – A library for multiprecision complex arithmetic with exact rounding. INRIA. Version 1.2.1, 2020, `http://mpc.multiprecision.org/`
- [EnMo03] Andreas Enge and François Morain. Fast decomposition of polynomials with known Galois group. In Marc Fossorier, Tom Høholdt and Alain Poli (eds.): *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes – AAECC-15*, 2003, pp. 254–264
- [EnMo14] Andreas Enge and François Morain. Generalised Weber functions. *Acta Arithmetica* 164 (4), 2014, pp. 309–341
- [EnSc04] Andreas Enge and Reinhard Schertz. Constructing elliptic curves over finite fields using double eta-quotients. *Journal de Théorie des Nombres de Bordeaux* 16, 2004, pp. 555–568
- [EnSc13] Andreas Enge and Reinhard Schertz. Singular values of multiple eta-quotients for ramified primes. *LMS Journal of Computation and Mathematics* 16, 2013, pp. 407–418
- [FrKlMoWi04] J. Franke and T. Kleinjung and F. Morain and T. Wirth. Proving the Primality of Very Large Numbers with fastECPP. In Duncan Buell (ed.): *Algorithmic Number Theory – ANTS-VI*, 2004, pp. 194–207
- [Gretal20] Torbjörn Granlund et al. `gmp` – GNU multiprecision library. Version 6.2.1, 2020, `http://gmplib.org/`
- [HaLePeZi20] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann et al. `mpfr` – A library for multiple-precision floating-point computations with exact rounding. Version 4.1.0, 2020, `http://www.mpfr.org/`
- [Morain07] FranImplementing the asymptotically fast version of the elliptic curve primality proving algorithm. *Mathematics of Computation* 76 (257), 2007, pp. 493–505
- [Pari] The PARI group. `PARI/GP`. Version 2.13.4, 2022, `https://pari.math.u-bordeaux.fr/`
- [Flint] William Hart et al. `FLINT`. Version 2.9.0, 2022, `https://flintlib.org/`
- [Primo] Marcel Martin. `Primo`. Version 4.3.3, 2020, `http://www.ellipsa.eu/public/primo/primo.html`

Index