

# ALGORITHMS FOR COMPUTER PROGRAMS

BERNDT E. SCHWERTDFEGER

*pour Cathérine*

ABSTRACT. This paper collects the algorithms that I developed for calculating the  $\zeta$ -functions of some particular curves. In particular, the TATE algorithm for handling WEIERSTRASS equations of elliptic curves is treated.

## PREFACE

This is an old collection of routines for doing elementary arithmetic on a computer, including calculations in finite fields. Some have emerged from some Algol programs of mine written in 1978–1980 at the Faculty of Mathematics in Bielefeld. They are now implemented in C (interim versions in REXX have been removed).

These notes are meant to supplement the program logic, with emphasis put on the conceptual ideas behind those algorithms. The only change in version 1.7 is to the factorisation routine. In version 1.8 the program listings make use of the package `listings`.

Berlin, October 29, 2018

© 1998–2018 Berndt E. Schwerdtfeger

v1.8, October 30, 2018

## CONTENTS

Preface	1
History	2
1. Greatest Common Divisor	2
1.1. GCD in REXX	3
1.2. Extended GCD in C	4
1.3. Factorization into primes	6
2. ERATOSTHENES	10
3. The JACOBI Programs	13
3.1. Jac1: calculations over $\mathbf{F}_p$	14
3.2. Jac2: calculations over $\mathbf{F}_{p^2}$	15
3.3. Jac2a: calculations over $\mathbf{F}_{p^2}$ for $p \equiv 3 \pmod{4}$	16
3.4. Jac3: calculations over $\mathbf{F}_{p^3}$	17
4. The ‘Elliptic’ Program	18
4.1. ELLALG	18

---

2010 *Mathematics Subject Classification*. Primary 11G05; Secondary 14H52.  
*Key words and phrases*. elliptic curves, semistable.

4.2. ELLIP	19
4.3. Elliptic Code in C	25
4.4. Calculation of a resultant	32
References	34

### History.

- v0.1:** April 5, 1998. Initial draft document starting with the gcd-routine (*Euclidean* algorithm).
- v1.0:** December 30, 2002. First published version containing
- the gcd and extended gcd algorithms,
  - the factorisation into primes and prime table programs (based on sieve of ERATOSTHENES),
  - two initial JACOBI programs (in Algol) that I developed in 1978 in support of my conjecture ([5]) about the structure of the JACOBI variety  $J_0(64)$  of the modular curve  $X_0(64) : x^4 + y^4 + z^4 = 0$  (for the proof see [6]).
  - the elliptic program code in C
- v1.1:** January 3, 2003. enhanced prime factorisation program, updated the elliptic code accordingly, enhanced the fiber type routine
- v1.2:** January 5, 2003. Added the remark that the sources of my programs are published under the GNU General Public License. Added also the remaining JACOBI programs.
- v1.3:** March 11, 2003. modified the elliptic program (display routine).
- v1.4:** April 5, 2003. Everything was put under CVS control. Updated the C-routines to latest level.
- Comme j'ai commencé cet article il y a cinq ans au jour exact, je l'ai dédié à *Catherine* pour son anniversaire.
- v1.5:** June 1, 2007. Added rcs package. Reference to BÉZOUT.
- v1.6:** December 3, 2010. Sources are now licensed under the Apache License, Version 2.0 (2005) <http://www.apache.org/licenses/LICENSE-2.0>.
- v1.7:** December 15, 2012. modified the factorization routine to use long long integers.
- v1.8:** October 29, 2018. making use of the listings package (removing verbatim from the sources)

## 1. GREATEST COMMON DIVISOR

The greatest common divisor  $d$  of the (positive, rational) integers  $a, b$  is defined by BÉZOUT's identity

$$a \cdot \mathbf{Z} + b \cdot \mathbf{Z} = d \cdot \mathbf{Z}$$

Of course,  $d$  is only unique up to the units in  $\mathbf{Z}$ , i.e. up to sign, but it is always understood that we take  $d > 0$ .

The *Euclidean* algorithm gives an effective way to modify the generators  $(a, b)$  by operations  $g \in GL(2, \mathbf{Z})$ . It rests on the integral division with remainder

$$a = q \cdot b + r \quad \text{with } 0 \leq r < b$$

Here  $q$  and  $r$ , subject to the above condition, are uniquely determined. In particular,

$$a \cdot \mathbf{Z} + b \cdot \mathbf{Z} = b \cdot \mathbf{Z} + r \cdot \mathbf{Z}$$

Let  $g \in GL(2, \mathbf{Z})$  be the matrix  $g = \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$ . The *Euclidean* algorithm now reads  $(a, b) = (b, r) \cdot g$  with  $\det g = -1$ .

We have finished our search, when  $b$  divides  $a$  ( $r = 0$ ) and then the gcd is  $d = b$ . Otherwise ( $r > 0$ ), we may start over again, now with the pair  $(b, r)$ . This process will come to an end, as we eventually must reach the case  $r = 0$ .

Let  $n$  be the number of steps and let now  $g \in GL(2, \mathbf{Z})$  be the product of the above elementary matrix operations. We have

$$(a, b) = (d, 0) \cdot g$$

$$\det g = (-1)^n$$

This will be re-written for  $g = \begin{pmatrix} u & v \\ w & x \end{pmatrix}$  as follows

$$a = u \cdot d$$

$$b = v \cdot d$$

$$a \cdot x - b \cdot w = (-1)^n d$$

Remark, by construction, that all matrix elements of  $g$  are  $\geq 0$ .

The iterative step in the program calculates the matrix product

$$\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} u & v \\ w & x \end{pmatrix} = \begin{pmatrix} qu+w & qv+x \\ u & v \end{pmatrix}$$

1.1. **GCD in REXX.** The following REXX program implements this algorithm, calculating the matrix  $g$ .

```

/* ----- */
//
// gcd          greatest common divisor
//
//   Input:  a, b  integers
//   Output: gcd(a,b)
//
//
// $Date: 2010-12-02 23:46:56 $
// $Revision: 1.4 $
//
// -----
//
// Copyright (C) 1998-2010 Berndt E. Schwerdtfeger
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//   http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//
// ----- */

/* get the arguments */

```

**Arg** a b .

```

If a = '?' | a = '' then
do
  say ' '
  say 'Syntax is: gcd_a_b'
  say ' '
  say 'Output is d=gcd(a,b), the factors of a and b'
  say 'and the linear relation d=ax+by'
exit
end

```

main:

```

d = a; r = b
u = 1; v = 0; w = 0; x = 1; det = 1

do until r = 0
  q = d % r
  d0= r          /* save old r */
  r = d - q*r    /* new r */
  d = d0         /* new d */
  u0= u; v0= v
  u = q*u + w; v = q*v + x /* matrix */
  w = u0 ; x = v0        /* ... product */
  det = - det
end

say ' '
say a '=' d '*' u
say b '=' d '*' v
say a '*' x '-' b '*' w '=' d*det

return

```

1.2. **Extended GCD in C.** The following C program implements the same algorithm, but using the inverse matrices  $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$ . Instead of the equation  $(a, b) = (d, 0) \begin{pmatrix} u & v \\ w & x \end{pmatrix}$  one considers now  $(u, v) \cdot \begin{pmatrix} u_1 & v_1 & t_1 \\ u_2 & v_2 & t_2 \end{pmatrix} = (u_0, v_0, t_0)$  in the notation of the C program below, with the ultimate goal that  $v_0 = 0$  vanishes and that  $u_0 = d$  is the  $gcd(u, v)$ . See KNUTH, *The Art of Computer Programming* [1, vol. 2, §4.5.2], Algorithm X.

/\* -----

Module: xgcd.c

Description:

*Extended Euclidean Algorithm  
finding the greatest common divisor*

*Input: a, b integers*

*Output:  $x*a + y*b = d$*

*\$Date: 2012-12-02 16:28:11 +0100 (Sun, 02 Dec 2012) \$*

*\$Revision: 316 \$*

*Version: 1.7*

*Copyright (C) 2002-2013 Berndt E. Schwerdtfeger*

*Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.  
You may obtain a copy of the License at*

*<http://www.apache.org/licenses/LICENSE-2.0>*

*Unless required by applicable law or agreed to in writing, software  
distributed under the License is distributed on an "AS IS" BASIS,  
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
See the License for the specific language governing permissions and  
limitations under the License.*

```

*/
#include <stdio.h>
#include <stdlib.h>

int main(void);

int main(void)
{
    long u, v, u0, v0, u1, v1, u2, v2, t0, t1, t2;
    long q;

    /* get the input */

    scanf("%d_%d", &u, &v);

    u0 = u; v0 = v;          /* the following matrix is the inverse .. */
    u1 = 1; v1 = 0;         /* .. of my rexx procedure gcd          */
    u2 = 0; v2 = 1;         /* (u,v) * mat(u1,v1 // u2,v2) = (u0,v0) */
                                /* u1*u + u2*v = u0 = gcd(u,v)          */
    while (v0) {            /* u = u0*|v2| , v = u0*|v1|          */
        q = u0 / v0;

```

```

    t0 = u0 - q*v0;
    t1 = u1 - q*v1;
    t2 = u2 - q*v2;

    u0 = v0;
    u1 = v1;
    u2 = v2;

    v0 = t0;
    v1 = t1;
    v2 = t2;

}

printf("gcd(%d,%d) = %d\n", u, v, u0);
printf("The coefficients in  $x*u + y*v = d$  are:  $x = %d, y = %d$ \n", u1, u2);
printf("... the divisor are  $u/d = %d, v/d = %d$ \n", abs(v2), abs(v1));

return 0;
}

```

1.3. **Factorization into primes.** The following C program implements the algorithm A of KNUTH [1, vol. 2, §4.5.4]. I have made use of the sieve of ERATOSTHENES (see §2).

I make use of long long integers, i.e. the input parameter can be as large as  $2^{64} - 1 = 18,446,744,073,709,551,615 > 18 \cdot 10^{18}$ . For my practical purposes this is sufficient.

```

/* -----
Module:      fact.c

Description:

    Factorisation into primes

    Input:  N integer to be factored (of type unsigned long long)
           maximal number handled is
            $2^{64} - 1 = 18,446,744,073,709,551,615 > 18E18$ 

    Output: p[] array of primes, e[] array of exponents

$Date: 2012-12-02 16:28:11 +0100 (Sun, 02 Dec 2012) $
$Revision: 316 $
Version: 1.7 changed prime array to long long
-----

```

Copyright (C) 2002–2013 Berndt E. Schwerdtfeger

Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

```

----- */

#include <stdio.h>
#include <inttypes.h>

int main(void);
int main(void)

{
// some "magic constants"

#define L      14000
#define M      2*L*(L+1)           // = 1152048000 (1152MB)
// #define M    1316300
#define YES    1                  // flag for primes
#define NO     0                  // flag for composite numbers

    unsigned long s, t;
// unsigned long long int N;           // integer to be factorized
// unsigned long long p[30];          // array of prime factors @ 1.7
// int e[30];                          // array of exponents
    unsigned long d[220000000];        // array of test divisors (880MB)

    int sieve(unsigned long []);
// int fact( unsigned long long, unsigned long [], unsigned long long [], int []);
//                                     // @ 1.7
    t = sieve(d);                      // t = 192080, p = 2632601 > 2^21 = 2097152

// for (s=0;s<t;s++)
//     printf("%lu%c", d[s], s%8 ? '\t' : '\n');

    printf("\nt = %lu, p = %lu", t, d[t-1]);
    printf("\nL = %lu, 2*L*(L+1) = %lu, M = %lu", L, 2*L*(L+1), M);
    return 0;                          // EARLY EXIT !!!
// *****
// scanf("%llu", &N);

// printf("\nN = ");

// if (N) {
//     t = fact(N, d, p, e);             // factorize the number

```

```

//   printf("%llu = ",N);
//   if (t) {
//       for (s=1; s<t ; s++)
//           printf("%llu(%u) * ",p[s], e[s]); // @ 1.7
//       printf("%llu(%u)",p[t], e[t]);      // @ 1.7
//   }
//   else
//       printf("1");                       // if no factors , it is 1
// } else
//   printf("0");

// return 0;
}

/* -----
**
** Subroutine sieve
**
** Description: int sieve(unsigned long d[]);
**
**      Generating array of test primes via sieve of Eratosthenes
**
**      Input: none
**
**      Output: return value, vector d[] of test divisors
**
** ----- */

int sieve(unsigned long d[])
{
// sieve of Eratosthenes

    unsigned int a, i;                // loop ind, exponent

    /* the array prime[x] test for primality of 2*x + 1 */

    char prime[M+1];                 // byte array for flags

    for (a = 0; a <= M; a++)          // initialize ...
        prime[a] = YES;              // ... to everything prime
    for (a = 1; a <= L; a++)          // go thru all primes and
        if (prime[a])                // throw out the multiples
            for (i = 2*a*(a+1); i <= M; i += 2*a+1)
                prime[i] = NO;        // multiples are not prime

// populate the vector of test divisors

    i = 0;
    d[i++] = 2;                       // set the even prime

```



```

for (a = 1; a<=M; a++)
    if (prime[a])
        d[i++] = 2*a + 1;           // set the uneven primes

return i;
}

/* -----
**
** Subroutine fact
**
** Description:
**     The number N is factored into primes
**      $N = p[1]**e[1] * \dots * p[t]**e[t]$ 
**
** Input: N (integer), test divisors in vector d[]
**
** Output: vector p[], vector e[]
**          the vector p[] contains the different primes
**          the vector e[] contains the orders of N at these primes
**          p[0] = t = return value = number of primes (also t = 0)
**          (if N = 0 then e[0] = 1, otherwise e[0] = 0)
**
** ----- */
int fact(unsigned long long N, unsigned long d[], unsigned long long p[], int e[])
{
    // @ 1.7

    // this is Knuth's algorithm, modified with exponents

    unsigned int i, t, k;
    unsigned long long n, q, r;

    n = N;           // A1 initialize
    i = t = k = 0;

    while (n > 1) { // A2 n = 1 ?
        q = n / d[k]; // A3 divide
        r = n - q * d[k];
        if (r == 0) { // A4 zero remainder ?
            if (i == 0)
                p[++t] = d[k]; // A5 new factor found
            e[t] = ++i; // increase exponent
            n = q;
        } // return to A2
        else {
            if (q > d[k]) { // A6 low quotient ?
                k = k + 1; // no, .. try next
                i = 0;
            } // return to A3
            else { // yes, ..
                p[++t] = n; // A7 n is prime
            }
        }
    }
}

```

```

        e[t] = 1;           // exponent is 1
        n = 1;           // terminate
    }
}
}
p[0] = t;                // number of factors
if (n)
    e[0] = 0;            // e[0] discriminates
else
    e[0] = 1;           // .. if n vanishes

return t;
} */

```

## 2. ERATOSTHENES

---

Module:     *prim.c*

Description:

*Eratosthenes (274–194 b.C.) used the following reasoning for obtaining all primes:*

*go thru all natural numbers  $q > 1$   
'flag' all multiples ( $2q, 3q, 4q \dots$ )  
stop at any time (number  $N$ , say).  
all numbers not flagged up to  $N$  are prime  
(since not a multiple of any other number)*

*This simple reasoning can be modified in the following way:*

*First only odd numbers need to be considered ( $q=2a+1$ ). So the flags for the  $a$ 's make up an array of half the size  $\text{prim}[M+1]$ , where the odd number  $N = 2*M + 1$ .*

*Second, each time we only need to flag multiples of  $q$ :  $q*q, q*(q+2), q*(q+4), \dots$  up to  $N$  ( $q$  is odd), since smaller multiples have already been accounted for.*

*This also means that  $q$  only loops from 3 up to  $\text{sqrt}(N)$ .*

*In the formula below we assume  $\text{sqrt}(N)$  is integral, odd, i.e.*

*$N = 2*M + 1 = (2*L+1)*(2*L+1)$ , therefore  $M = 2*L*(L+1)$ .*

*This program uses the maximum  $L = 127$  for the current implementation (memory model). Either other memory models or usage of bitflags instead of byte arrays can generate larger tables. This is done elsewhere (see *primx.c*).*

*\$Date: 2013-01-06 11:12:37 \$*

*\$Revision: 1.5 \$*

History:

06.08.1989 1.00 initial version

05.12.1989 1.01 use the "? : " construct for deciding newline  
 31.01.1993 1.01.01 provide a prolog and description  
 19.02.1994 1.02 modified algorithm  
 20.02.1994 1.02.01 fully document the algorithm in the description  
 30.03.1997 1.02.02 copyright statement  
 05.04.2003 1.1 put into CVS repository  
 06.01.2013 1.5 modified description, no program change

Notes: adopted from an original Algol60 program written in 1979

Copyright (C) 1989–2013 Berndt E. Schwerdtfeger

Licensed under the Apache License, Version 2.0 (the "License");  
 you may not use this file except in compliance with the License.  
 You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software  
 distributed under the License is distributed on an "AS IS" BASIS,  
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 See the License for the specific language governing permissions and  
 limitations under the License.

```

#define L      127           // has to be constant
#define M      2*L*(L+1)     // = 35512, N = 65025 = 255*255
#define COLUMNS 8         // for the screen output
#define ON      0           // flag for primes
#define OFF     1           // flag for composite numbers

#include <stdio.h>

int main(void)
{
    unsigned int a,b;         // loop ind
    char prim[M+1];          // byte array for flags

    for (a = 0; a <= M; ++a)  // initialize ...
        prim[a] = ON;       // ... to everything prime
    for (a = 1; a <= L; ++a)  // go thru all primes and
        if ( prim[a] == ON ) // throw out the multiples
            for (b = 2*a*(a+1); b <= M; b += 2*a+1)
                prim[b] = OFF; // multiples are not prime

    b = 1;
    printf("%u%c",2,'\t');   // 2 is the first prime
    for (a = 1; a <= M; ++a) // put out the result
        if ( prim[a] == ON ) // if it is a prime

```

```

        printf("%0c", 2*a + 1 , ++b % COLUMNS ?'\t':'\n');
    return (0);
}

```

---

*Module:*        *primx.c*

*Description:*

*The general reasoning is explained in prim.c, which uses a byte array for the flags, whereas here I use a bit-array (put into an array of long's).*

*I use 32-bit flat address space. On my particular machine with 2 GByte of storage the largest prime generated was*

*$p = 4, x??, ???, ???$  (*

*\$Date: 2010-12-03 00:15:38 \$*

*\$Revision: 1.4 \$*

*History:*

21.05.1979            *original version in Algol60 written in Bielefeld*  
 20.02.1994    1.00    *rewrite in C*  
 21.02.1994    1.00.01 *loop restriction added (overflow fixed)*  
 30.03.1997    1.00.02 *copyright notice*  
 28.07.2001    1.00.03 *adaption to GNU/Debian platform*  
 30.03.2003    1.01.04 *M = 520128, max prime = 33,288,251*  
 21.01.2013    1.5        *M = 2<sup>26</sup>-1 = 67108863, max prime =*

---

*Copyright (C) 1979-2013 Berndt E. Schwerdtfeger*

*Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at*

*<http://www.apache.org/licenses/LICENSE-2.0>*

*Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.*

---

```

#define M      67108863           // 226-1                @1.5
#define N      33554432           // 225
#define COLUMNS 256             // for file output

#include <stdio.h>

```

```

int main(void)
{
    unsigned long p, q, r, y, z;           // loop ind
    unsigned long L[M+1];                 // long array for bitflags

    for (p = 0; q <= M; ++p)               // initialize ...
        L[p] = 0;                          // ... to everything prime

    for (q = 0; 32*q*q <= M; ++q)          // go thru all flag fields .. @1.00.01
        for (r = 0; r <= 31; ++r)        // .. for all bits in a long
            if (!(1 & L[q]>>r))            // p = 64*q+2*r+3 is prime
            {
                y = 2*q*(32*q+2*r+3) + (2*r*(r+3)+3)/32;
                z = (2*r*(r+3)+3)%32;
                while ( 32*y + z <= (long) 32*M + 31 )
                {
                    L[y] |= (long)1 << z; // flag as composite
                    y += 2*q+(z+2*r+3)/32;
                    z = (z+2*r+3)%32;
                }
            }

    z = 1;
    printf("%a%c_", 2, ', ', ');           // 2 is the first prime

    FILE *fp = fopen("first.txt", "w");

    for (q = 0; q <= N; ++q)               // go thru all flag fields ..
        for (r = 0; r <= 31; ++r)        // .. for all bits in a long
            if (!(1 & L[q]>>r))            // the r-bit in L[q] is off (prime!)
                fprintf(fp, "%lu%c_", 64*q+2*r+3, ++z % COLUMNS ? ', ':'\n');
    fclose(fp);

    FILE *zp = fopen("zweit.txt", "w");
    for (q = N; q <= M; ++q)               // go thru all flag fields ..
        for (r = 0; r <= 31; ++r)        // .. for all bits in a long
            if (!(1 & L[q]>>r))            // the r-bit in L[q] is off (prime!)
                fprintf(zp, "%lu%c_", 64*q+2*r+3, ++z % COLUMNS ? ', ':'\n');
    fprintf(zp, "\nt_=%lu", z);
    fclose(zp);
    return (0);
}

```

### 3. THE JACOBI PROGRAMS

The two curves, that I was concerned with, are

$$\mathbf{E}: y^2 = x^3 + x$$

$$\mathbf{C}: x^4 + y^4 + z^4 = 0$$

Consult my article ‘The Zeta-Function of  $x^4 + y^4 + z^4 = 0$ ’ [6] for the theory and more background.

The JACOBI programs calculate the number of points of them over the prime field  $\mathbf{F}_p$  and the quadratic and cubic extensions  $\mathbf{F}_{p^2}$ ,  $\mathbf{F}_{p^3}$ .

There are these ALGO1 programs to do the job:

**JAC1:** calculations over  $\mathbf{F}_p$   
**JAC2:** calculations over  $\mathbf{F}_{p^2}$   
**JAC2A:** calculations over  $\mathbf{F}_{p^2}$  for primes  $p \equiv 3 \pmod{4}$ .  
**JAC3:** calculations over  $\mathbf{F}_{p^3}$

Although it is not mentioned in the sources, they are provided on the basis of the Apache License.

### 3.1. Jac1: calculations over $\mathbf{F}_p$ .

```
begin comment JAC1 ;
  integer n,q,p,e1,c1,u1,s1;
  integer procedure mod(a);
  value a;
  integer a;
  begin
    a := a - entier(a/p)*p ;
    if a > (p-1)/2 then
      a := a-p ;
      mod := a
  end; comment
Es folgen die Anweisungen;
output(4, '*4B, 'P E1 C1', 9B, 'U1 S1' );
input(1, 'B3D', n); for a := 1 step 1 until n do
begin
  input(1, 'B4D', p);
  begin
    integer array q[-(p-1)/2:(p-1)/2];
    integer x,y;
    for x := -(p-1)/2 step 1 until (p-1)/2 do q[x] := 0;
    for x := 0 step 1 until (p-1)/2 do
      begin
        y := mod(x*x);
        q[y] := 1
      end;
    for x := -(p-1)/2 step 1 until (p-1)/2 do
      begin
        y := mod(x*mod(x*x+1));
        if q[y] = 1 then e1 := e1 + 2
      end;
    for x := -(p-1)/2 step 1 until (p-1)/2 do q[x] := 0;
    for x := 0 step 1 until (p-1)/2 do
      begin
        y := mod(x*x);
        y := mod(y*y);
        q[y] := 1
      end;
    for x := -(p-1)/2 step 1 until (p-1)/2 do
      begin
```

```

        y := mod(x*x);
        y := -mod(mod(y*y)+1);
        if q[y] = 1 then c1 := c1 + 4
    end;
    if entier((p-1)/8)*8 + 1 = p then c1 := c1 - 8;
    ul := 1 + p - e1;
    s1 := 1 + p - c1;
    output(4, '/4ZD,4ZD,4ZD,-9ZD,-3ZD', p, e1, c1, ul, s1);
    if s1 != 3*ul then
        output(4, '5B, 'Gegenbeispiel' ');
    if 6*sqrt(p) - abs(s1) < 1 then
        output(4, '4B,ZZD DD', 6*sqrt(p))
    end
end
end
end Programmende von JAC1

```

### 3.2. Jac2: calculations over $F_{p^2}$ .

```

begin comment JAC2 ;
    integer p,d,e,c,n,l;
    integer procedure mod(a);
    value a;
    integer a;
    begin
        a := a - entier(a/p)*p ;
        if a > (p-1)/2 then
            a := a-p ;
            mod := a
        end; comment
Anweisungen;
    output(4, '*4B, 'P'8B'E2')8B'C2'8B'U2'8B'S2' ');
    input(1, 'BDD', n); for l := 1 step 1 until n do
    begin
        integer q;
        input(1, 'B3D', p);
        q := (p-1)/2;
        begin
            integer array a[-q:q,-q:q];
            integer x,y,t,u,v;
            for x := -q step 1 until q do a[x,0] := 0;
            for x := 0 step 1 until q do
                begin
                    y := mod(x*x);
                    a[y,0] := 1
                end;
            e := -2;
            for x := -q step 1 until q do
                begin
                    y := mod(x*mod(x*x+1));
                    if a[y,0] = 1 then e := e + 2
                end;
            e := e*((1+p)*2 - e);
            output(4, '/4ZD,9ZD', p, e);

```

```

e := 1 + p*p - e;
for x := -q step 1 until q do
  if a[x,0] = 0 then 'goto' nr;
nr: d := x;
for x := -q step 1 until q do
  for y := -q step 1 until q do a[x,y] := 0;
for x := -q step 1 until q do
  for y := -q step 1 until q do
    begin
      t := mod(x*x + y*y*d);
      v := mod(2*x*y);
      u := mod(t*t + v*v*d);
      v := mod(2*t*v);
      a[u,v] := 1
    end;
c := -8
for x := -q step 1 until q do
  for y := -q step 1 until q do
    begin
      t := mod(x*x + y*y*d);
      v := mod(2*x*y);
      u := - mod(t*t + v*v*d + 1);
      v := - mod(2*t*v);
      if a[u,v] = 1 then c := c + 4
    end;
output(4, '9ZD,-8ZD', c, e);
c := 1 + p*p - c;
output(4, '-8ZD', c);
if c != 3*e then
  output(4, 'Gegenbeispiel');
if 6*p - abs(c) < 1 then
  output(4, 'ZZD DD', 6*p);
end
end
end Programmende von JAC2

```

### 3.3. Jac2a: calculations over $F_{p^2}$ for $p \equiv 3 \pmod{4}$ .

```

begin comment JAC2A fuer Primzahlen kongruent 3 modulo 4 ;
  integer p, e, c, n, q ;
  integer procedure mod(a); value a ; integer a ;
  begin
    a := a - entier(a/p)*p ;
    if a > (p-1)/2 then
      a := a-p ;
      mod := a
    end; comment
Anweisungen; n := 0;
output(4, '*4B, 'P'8B'E2')8B'C2'8B'U2'8B'S2' ');
marke:
input(1, '/5BD', p); q := (p-1)/2;
begin integer 'array' a[-q:q, -q:q] ;
  integer x, y, t, u, v;

```



```

e := (1+p)*(1+p) ; output(4, '/4ZD,9ZD', p, e);
e := -2*p;
for x := -q step 1 until q do
  for y := -q step 1 until q do a[x,y] := 0;
for x := -q step 1 until q do
  for y := -q step 1 until q do
    begin
      t := mod(x*x - y*y); v := mod(2*x*y);
      u := mod(t*t - v*v); v := mod(2*t*v);
      a[u,v] := 1
    end;
c := -8
for x := -q step 1 until q do
  for y := -q step 1 until q do
    begin
      t := mod(x*x - y*y); v := mod(2*x*y);
      u := - mod(t*t - v*v + 1); v := - mod(2*t*v);
      if a[u,v] = 1 then c := c + 4
    end;
output(4, '9ZD,-8ZD', c, e);
c := 1 + p*p - c;
output(4, '-8ZD', c);
if c != 3*e then
  output(4, 'Gegenbeispiel');
if 6*p - abs(c) < 1 then
  output(4, 'ZZD DD', 6*p);
n := n + 1;
if n < 25 then goto marke
end
end Programmende von JAC2A

```

### 3.4. Jac3: calculations over $F_{p^3}$ .

```

begin comment JAC 3 ;
  integer p, q, e, c, d, s, t, n;
  integer procedure mod(a); value a ; integer a ;
  begin a := a - entier(a/p)*p ; if a > (p-1)/2 then
    a := a-p ; mod := a
  end ;
  procedure vier(x,y,z,u,v,w) ; integer x,y,z,u,v,w ;
  begin
    s := mod(x*x + 2* y*z*d) ;
    t := mod(2*x*y + 2*y*z + z*z*d) ;
    w := mod(2*x*z + y*y + z*z) ;
    u := mod(s*s + 2*t*w*d) ;
    v := mod(2*s*t + 2*t*w + w*w*d) ;
    w := mod(2*s*w + t*t + w*w) ;
  end ;
  output(4, '*4B, 'P'8B'E3'8B'C3'8B'U3'8B'S3'') ;
  n := 0;
marke:
  input(1, '/5BD,-DDB,DDD', p, e, d ) ; q := (p-1)/2;
  e := 1 + p*p*p + 3*p*e - e*e*e ; output(4, '/4ZD,9ZD', p, e);

```

```

e := 1 + p*p*p - e ;
begin integer array l[-q:q , -q:q , -q:q ] ;
  integer u, v, w, x, y, z ;
  for x := -q step 1 until q do
    for y := -q step 1 until q do
      for z := -q step 1 until q do
        l[x,y,z] := 0;
      for x := -q step 1 until q do
        for y := -q step 1 until q do
          for z := -q step 1 until q do
            begin vier(x,y,z,u,v,w) ; l[u,v,w] := 1 end ;
          if entier(q/4)*4 = q then c := -8 else c := 0 ;
          for x := -q step 1 until q do
            for y := -q step 1 until q do
              for z := -q step 1 until q do
                begin
                  vier(x,y,z,u,v,w) ; u := -mod(u+1) ; v := -v ; w := -w ;
                  if l[u,v,w]=1 then c := c + 4
                end;
              output(4, '9ZD,-8ZD',c,e) ; c := 1 + p*p*p - c ;
              output(4, '-8ZD',c) ;
              if c != 3*e then
                output(4, 'Gegenbeispiel') ;
              n := n + 1 ;
              if n < 4 then goto marke
            end
          end Programmende von JAC 3

```

#### 4. THE 'ELLIPTIC' PROGRAM

There are two Algol programs

ELLALG: version 3.3.1980 (without reduction)

ELLIP: version 11.3.1980 (with incomplete reduction)

each of which contains some more routines. There are also versions from the 6<sup>th</sup> and 10<sup>th</sup> of March as well, which describe the development from ELLALG towards ELLIP, which I did not finish then. I was obviously trying to add the necessary modifications to calculate the minimal WEIERSTRASS equation.

Here is first the high level structure of ELLALG:

##### 4.1. ELLALG. The main program does the following (only pseudo-code)

```

read number of Weierstrass equations e
for k = 1 to e
do
  read the coefficients
  calculate the Tate parms (b2,b4,b6,b8,c4,c6,d)
  write out the equation (drop zero's)
  if singular (d=0) then
    if c4 = 0 then write 'cusp' (additive)
    else write 'node' (multiplicative)
  endif

```

```

else (regular, that is rational elliptic curve)
  write discriminant
  call tp(c4,d,b,t)      /* prime factor decomposition */
  write instable fibers (ord(p,d) > 0)
    semistable fiber at p for ord(p,c4)=0
    instable fiber at p, potentially good for ord(p,j)>=0
    instable fiber at p, potentially bad for ord(p,j) <0
  endif
end
end

```

This is the list of subroutines

- ord(p,m) valuation (exponent) of  $m$  at prime  $p$
- max(m,n) maximum of  $m$  and  $n$
- sig(z) sign of  $z$
- out(d) formatting routine
- tp(m,n,a,t) prime factor decomposition of the pair  $(m,n)$  into  $a$  prime factors, exponents in the matrix  $t$

4.2. ELLIP. The list of subroutines is the same as above, plus the *Euclidean* algorithm mod(a,n,q,r) and the reduction routine red(w,x), which, in fact, is *not called*.

- ord(p,m) valuation (exponent) of  $m$  at prime  $p$
- mod(a,n,q,r) Euclidean algorithm (one step)  $a = q \cdot n + r$
- red(w,x) reduction routine
- max(m,n) maximum of  $m$  and  $n$
- sig(z) sign of  $z$
- out(d) formatting routine
- tp(m,n,a,t) prime factor decomposition of the rational  $m/n$  into  $a$  prime factors, exponents in the matrix  $t$

It appears that the main routine is not completed: the calculation of a minimal WEIERSTRASS equation was never done (the pieces that appear in the code are commented out).

Nevertheless, this is what I have:

**comment** *ellip*

*Version: 11.3.1980*

*Weierstrass-Gleichungen ueber Z*

*singulaer: Knoten / Spitze*

*elliptisch:*

*J - Invariante*

*Diskriminante D*

*Singulaere Faser ueber p <=> P-ORD(D) > 0*

*Typ \* semistabil* *P-ORD(J.D) = 0*

*+ instabil, potentiell gut* *P-ORD(J) > 0*

*++ instabil, potentiell schlecht* *P-ORD(J.D) > 0, P-ORD(J) < 0*

*Fuehrer-Exponent F(P) = 1 bei Typ \**

*F(P) >= 2 bei Typ +, ++ (und = 2, wenn p > 3)*

*minimale Gleichung ? ;*

**begin**

```

integer a1, a2, a3, a4, a6, b2, b4, b6, b8, c4, c6, d, e, b, v;
integer array t[0:2, 1:15], w, x[1:5];

integer procedure ord(p, m);
begin
  integer e;
  e := 0;
w: if m / 'p*p = m then
  begin
    e := e + 1;
    m := m / p;
    goto w
  end;
  ord := e
end;

procedure mod(a, n, q, r);
  integer a, n, q, r;
begin
  q := entier(a/n);
  r := a - n*q;
  if n < 2*r then
  begin
    q := q + 1;
    r := r - n;
  end;
end;

procedure red(w, x);
  integer array w, x;
begin
  integer r, s, t;

  mod(w[1], 2, s, x[1]);
  mod(w[3] + s*w[1] - s*s, 3, r, x[3]);
  mod(w[2] - r*w[1], 2, t, x[2]);
  x[4] := w[4] + s*w[2] - 2*r*w[3] + (t - r*s)*w[1] + 3*r*r - 2*s*t;
  x[5] := w[5] - r*w[4] + r*r*w[3] - r*r*r + t*w[2] - r*t*w[1] - t*t;
end;

integer procedure max(m, n);
begin
  max := if m < n then n else m;
end;

procedure sig(z);
begin
  if z < 0 then output(4, '- ')
    else output(4, '+ ')
end;

```

```

procedure out(d);
begin
  'switch' s := s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12;
  goto s[entier(ln(d)/ln(10)) + 1];
  s1:  output(4,'d',d);  goto p;
  s2:  output(4,'2d',d); goto p;
  s3:  output(4,'3d',d); goto p;
  s4:  output(4,'4d',d); goto p;
  s5:  output(4,'5d',d); goto p;
  s6:  output(4,'6d',d); goto p;
  s7:  output(4,'7d',d); goto p;
  s8:  output(4,'8d',d); goto p;
  s9:  output(4,'9d',d); goto p;
  s10: output(4,'10d',d); goto p;
  s11: output(4,'11d',d); goto p;
  s12: output(4,'12d',d); goto p;
  p:
end;

procedure tp(m,n,a,t);
integer a; integer array t;

begin
  integer q,r,s;

  s := max(1,entier((sqrt(max(m,n)) - 1)/2));
  begin
    integer array l[1:s];

    for q := 1 step 1 until s do l[q] := 1;
    for q := 1 step 1 until s do
      if l[q] = 1 then
        begin
          for r := 3*q + 1 step 2*q + 1 until s do
            l[r] := 0;
          if m // (2*q + 1) * (2*q + 1) < m and
              n // (2*q + 1) * (2*q + 1) < n then l[q] := 0
        end;

    if m // 2*2 = m or n // 2*2 = n then
      begin
        a := 1;
        t[0,a] := 2;
        t[1,a] := ord(2,m);
        t[2,a] := ord(2,n)
      endelse a := 0;

    for q := 1 step 1 until s do
      if l[q] = 1 then
        begin
          a := a + 1;
          t[0,a] := 2*q + 1;

```

```

        t[1,a] := ord(2*q + 1,m);
        t[2,a] := ord(2*q + 1,n)
    end;

    if m > 1 and n = 1 then
    begin
        a := a + 1;
        t[0,a] := m;
        t[1,a] := 1;
        t[2,a] := 0
    end;

    if m = 1 and n > 1 then
    begin
        a := a + 1;
        t[0,a] := n;
        t[1,a] := 0;
        t[2,a] := 1
    end;

    if m > 1 and n > m then
    begin
        a := a + 2;
        t[0,a-1] := m; t[0,a] := n;
        t[1,a-1] := 1; t[1,a] := 0;
        t[2,a-1] := 0; t[2,a] := 1
    end;

    if n > 1 and m > n then
    begin
        a := a + 2;
        t[0,a-1] := n; t[0,a] := m;
        t[1,a-1] := 0; t[1,a] := 1;
        t[2,a-1] := 1; t[2,a] := 0
    end;

    if m > 1 and m = n then
    begin
        a := a + 1;
        t[0,a] := m;
        t[1,a] := 1;
        t[2,a] := 1
    end;
end;

output(4, 'gib den modus ein;');
output(4, '/0 bei dateneingabe ueber eine datei auf den kanal 1');
output(4, '/1 bei eingabe einzelner gleichungen');
output(4, '/2 bei eingabe aller gleichungen ueber konsole');
read(v); weiter:
if v = 0 then input(1, 'zz0', e);

```

```

if v = 1 then e := 1;
if v = 2 then begin
  output(4, '/', 'Wieviele Gleichungen werden eingegeben ?');
  read(e) end;

begin
  integer array a[1:5,1:e]; integer i,k,m,n,jz,jn;
  boolean minimal;

  if v > 0 then
  begin
    output(4, '/', 'gib die Koeffizienten so ein');
    output(4, '/', 'a1 a3 a2 a4 a6');
    for k := 1 step 1 until e do
      for i := 1, 2, 3, 4, 5 do read(a[i,k]);
  endelse
  begin
    for k := 1 step 1 until e do
      input(1, 'd,d,-d,-3zd,-4zd', a[1,k], a[2,k], a[3,k], a[4,k], a[5,k]);
  end;

  k := 0;
mark: k := k + 1;

  a1 := w[1] := a[1,k];
  a3 := w[2] := a[2,k];
  a2 := w[3] := a[3,k];
  a4 := w[4] := a[4,k];
  a6 := w[5] := a[5,k];
  b2 := a1*a1 + 4*a2;
  b4 := a1*a3 + 2*a4;
  b6 := a3*a3 + 4*a6;
  b8 := b2*a6 - a1*a3*a4 + a2*a3*a3 - a4*a4;
  c4 := b2*b2 - 24*b4;
  c6 := -b2*b2*b2 + 36*b2*b4 - 216*b6;
  d := -b2*b2*b8 - 8*b4*b4*b4 - 27*b6*b6 + 9*b2*b4*b6;

  output(4, '//, '=====');

  output(4, '/', 'YY ');
  if a1 = 0 then goto m3;
  sig(a1); output(abs(a1)); output(4, '*XY ');
m3: if a3 = 0 then goto m2;
  sig(a3); output(abs(a3)); output(4, '*Y ');
m2: output(4, '= XXX ');
  if a2 = 0 then goto m4;
  sig(a2); output(abs(a2)); output(4, '*XX ');
m4: if a4 = 0 then goto m6;
  sig(a4); output(abs(a4)); output(4, '*X ');
m6: if a6 = 0 then goto disk;
  sig(a6); output(abs(a6));

```

```

disk: if d = 0 then
  begin output(4, '/ 's i n g u l a e r !');
    if c4 = 0 then output(4, ' ( Spitze )')
      else output(4, ' ( Knoten )');
    goto stop
  end;

m := max(1,abs(c4)); n := abs(d);

tp(m,n,b,t);

output(4, '//, 'J - Invariante:');
output(4,/, '-----');
output(4,/, 'j = '); sig(c4/d);
if c4 = 0 then
  begin output(4, '0 '); goto delta end;
jz := jn := 1;
for i := 1 step 1 until b do
  begin
    integer exp;
    exp := 3*t[1,i] - t[2,i];
    if exp = 0 then goto inf;
    out(t[0,i]);
    output(4, '**[ ');
    sig(exp); out(abs(exp));
    output(4, ' ] * ');
    if exp > 0 then jz := jz*t[0,i]**exp
      else jn := jn*t[0,i]**(-exp);
  end;
inf:
end;
output(4, '= '); sig(c4/d);
out(jz); output(4, '/ '); out(jn);

delta: output(4, '//, 'Diskriminante:');
output(4,/, '-----');
output(4,/, 'Delta = '); sig(d);
for i := 1 step 1 until b do
  if t[2,i] > 0 then
  begin
    out(t[0,i]);
    output(4, '**[ ');
    out(t[2,i]);
    output(4, ' ] * ');
  end;
  end;
output(4, '= ');
sig(d); out(abs(d));

output(4,/, 'singulaere Fasern ueber :');
for i := 1 step 1 until b do
  if t[2,i] > 0 then
  begin
    output(4, '/ '); out(t[0,i]);
  end;

```



```

if c4 = 0 then
  output(4, ' (+) instabile faser, potentiell gut')
  else
    begin
      if t[1,i] = 0 then
        output(4, ' (*) semistabile Faser');
      if 3*t[1,i] >= t[2,i] then
        output(4, ' (+) instabile Faser, potentiell gut');
      if t[1,i] > 0 and 3*t[1,i] < t[2,i] then
        output(4, ' (+*) instabile Faser, potentiell schlecht');
    end;
  end;

minimal := true; comment
for i := 1 step 1 until b do
  if t[2,i] > 0 then
    begin
      if t[0,i] > 3 and t[1,i] > 3 and t[2,i] > 11 then
        if c4 = 0 then
          begin
            if t[0,i] > 3 and t[2,i] > 11 then
              minimal := false; comment
            if t[0,i] = 3 and t[2,i] > 11 then

              if t[0,i] > 3 and t[1,i] > 3 and t[2,i] > 11 then
                minimal := false;
            end;
          end;
        end;
      if k < e then goto mark;
    end
    output(4, '//, '====='');
    if v > 0 then
      begin
        output(4, '//, 'Sollen weitere Gleichungen gerechnet werden ?');
        output(4, ' (nein = 0, ja = 1)');
        read(v);
        if v = -1 then begin v := 0; goto weiter end;
        if v > 0 then goto weiter;
      end;
    end
  end Version vom 11.3.80

```

4.3. **Elliptic Code in C.** The previous Algol program has been ported into C (still without the reduction routine for *non-minimal* WEIERSTRASS equations, which is incomplete in the Algol program). The only difference is that in Algol it is possible to nest procedures, which allows dynamical definition<sup>1</sup> of arrays depending on parameters. The procedure `tp(m, n, a, t)` for the prime factorisation in the Algol program does this quite clever, by decomposing  $c_4$  and  $\Delta$  in the same pass, which makes it easy to write the  $j$ -invariant in reduced form (canceling common factors).

The corresponding procedure here is `fact(N, d[], p[], e[])`, where  $N$  is factorised into the vector `p[]` of primes with exponents in the vector `e[]`:  $N = p_1^{e_1} \cdots p_t^{e_t}$ .

<sup>1</sup>in the advent of C99 this is now also feasible in C - it remains to be added in my program

The array `d[]` consists of the test primes with  $d[0] = 2, d[1] = 3, \dots, d[155612] = 2097169$ , exhausting all primes  $p < 2^{21}$  (we have in fact that  $p = 2097169 > 2^{21}$ ). The routine to generate the prime divisors uses the sieve of ERATOSTHENES (see section 2), the factorisation into primes uses an algorithm of KNUTH (*The Art of Computer Programming* [1, vol. 2, §4.5.4], Algorithm A).

```

/* -----
Module:      ell.c

Description:

    The Tate formulas for plane Weierstrass models
    of elliptic curves over rational integers

Input:  coefficients  a1, a3, a2, a4, a6 (rational integers)

Output: prime factor decomposition of c4, c6, D and j
        reduction fiber type at bad reduction
        (Neron fiber type -- to be added) (conductor -- to be added),

Subroutines: sig, sign, display, sieve, fact, fiber

$Date: 2012-12-02 16:28:11 +0100 (Sun, 02 Dec 2012) $
$Revision: 316 $
Version: 1.7

-----

Copyright (C) 2002-2013 Berndt E. Schwerdtfeger

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

----- */

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int main(void)
{
    // constants

```

```

#define L      1024           // 2^10
#define M      1048584       // 2^20 + 8
#define YES    1             // flag for primes
#define NO     0             // flag for composite numbers

// variables

int s,t,i;
long a1, a3, a2, a4, a6;
long long b2, b4, b6, b8, c4, c6, d6, d8, D;
unsigned long p[30];        // array of prime factors
unsigned long q[30];        // array of bad primes
int e[30];                  // array of exponents
int f[30];                  // array of exponents
unsigned long d[155620];    // array of test divisors

// prototype statements for functions

int sig(long long);
int sign(long long);
int sieve(unsigned long []);
int fact( unsigned long long, unsigned long [], unsigned long [], int []);
int fiber(unsigned long, int, unsigned long [], int []);
void display(long long, unsigned long []);

// start of main processing

t = sieve(d);                // initialize test divisors
                             // t = 155612, p = 2097169 > 2^21

printf("Enter the coefficients of the Weierstrass equation\n");
printf("yy+a_1.xy+a_3.y=-xxx+a_2.xx+a_4.x+a_6\n\n");

scanf("%d%d%d%d%d", &a1, &a3, &a2, &a4, &a6);

// the Weierstrass-Tate formulaire

b2 = a1*a1 + 4*a2;
b4 = a1*a3 + 2*a4;
b6 = a3*a3 + 4*a6;
b8 = b2*a6 - a1*a3*a4 + a2*a3*a3 - a4*a4;

c4 = b2*b2 - 24*b4;
c6 = - b2*b2*b2 + 36*b2*b4 - 216*b6;

d6 = b2*b4 - 18*b6;
d8 = b2*b2*b4 - 48*b4*b4 + 18*b2*b6;

D = - b2*b2*b8 + b4*b4*b4 + 36*b4*b8 - 27*b6*b6 ;

// display the Weierstrass equation of the curve

```

```

printf("\nyy_");
if (a1) {sig(a1); printf("_dxy_",abs(a1));}
if (a3) {sig(a3); printf("_dy_",abs(a3));}
printf("=_xxx_");
if (a2) {sig(a2); printf("_dxx_",abs(a2));}
if (a4) {sig(a4); printf("_dx_",abs(a4));}
if (a6) {sig(a6); printf("_d_",abs(a6));}

if (D) {
  printf("_elliptic_curve\n");
//  printf("j - invariant: %d / %d \n", c4*c4*c4,D);

  printf("\nc4=_");
  display(c4,d);

  printf("\nc6=_");
  display(c6,d);

  printf("\nD=_");
  display(D,d);

// fiber type at bad reduction primes

  printf("\n\n**_reduction_fiber_type**\n");

  t = fact(sign(c4)*c4, d, p, e);
  t = fact(sign(D)*D, d, q, f);

// in q[1], ... , q[t] are the different primes of bad reduction

  for (t=1; t<=q[0]; t++)
    switch (fiber(q[t],f[t],p,e)) {
      case +1:
        printf("unstable_fiber_at_%u_(cusp),_potentially_good_\n", q[t]);
        break;
      case -1:
        printf("unstable_fiber_at_%u_(cusp),_potentially_bad", q[t]);
        printf("_..._and_more_details_here\n");
        break;
      case 0:
        printf("semi-stable_fiber_at_%d_(node)\n", q[t]);
        break;
    }
  }
  else {
    printf("_singular_curve_");
    if (c4)
      printf("node\n");
    else
      printf("cusp\n");
  }
}

```

```

    return 0;
}

/* -----
**
** Subroutines sig, sign
**
** Description: sign of an integer (symbol resp. value)
**
** Input: N (integer)
**
** Output: '+' or '-' resp. return value +1 or -1
** ----- */

int sig(long long N)
{
    if (N<0)
        printf("-");
    else
        printf("+");
}

int sign(long long N)
{
    if (N<0)
        return -1;
    else
        return +1;
}

/* -----
**
** Subroutines display
**
** Description:
**
** Input: x (integer), d (array of test divisors)
**
** Output: prime factor decomposition in exponential notation
** ----- */

void display(long long x, unsigned long d[])
{
    int s,t;
    unsigned long q[30];
    int f[30];
    if (x) {
        t = fact(sign(x)*x, d, q, f); // factorize the number
        printf("%"PRIu64"=", x);
        sig(x);
    }
}

```

```

    if (t) {
        for (s=1; s<t ; s++)
            printf("%u(%u) * _",q[s], f[s]);
            printf("%u(%u)",q[t], f[t]);
        }
        else
            printf("1");           // if no factors, it is 1
    } else
        printf("0");
}

/* -----
**
** Subroutine sieve
**
** Description: int sieve(unsigned long d[]);
**
**      Generating array of test primes via sieve of Eratosthenes
**
**      Input: none
**
**      Output: return value, vector d[] of test divisors
** ----- */

int sieve(unsigned long d[])
{
    // sieve of Eratosthenes

    unsigned int a, i;           // loop ind, exponent

    /* the array prime[x] test for primality of 2*x + 1 */
    char prime[M+1];           // byte array for flags

    for (a = 0; a <= M; a++)           // initialize ...
        prime[a] = YES;           // ... to everything prime
    for (a = 1; a <= L; a++)           // go thru all primes and
        if (prime[a])           // throw out the multiples
            for (i = 2*a*(a+1); i <= M; i += 2*a+1)
                prime[i] = NO;           // multiples are not prime

    // populate the vector of test divisors

    i = 0;
    d[i++] = 2;           // set the even prime

    for (a = 1; a<=M; a++)
        if (prime[a])
            d[i++] = 2*a + 1;           // set the uneven primes

```

```

    return i;
}

/* -----
**
** Subroutine fact
**
** Description:
**   The number N is factored into primes
**    $N = p[1]**e[1] * \dots * p[t]**e[t]$ 
**
**   Input: N (integer), test divisors in vector d[]
**
**   Output: vector p[], vector e[]
**           the vector p[] contains the different primes
**           the vector e[] contains the orders of N at these primes
**           p[0] = t = return value = number of primes (also t = 0)
**           (if N = 0 then e[0] = 1, otherwise e[0] = 0)
** ----- */

int fact(unsigned long long N, unsigned long d[], unsigned long p[], int e[])
{
    // this is Knuth's algorithm, modified with exponents

    unsigned int i, t, k;
    unsigned long long n, q, r;

    n = N; // A1 initialize
    i = t = k = 0;

    while (n > 1) { // A2 n = 1 ?
        q = n / d[k]; // A3 divide
        r = n - q * d[k];
        if (r == 0) { // A4 zero remainder ?
            if (i == 0)
                p[++t] = d[k]; // A5 new factor found
            e[t] = ++i; // increase exponent
            n = q;
        } // return to A2
        else {
            if (q > d[k]) { // A6 low quotient ?
                k = k + 1; // no, .. try next
                i = 0;
            } // return to A3
            else { // yes, ..
                p[++t] = n; // A7 n is prime
                e[t] = 1; // exponent is 1
                n = 1; // terminate
            }
        }
    }
}

```

```

    }
    p[0] = t; // number of factors
    if (n)
        e[0] = 0; // e[0] discriminates
    else
        e[0] = 1; // .. if n vanishes

    return t;
}

/* -----
**
** Subroutine fiber
**
** Description:
** fiber returns the fiber type
** +0 semi-stable fiber (node)
** +1 unstable fiber (cusp), but potentially good
** -1 unstable fiber (cusp), and potentially bad
**
** Input: prime q (of bad reduction), exponent f (of D)
** vector p[], exponent vector e[] (of c4)
**
** Output: return value is fiber type
** ----- */

int fiber(unsigned long q, int f, unsigned long p[], int e[])
{
    unsigned int t;
    int type = 0; // default type ss

    if (e[0])
        type = 1; // if c4 = 0
    else {
        for (t=1; t<=p[0]; t++) // test for common prime
            if (q == p[t]) { // if found,
                type = sign(3*e[t]-f); // set type accordingly
                break;
            }
    }
    return type;
}

```

**4.4. Calculation of a resultant.** In fact, in this section there is no computer algorithm, only a normal calculation of a resultant.

I make use of TATE's *formulaire*, as it is used by TATE in [8] or in the appendix to LANG [3], or in his letter to CASSELS in [2]. See also SILVERMAN [7, chap. III, §1].

In the formula for duplication of points on an elliptic curve we have for  $P = (x, y)$  and  $[2]P = (x_2, y_2)$ :

$$x_2 = \frac{x^4 - b_4x^2 - 2b_6x - b_8}{4x^3 + b_2x^2 + 2b_4x + b_6}$$



(see SILVERMAN [7, chap. III, §2, 2.3d]).

We are going to calculate the resultant  $R = R((1, 0, -b_4, -2b_6, -b_8), (4, b_2, 2b_4, b_6))$  of the two polynomials occurring in the numerator and denominator. We will simplify this determinant before multiplying it out, and also make use of  $4b_8 = b_2b_6 - b_4^2$  at times.

The resultant is (cf. LANG *Algebra* [4, IV. §8]):

$$\begin{aligned}
 R &= \begin{vmatrix} 1 & 0 & -b_4 & -2b_6 & -b_8 & 0 & 0 \\ 0 & 1 & 0 & -b_4 & -2b_6 & -b_8 & 0 \\ 0 & 0 & 1 & 0 & -b_4 & -2b_6 & -b_8 \\ 4 & b_2 & 2b_4 & b_6 & 0 & 0 & 0 \\ 0 & 4 & b_2 & 2b_4 & b_6 & 0 & 0 \\ 0 & 0 & 4 & b_2 & 2b_4 & b_6 & 0 \\ 0 & 0 & 0 & 4 & b_2 & 2b_4 & b_6 \end{vmatrix} = \\
 &= \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -4 & -b_2 & -6b_4 & 1 & 0 & 0 & 0 \\ 0 & -4 & -b_2 & 0 & 1 & 0 & 0 \\ 0 & 0 & -4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & -b_4 & -2b_6 & -b_8 & 0 & 0 \\ 0 & 1 & 0 & -b_4 & -2b_6 & -b_8 & 0 \\ 0 & 0 & 1 & 0 & -b_4 & -2b_6 & -b_8 \\ 4 & b_2 & 2b_4 & b_6 & 0 & 0 & 0 \\ 0 & 4 & b_2 & 2b_4 & b_6 & 0 & 0 \\ 0 & 0 & 4 & b_2 & 2b_4 & b_6 & 0 \\ 0 & 0 & 0 & 4 & b_2 & 2b_4 & b_6 \end{vmatrix} = \\
 &= \begin{vmatrix} b_2b_4 + 9b_6 & 3b_2b_6 + 5b_4^2 & b_2b_8 + 12b_4b_6 & 6b_4b_8 \\ 6b_4 & b_2b_4 + 9b_6 & 3b_2b_6 - b_4^2 & b_2b_8 \\ b_2 & 6b_4 & 9b_6 & 4b_8 \\ 4 & b_2 & 2b_4 & b_6 \end{vmatrix}
 \end{aligned}$$

We simplify this slightly and develop the determinant according to its first row:

$$\begin{aligned}
 R &= \begin{vmatrix} 1 & 0 & -b_4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} b_2b_4 + 9b_6 & 3b_2b_6 + 5b_4^2 & b_2b_8 + 12b_4b_6 & 6b_4b_8 \\ 6b_4 & b_2b_4 + 9b_6 & 3b_2b_6 - b_4^2 & b_2b_8 \\ b_2 & 6b_4 & 9b_6 & 4b_8 \\ 4 & b_2 & 2b_4 & b_6 \end{vmatrix} \\
 &= \begin{vmatrix} 9b_6 & 3b_2b_6 - b_4^2 & b_2b_8 + 3b_4b_6 & 2b_4b_8 \\ 6b_4 & b_2b_4 + 9b_6 & 3b_2b_6 - b_4^2 & b_2b_8 \\ b_2 & 6b_4 & 9b_6 & 4b_8 \\ 4 & b_2 & 2b_4 & b_6 \end{vmatrix} = \\
 &= 9b_6 \begin{vmatrix} b_2b_4 + 9b_6 & 3b_2b_6 - b_4^2 & b_2b_8 \\ 6b_4 & 9b_6 & 4b_8 \\ b_2 & 2b_4 & b_6 \end{vmatrix} - (3b_2b_6 - b_4^2) \begin{vmatrix} 6b_4 & 3b_2b_6 - b_4^2 & b_2b_8 \\ 4 & 2b_4 & b_6 \end{vmatrix} + \\
 &+ (b_2b_8 + 3b_4b_6) \begin{vmatrix} 6b_4 & b_2b_4 + 9b_6 & b_2b_8 \\ b_2 & 6b_4 & 4b_8 \\ 4 & b_2 & 2b_4 \end{vmatrix} - 2b_4b_8 \begin{vmatrix} 6b_4 & b_2b_4 + 9b_6 & 3b_2b_6 - b_4^2 \\ b_2 & 6b_4 & 9b_6 \\ 4 & b_2 & 2b_4 \end{vmatrix}
 \end{aligned}$$

Writing out the determinants

$$\begin{aligned}
 R &= 9b_6(9(b_2b_4 + 9b_6)b_6^2 + 4(3b_2b_6 - b_4^2)b_2b_8 + 12b_2b_4^2b_8 - 9b_2^2b_6b_8 - \\
 &- 6(3b_2b_6 - b_4^2)b_4b_6 - 8(b_2b_4 + 9b_6)b_4b_8) - \\
 &- (3b_2b_6 - b_4^2)(54b_4b_6^2 + 16(3b_2b_6 - b_4^2)b_8 + 2b_2^2b_4b_8 - \\
 &- 36b_2b_6b_8 - (3b_2b_6 - b_4^2)b_2b_6 - 48b_4^2b_8) + \\
 &+ (b_2b_8 + 3b_4b_6)(36b_4^2b_6 + 16(b_2b_4 + 9b_6)b_8 + b_2^3b_8 - \\
 &- 24b_2b_4b_8 - (b_2b_4 + 9b_6)b_2b_6 - 24b_2b_4b_8) - \\
 &- 2b_4b_8(72b_4^3 + 36(b_2b_4 + 9b_6)b_6 + b_2^2(3b_2b_6 - b_4^2) -
 \end{aligned}$$

$$\begin{aligned}
& -24(3b_2b_6 - b_4^2)b_4 - 2b_2b_4(b_2b_4 + 9b_6) - 54b_2b_4b_6 = \\
& = 9b_6(3b_2^2b_6b_8 - 9b_2b_4b_6^2 + 6b_4^3b_6 - 72b_4b_6b_8 + 81b_6^3) - \\
& - (3b_2b_6 - b_4^2)(2b_2^2b_4b_8 - 3b_2^2b_6^2 + b_2b_4^2b_6 + 12b_2b_6b_8 - 64b_4^2b_8 + 54b_4b_6^2) + \\
& + (b_2b_8 + 3b_4b_6)(b_2^3b_8 - b_2^2b_4b_6 - 32b_2b_4b_8 - 9b_2b_6^2 + 36b_4^2b_6 + 144b_6b_8) - \\
& - 2b_4b_8(3b_2^3b_6 - 3b_2^2b_4^2 - 108b_2b_4b_6 + 96b_4^3 + 324b_6^2) =
\end{aligned}$$

use  $4b_8 = b_2b_6 - b_4^2$

$$\begin{aligned}
& = 9b_6(3b_2^2b_6b_8 - 27b_2b_4b_6^2 + 24b_4^3b_6 + 81b_6^3) - \\
& - (3b_2b_6 - b_4^2)(2b_2^2b_4b_8 - 18b_2b_4^2b_6 + 16b_4^4 + 54b_4b_6^2) + \\
& + (b_2b_8 + 3b_4b_6)(b_2^3b_8 - 9b_2^2b_4b_6 + 8b_2b_4^3 + 27b_2b_6^2) - \\
& - 6b_2^3b_4b_6b_8 + 6b_2^2b_4^3b_8 + 54b_2^2b_4^2b_6^2 - 102b_2b_4^4b_6 - 162b_2b_4b_6^3 + 48b_4^6 + 162b_4^3b_6^2 = \\
& = 27b_2^2b_6^2b_8 - 243b_2b_4b_6^3 + 216b_4^3b_6^2 + 729b_6^4 - \\
& - 6b_2^3b_4b_6b_8 + 2b_2^2b_4^3b_8 + 54b_2^2b_4^2b_6^2 - 66b_2b_4^4b_6 - 162b_2b_4b_6^3 + 16b_4^6 + 54b_4^3b_6^2 + \\
& + b_4^2b_6^2 - 6b_2^3b_4b_6b_8 - 27b_2^2b_4^2b_6^2 + 8b_2^2b_4^3b_8 + 24b_2b_4^4b_6 + 27b_2^2b_6^2b_8 + 81b_2b_4b_6^3 - \\
& - 6b_2^3b_4b_6b_8 + 6b_2^2b_4^3b_8 + 54b_2^2b_4^2b_6^2 - 102b_2b_4^4b_6 - 162b_2b_4b_6^3 + 48b_4^6 + 162b_4^3b_6^2 = \\
& = b_4^2b_6^2 - 18b_2^3b_4b_6b_8 + 81b_2^2b_4^2b_6^2 + 16b_2^2b_4^3b_8 + 54b_2^2b_6^2b_8 - \\
& - 144b_2b_4^4b_6 - 486b_2b_4b_6^3 + 432b_4^3b_6^2 + 64b_4^6 + 729b_6^4 = \Delta^2
\end{aligned}$$

where  $\Delta = -b_2^2b_8 + 9b_2b_4b_6 - 8b_4^3 - 27b_6^2$  is the discriminant of the elliptic curve.

#### REFERENCES

- [1] Donald E. Knuth, *The Art of Computer Programming*, Addison–Wesley, 1997/1998.
- [2] Willem Kuyk and B. J. Birch (eds.), *Modular Functions of One Variable IV*, Lecture Notes in Math., vol. 476, Springer, 1975.
- [3] Serge Lang, *Elliptic Functions*, Graduate Texts in Mathematics, vol. 112, Springer, 1987.
- [4] ———, *Algebra*, 3rd ed., Graduate Texts in Mathematics, vol. 211, Springer, 2002.
- [5] Berndt E. Schwerdtfeger, *Über Kurven ohne rationale Punkte*, August 1978, unpublished.
- [6] ———, *The Zeta–Function of  $x^4 + y^4 + z^4 = 0$*  (2003), available at <https://berndt-schwerdtfeger.de/wp-content/uploads/pdf/v4.pdf>.
- [7] Joseph H. Silverman, *The Arithmetic of Elliptic Curves*, Graduate Texts in Mathematics, vol. 106, Springer, 1986.
- [8] John T. Tate, *The Arithmetic of Elliptic Curves*, *Inventiones math.* **23** (1974), 179–206.