

FactInt

Advanced Methods for Factoring Integers

1.6.3

15 November 2019

Stefan Kohl

Stefan Kohl

Email: stefan@gap-system.org

Homepage: <https://stefan-kohl.github.io/>

Abstract

This package for GAP 4 provides a general-purpose integer factorization routine, which makes use of a combination of factoring methods. In particular it contains implementations of the following algorithms:

- Pollard's $p - 1$
- Williams' $p + 1$
- Elliptic Curves Method (ECM)
- Continued Fraction Algorithm (CFRAC)
- Multiple Polynomial Quadratic Sieve (MPQS)

It also contains code by Frank Lübeck for making use of Richard P. Brent's tables of factors of integers of the form $b^k \pm 1$. FactInt is completely written in the GAP language and contains / requires no external binaries. It needs GAPDoc 1.6 [LN17] or higher. FactInt must be installed in the pkg subdirectory of the GAP distribution.

Copyright

© 1999 – 2017 by Stefan Kohl.

FactInt is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 2 of the License, or (at your option) any later version.

FactInt 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 License for more details.

For a copy of the GNU General Public License, see the file GPL in the etc directory of the GAP distribution or see <http://www.gnu.org/licenses/gpl.html>.

Acknowledgements

I would like to thank Bettina Eick and Steve Linton for their support and many interesting discussions.

Contents

1	Preface	4
2	The General Factorization Routine	5
2.1	The method for Factors	5
2.2	Getting information about the factoring process	7
3	The Routines for Specific Factorization Methods	8
3.1	Trial division	8
3.2	Pollard's $p - 1$	8
3.3	Williams' $p + 1$	9
3.4	The Elliptic Curves Method (ECM)	10
3.5	The Continued Fraction Algorithm (CFRAC)	11
3.6	The Multiple Polynomial Quadratic Sieve (MPQS)	12
4	How much Time does a Factorization take?	13
4.1	Timings for the general factorization routine	13
4.2	Timings for the ECM	13
4.3	Timings for the MPQS	14
	References	15
	Index	16

Chapter 1

Preface

Factoring large integers is a computationally very difficult problem, and there is no general factorization algorithm known which can be used for factoring products of two primes with more than about 100 decimal digits each on currently existing computers. But there are methods (not algorithms in the sense that it is guaranteed that they will give the desired result after a finite number of steps) for factoring integers with prime factors being far beyond the range where trial division is feasible.

One important class of such methods is based on exponentiation in suitably chosen groups acting on subsets of the k -fold cartesian product of the set of residue classes $(\text{mod } n)$, where n denotes the number to be factored. Representatives of this class are the Elliptic Curves Method (ECM), Pollard's $p - 1$ and Williams' $p + 1$. These methods have the important property that they find smaller factors usually considerably faster than larger ones. This however comes along with the drawback of suboptimal performance for large factors.

The other important class consists of the so-called factor base methods. Their run time depends only on the size of the number n to be factored, and not on the size of its factors. Factor base methods compute factorizations of perfect squares $(\text{mod } n)$ over an appropriately chosen factor base. A factor base is a set of small prime numbers, or of prime ideals in the case of the Generalized Number Field Sieve. The factor base methods use these factorizations to determine a pair of integers (x, y) such that x^2 and y^2 are congruent $(\text{mod } n)$, but $\pm x$ and $\pm y$ are not. In this situation, taking $\text{gcd}(n, x - y)$ will yield a nontrivial divisor of n . Representatives of this class are the Continued Fraction Algorithm (CFRAC), the Multiple Polynomial Quadratic Sieve (MPQS) and the already mentioned Generalized Number Field Sieve (GNFS). The latter has the asymptotically lowest average-case complexity of all factoring methods known today. It has however the drawback of being more efficient than the MPQS only for numbers with more than about 100 decimal digits, which is probably not within the feasible range of such a function implemented in **GAP**. The first two methods are implemented in this package.

Except of the “naive” methods like trial division and some “historical” methods, the only method which I am aware of that does not fit into one of the two classes mentioned above is Pollard's Rho, which is already implemented in the **GAP** Library.

With respect to the current state-of-the-art in integer factorization, see the [Factoring Challenge](#) of the RSA Laboratories.

Chapter 2

The General Factorization Routine

2.1 The method for Factors

The `FactInt` package provides a better method for the operation `Factors` for integer arguments, which supersedes the one included in the `GAP` Library:

2.1.1 Factors (`FactInt`'s method, for integers)

▷ `Factors(n)` (method)

Returns: a sorted list of the prime factors of n .

The returned factors pass the built-in probabilistic primality test of `GAP` (`IsProbablyPrimeInt`, Baillie-PSW Primality Test; see the `GAP` Reference Manual). If the method fails to compute the prime factorization of n , an error is signalled. The same holds for all other factorization routines provided by this package. It follows a rough description how the factorization method works:

First of all, the method checks whether $n = b^k \pm 1$ for some b, k and looks for factors corresponding to polynomial factors of $x^k \pm 1$. Provided that b and k are not too large, the factors that do not correspond to polynomial factors are taken from Richard P. Brent's Factor Tables [Bre04]. The code for accessing these tables has been contributed by Frank Lübeck.

Then the method uses trial division and a number of cheap methods for various common special cases. After the small and other “easy” factors have been found this way, `FactInt`'s method searches for “medium-sized” factors using Pollard's Rho (by the library function `FactorsRho`, see the `GAP` Reference Manual), Pollard's $p - 1$ (see `FactorsPminus1` (3.2.1)), Williams' $p + 1$ (see `FactorsPplus1` (3.3.1)) and the Elliptic Curves Method (ECM, see `FactorsECM` (3.4.1)) in this order.

If there is still an unfactored part remaining after that, it is factored using the Multiple Polynomial Quadratic Sieve (MPQS, see `FactorsMPQS` (3.6.1)).

The following options are interpreted:

TDHints

A list of additional trial divisors. This is useful only if certain primes p are expected to divide n with probability significantly larger than $\frac{1}{p}$.

RhoSteps

The number of steps for Pollard's Rho.

RhoCluster

The number of steps between two gcd computations in Pollard's Rho.

Pminus1Limit1 / Pminus1Limit2

The first- / second stage limit for Pollard's $p - 1$ (see [FactorsPminus1 \(3.2.1\)](#)).

Pplus1Residues

The number of residues to be tried by Williams' $p + 1$ (see [FactorsPplus1 \(3.3.1\)](#)).

Pplus1Limit1 / Pplus1Limit2

The first- / second stage limit for Williams' $p + 1$ (see [FactorsPplus1 \(3.3.1\)](#)).

ECMCurves

The number of elliptic curves to be tried by the Elliptic Curves Method (ECM) (see [FactorsECM \(3.4.1\)](#)). Also admissible: a function that takes the number n to be factored as an argument and returns the desired number of curves to be tried.

ECMLimit1 / ECMLimit2

The initial first- / second stage limit for ECM (see [FactorsECM \(3.4.1\)](#)).

ECMDelta

The increment per curve for the first stage limit in ECM. The second stage limit is adjusted appropriately (see [FactorsECM \(3.4.1\)](#)).

ECMDeterministic

If true, ECM chooses its curves deterministically, i.e. repeatable (see [FactorsECM \(3.4.1\)](#)).

FBMethod

Specifies which of the factor base methods should be used to do the “hard work”. Currently implemented: "CFRAC" and "MPQS" (see [FactorsCFRAC \(3.5.1\)](#) and [FactorsMPQS \(3.6.1\)](#), respectively). Default: "MPQS".

For the use of the **GAP Options Stack**, see Chapter *Options Stack* in the **GAP Reference Manual**.

Setting *RhoSteps*, *Pminus1Limit1*, *Pplus1Residues*, *Pplus1Limit1*, *ECMCurves* or *ECMLimit1* equal to zero switches the respective method off. The method chooses defaults for all option values that are not explicitly set by the user. The option values are also interpreted by the routines for the particular factorization methods described in the next chapter.

Example

```
gap> Factors( Factorial(44) + 1 );
[ 694763, 9245226412016162109253, 413852053257739876455072359 ]
gap> Factors( 2^997 - 1 );
[ 167560816514084819488737767976263150405095191554732902607,
  79934306053602222928609369601238840619880168466272137576868879760059\
  3002563860297371289151859287894468775962208410650878341385577817736702\
  2158878920741413700868182301410439178049533828082651513160945607018874\
  830040978453228378816647358334681553 ]
```

The above method for `Factors` calls the following function, which is the actual “working horse” of this package:

2.1.2 FactInt (factorization of an integer)

▷ `FactInt(n)` (function)

Returns: a list of two lists, where the first list contains the determined prime factors of *n* and the second list contains the remaining unfactored parts of *n*, if there are any.

This function interprets all options which are interpreted by the method for `Factors` described above. In addition, it interprets the options `cheap` and `FactIntPartial`. If the option `cheap` is set, only usually cheap factorization attempts are made. If the option `FactIntPartial` is set, the factorization process is stopped before invoking the (usually time-consuming) MPQS or CFRAC, if the number of digits of the remaining unfactored part exceeds the bound passed as option value `MPQSLimit` or `CFRACLimit`, respectively.

`Factors(n)` is equivalent to `FactInt(n : cheap := false, FactIntPartial := false) [1]`.

Example

```
gap> FactInt( Factorial(300) + 1 : cheap );
[ [ 461, 259856122109, 995121825812791, 3909669044842609,
    4220826953750952739, 14841043839896940772689086214475144339 ],
  [ 104831288231765723173983836560438594053336296629073932563520618687\
    9287645058010688827246061541065631119345674081834085960064144597037243\
    9235869682208979384309498719255615067943353399357029226058930732298505\
    5816977495398426741656633461747046623641451042655247093315505417820370\
    9451745871701742000546384614472756584182478531880962594857275869690727\
    9733563594352516014206081210368516157890709802912711149521530885498556\
    1244667790208245620301404499928532222524585946881528337257061789593197\
    99211283640357942345263781351 ] ]
```

2.2 Getting information about the factoring process

Optionally, the `FactInt` package prints information on the progress of the factorization process:

2.2.1 InfoFactInt (FactInt's Info class)

▷ `InfoFactInt` (info class)

▷ `FactIntInfo(level)` (function)

This `Info` class allows to monitor what happens during the factoring process.

If `InfoLevel(InfoFactInt) = 1`, then basic information about the factoring techniques used is displayed. If this `InfoLevel` has value 2, then additionally all “relevant” steps in the factoring algorithms are mentioned. If it is set equal to 3, then large amounts of details of the progress of the factoring process are shown.

Enter `FactIntInfo(level)` to set the `InfoLevel` of `InfoFactInt` to the positive integer *level*. The call `FactIntInfo(level)`; is equivalent to `SetInfoLevel(InfoFactInt, level)`;

The informational output is usually not literally the same in each factorization attempt to a given integer with given parameters. For a description of the `Info` mechanism, see Section *Info Functions* in the GAP Reference Manual.

Chapter 3

The Routines for Specific Factorization Methods

Descriptions of the factoring methods implemented in this package can be found in [Bre89] and [Coh93]. Cohen's book contains also descriptions of the other methods mentioned in the preface.

3.1 Trial division

3.1.1 FactorsTD (trial division)

▷ FactorsTD(n [, *Divisors*]) (function)

Returns: a list of two lists: The first list contains the prime factors found, and the second list contains remaining unfactored parts of n , if there are any.

This function tries to factor n by trial division. The optional argument *Divisors* is the list of trial divisors. If not given, it defaults to the list of primes $p < 1000$.

Example

```
gap> FactorsTD(12^25+25^12);  
[ [ 13, 19, 727 ], [ 5312510324723614735153 ] ]
```

3.2 Pollard's $p - 1$

3.2.1 FactorsPminus1 (Pollard's p-1)

▷ FactorsPminus1(n [[, a], *Limit1*[, *Limit2*]]) (function)

Returns: a list of two lists: The first list contains the prime factors found, and the second list contains remaining unfactored parts of n , if there are any.

This function tries to factor n using Pollard's $p - 1$. It uses a as base for exponentiation, *Limit1* as first stage limit and *Limit2* as second stage limit. If the function is called with three arguments, these arguments are interpreted as n , *Limit1* and *Limit2*. Defaults are chosen for all arguments which are omitted.

Pollard's $p - 1$ is based on the fact that exponentiation (mod n) can be done efficiently enough to compute $a^{k!} \bmod n$ for sufficiently large k in a reasonable amount of time. Assume that p is a prime

factor of n which does not divide a , and that $k!$ is a multiple of $p - 1$. Then Lagrange's Theorem states that $a^{k!} \equiv 1 \pmod{p}$. If $k!$ is not a multiple of $q - 1$ for another prime factor q of n , it is likely that the factor p can be determined by computing $\gcd(a^{k!} - 1, n)$. A prime factor p is usually found if the largest prime factor of $p - 1$ is not larger than $Limit2$, and the second-largest one is not larger than $Limit1$. (Compare with `FactorsPplus1` (3.3.1) and `FactorsECM` (3.4.1).)

Example

```
gap> FactorsPminus1( Factorial(158) + 1, 100000, 1000000 );
[ [ 2879, 5227, 1452486383317, 9561906969931, 18331561438319,
    4837142997094837608115811103417329505064932181226548534006749213\
    4508231090637045229565481657130504121732305287984292482612133314325471\
    3674832962773107806789945715570386038565256719614524924705165110048148\
    7161609649806290811760570095669 ], [ ] ]
gap> List( last[ 1 ] {[ 3, 4, 5 ]}, p -> Factors( p - 1 ) );
[ [ 2, 2, 3, 3, 81937, 492413 ], [ 2, 3, 3, 3, 5, 7, 7, 1481, 488011 ]
  , [ 2, 3001, 7643, 399613 ] ]
```

3.3 Williams' $p + 1$

3.3.1 FactorsPplus1 (Williams' $p+1$)

▷ `FactorsPplus1(n[[, Residues], Limit1[, Limit2]])` (function)

Returns: a list of two lists: The first list contains the prime factors found, and the second list contains remaining unfactored parts of n , if there are any.

This function tries to factor n using Williams' $p + 1$. It tries *Residues* different residues, and uses *Limit1* as first stage limit and *Limit2* as second stage limit. If the function is called with three arguments, these arguments are interpreted as n , *Limit1* and *Limit2*. Defaults are chosen for all arguments which are omitted.

Williams' $p + 1$ is very similar to Pollard's $p - 1$ (see `FactorsPminus1` (3.2.1)). The difference is that the underlying group here can either have order $p + 1$ or $p - 1$, and that the group operation takes more time. A prime factor p is usually found if the largest prime factor of the group order is at most $Limit2$ and the second-largest one is not larger than $Limit1$. (Compare also with `FactorsECM` (3.4.1).)

Example

```
gap> FactorsPplus1( Factorial(55) - 1, 10, 10000, 100000 );
[ [ 73, 39619, 277914269, 148257413069 ],
  [ 106543529120049954955085076634537262459718863957 ] ]
gap> List( last[ 1 ], p -> [ Factors( p - 1 ), Factors( p + 1 ) ] );
[ [ [ 2, 2, 2, 3, 3 ], [ 2, 37 ] ],
  [ [ 2, 3, 3, 31, 71 ], [ 2, 2, 5, 7, 283 ] ],
  [ [ 2, 2, 2207, 31481 ], [ 2, 3, 5, 9263809 ] ],
  [ [ 2, 2, 47, 788603261 ], [ 2, 3, 5, 13, 37, 67, 89, 1723 ] ] ]
```

3.4 The Elliptic Curves Method (ECM)

3.4.1 FactorsECM (Elliptic Curves Method, ECM)

▷ `FactorsECM(n[, Curves[, Limit1[, Limit2[, Delta]]]])` (function)

▷ `ECM(n[, Curves[, Limit1[, Limit2[, Delta]]]])` (function)

Returns: a list of two lists: The first list contains the prime factors found, and the second list contains remaining unfactored parts of n , if there are any.

This function tries to factor n using the Elliptic Curves Method (ECM). The argument *Curves* is the number of curves to be tried. The argument *Limit1* is the initial first stage limit, and *Limit2* is the initial second stage limit. The argument *Delta* is the increment per curve for the first stage limit. The second stage limit is adjusted appropriately. Defaults are chosen for all arguments which are omitted.

`FactorsECM` recognizes the option *ECMDeterministic*. If set, the choice of the curves is deterministic. This means that in repeated runs of `FactorsECM` the same curves are used, and hence for the same n the same factors are found after the same number of trials.

The Elliptic Curves Method is based on the fact that exponentiation in the elliptic curve groups $E(a,b)/n$ can be performed fast enough to compute for example $g^{k!}$ for k large enough (e.g. 100000 or so) in a reasonable amount of time and without using much memory, and on Lagrange's Theorem. Assume that p is a prime divisor of n . Then Lagrange's Theorem states that if $k!$ is a multiple of $|E(a,b)/p|$, then for any elliptic curve point g , the power $g^{k!}$ is the identity element of $E(a,b)/p$. In this situation -- under reasonable circumstances -- the factor p can be determined by taking an appropriate gcd.

In practice, the algorithm chooses in some sense "better" products P_k of small primes rather than $k!$ as exponents. After reaching the first stage limit with P_{Limit1} , it considers further products $P_{Limit1}q$ for primes q up to the second stage limit *Limit2*, which is usually set equal to something like 100 times the first stage limit. The prime q corresponds to the largest prime factor of the order of the group $E(a,b)/p$.

A prime divisor p is usually found if the largest prime factor of the order of one of the examined elliptic curve groups $E(a,b)/p$ is at most *Limit2* and the second-largest one is at most *Limit1*. Thus trying a larger number of curves increases the chance of factoring n as well as choosing a larger value for *Limit1* and/or *Limit2*. It turns out to be not optimal either to try a large number of curves with very small *Limit1* and *Limit2* or to try only a few curves with very large limits. (Compare with `FactorsPminus1` (3.2.1).)

The elements of the group $E(a,b)/n$ are the points (x,y) given by the solutions of $y^2 = x^3 + ax + by$ in the residue class ring (mod n), and an additional point ∞ at infinity, which serves as identity element. To turn this set into a group, define the product (although elliptic curve groups are usually written additively, I prefer using the multiplicative notation here to retain the analogy to `FactorsPminus1` (3.2.1) and `FactorsPplus1` (3.3.1)) of two points p_1 and p_2 as follows: If $p_1 \neq p_2$, let l be the line through p_1 and p_2 , otherwise let l be the tangent to the curve C given by the above equation in the point $p_1 = p_2$. The line l intersects C in a third point, say p_3 . If l does not intersect the curve in a third affine point, then set $p_3 := \infty$. Define $p_1 \cdot p_2$ by the image of p_3 under the reflection across the x -axis. Define the product of any curve point p and ∞ by p itself. This -- more or less obviously, checking associativity requires some calculation -- turns the set of points on the given curve into an abelian group $E(a,b)/n$.

However, the calculations are done in projective coordinates to have an explicit representation of the identity element and to avoid calculating inverses (mod n) for the group operation. Otherwise this

would require using an $O((\log n)^3)$ -algorithm, while multiplication (mod n) is only $O((\log n)^2)$. The corresponding equation is given by $bY^2Z = X^3 + aX^2Z + XZ^2$. This form allows even more efficient computations than the Weierstrass model $Y^2Z = X^3 + aXZ^2 + bZ^3$, which is the projective equivalent of the affine representation $y^2 = x^3 + ax + by$ mentioned above. The algorithm only keeps track of two of the three coordinates, namely X and Z . The curves are chosen in a way that ensures the order of the corresponding group to be divisible by 12. This increases the chance that it is smooth enough to find a factor of n . The implementation follows the description of R. P. Brent given in [Bre96], pp. 5 -- 8. In terms of this paper, for the second stage the “improved standard continuation” is used.

Example

```
gap> FactorsECM(2^256+1, 100, 10000, 1000000, 100);
[ [ 1238926361552897,
    93461639715357977769163558199606896584051237541638188580280321 ]
, [ ] ]
```

3.5 The Continued Fraction Algorithm (CFRAC)

3.5.1 FactorsCFRAC (Continued Fraction Algorithm, CFRAC)

▷ FactorsCFRAC(n)

(function)

▷ CFRAC(n)

(function)

Returns: a list of the prime factors of n .

This function tries to factor n using the Continued Fraction Algorithm (CFRAC), also known as Brillhart–Morrison Algorithm. In case of failure an error is signalled.

Caution: The run time of this function depends only on the size of n , and not on the size of the factors. Thus if a small factor is not found during the preprocessing which is done before invoking the sieving process, the run time is as long as if n would have two prime factors of roughly equal size.

The Continued Fraction Algorithm tries to find integers x and y such that $x^2 \equiv y^2 \pmod{n}$, but not $\pm x \equiv \pm y \pmod{n}$. In this situation, taking $\gcd(x - y, n)$ yields a nontrivial divisor of n . For determining such a pair (x, y) , the algorithm uses the continued fraction expansion of the square root of n . If x_i/y_i is a continued fraction approximation of the square root of n , then $c_i := x_i^2 - ny_i^2$ is bounded by a small constant times the square root of n . The algorithm tries to find as many c_i as possible which factor completely over a chosen factor base (a list of small primes) or with only one factor not in the factor base. The latter ones can be used if and only if a second c_i with the same “large” factor is found. Once enough values c_i have been factored, as a final stage Gaussian Elimination over GF(2) is used to determine which of the congruences $x_i^2 \equiv c_i \pmod{n}$ have to be multiplied together to obtain a congruence of the desired form $x^2 \equiv y^2 \pmod{n}$. Let M be the corresponding matrix. Then the entries of M are given by $M_{ij} = 1$ if an odd power of the j -th element of the factor base divides the i -th usable factored value, and $M_{ij} = 0$ otherwise. To obtain the desired congruence, it is necessary that the rows of M are linearly dependent. In other words, this means that the number of factored c_i needs to be larger than the rank of M , which is approximately given by the size of the factor base. (Compare with FactorsMPQS (3.6.1).)

Example

```
gap> FactorsCFRAC( Factorial(34) - 1 );
[ 10398560889846739639, 28391697867333973241 ]
```

3.6 The Multiple Polynomial Quadratic Sieve (MPQS)

3.6.1 FactorsMPQS (Multiple Polynomial Quadratic Sieve, MPQS)

▷ `FactorsMPQS(n)` (function)

▷ `MPQS(n)` (function)

Returns: a list of the prime factors of n .

This function tries to factor n using the Single Large Prime Variation of the Multiple Polynomial Quadratic Sieve (MPQS). In case of failure an error is signalled.

Caution: The run time of this function depends only on the size of n , and not on the size of the factors. Thus if a small factor is not found during the preprocessing which is done before invoking the sieving process, the run time is as long as if n would have two prime factors of roughly equal size.

The intermediate results of a computation can be saved by interrupting it with `[Ctrl] [C]` and calling `Pause()`; from the break loop. This causes all data needed for resuming the computation again to be pushed as a record `MPQSTmp` on the options stack. When called again with the same argument n , `FactorsMPQS` takes the record from the options stack and continues with the previously computed factorization data. For continuing the factorization process in another session, one needs to write this record to a file. This can be done by the function `SaveMPQSTmp(filename)`. The file written by this function can be read by the standard `Read`-function of `GAP`.

The Multiple Polynomial Quadratic Sieve tries to find integers x and y such that $x^2 \equiv y^2 \pmod{n}$, but not $\pm x \equiv \pm y \pmod{n}$. In this situation, taking $\gcd(x - y, n)$ yields a nontrivial divisor of n . For determining such a pair (x, y) , the algorithm chooses polynomials f_a of the form $f_a(r) = ar^2 + 2br + c$ with suitably chosen coefficients a , b and c which satisfy $b^2 \equiv n \pmod{a}$ and $c = (b^2 - n)/a$. The identity $a \cdot f_a(r) = (ar + b)^2 - n$ yields a congruence \pmod{n} with a perfect square on one side and $a \cdot f_a(r)$ on the other. The algorithm uses a sieving technique similar to the Sieve of Eratosthenes over an appropriately chosen sieving interval to search for factorizations of values $f_a(r)$ over a chosen factor base. Any two factorizations with the same single “large” factor which does not belong to the factor base can also be used. Taking more polynomials and hence shorter sieving intervals has the advantage of having to factor smaller values $f_a(r)$ over the factor base.

Once enough values $f_a(r)$ have been factored, as a final stage Gaussian Elimination over $\text{GF}(2)$ is used to determine which congruences have to be multiplied together to obtain a congruence of the desired form $x^2 \equiv y^2 \pmod{n}$. Let M be the corresponding matrix. Then the entries of M are given by $M_{ij} = 1$ if an odd power of the j -th element of the factor base divides the i -th usable factored value, and $M_{ij} = 0$ otherwise. To obtain the desired congruence, it is necessary that the rows of M are linearly dependent. In other words, this means that the number of usable factorizations of values $f_a(r)$ needs to be larger than the rank of M . The latter is approximately equal to the size of the factor base. (Compare with `FactorsCFRAC` (3.5.1).)

Example

```
gap> FactorsMPQS( Factorial(38) + 1 );
[ 14029308060317546154181, 37280713718589679646221 ]
```

Chapter 4

How much Time does a Factorization take?

4.1 Timings for the general factorization routine

A few words in advance: In general, it is not possible to give a precise prediction for the CPU time needed for factoring a given integer. This time depends heavily on the sizes of the factors of the given number and on some other properties which cannot be tested before actually doing the factorization. But nevertheless, rough run time estimates can be given for numbers with factors of given orders of magnitude.

After casting out the small and other “easy” factors -- which should not take more than at most a few minutes for numbers of “reasonable” size -- the general factorization routine uses first ECM (see `FactorsECM` (3.4.1)) for finding factors very roughly up to the third root of the remaining composite and then the MPQS (see `FactorsMPQS` (3.6.1)) for doing the “rest” of the work. The latter is often the most time-consuming part.

In the sequel, some timings for the ECM and for the MPQS are given. These methods are by far the most important ones with respect to run time statistics (the $p \pm 1$ -methods (see `FactorsPminus1` (3.2.1) and `FactorsPplus1` (3.3.1)) are only suitable for finding factors with certain properties and `CFRAC` (see `FactorsCFRAC` (3.5.1)) is just a slower predecessor of the MPQS). All absolute timings are given for a Pentium 200 under Windows as a reference machine (this was a fast machine at the time the first version of this package has been written).

4.2 Timings for the ECM

The run time of `FactorsECM` depends mainly on the size of the factors of the input number. On average, finding a 12-digit factor of a 100-digit number takes about 1 min 40 s, finding a 15-digit factor of a 100-digit number takes about 10 min and finding an 18-digit factor of a 100-digit number takes about 50 min. A general rule of thumb is the following: one digit more increases the run time by a bit less than a factor of two. These timings are very rough, and they may vary by a factor of 10 or more. You can compare trying an elliptic curve with throwing a couple of dice, where a success corresponds to the case where all of them show the same side -- it is possible to be successful with the first trial, but it is also possible that this takes much longer. In particular, all trials are independent of one another. In general, ECM is superior to Pollard’s Rho for finding factors with at least 10 decimal

digits. In the same time needed by Pollard's Rho for finding a 13-digit factor one can reasonably expect to find a 17-digit factor when using ECM, for which Pollard's Rho in turn would need about 100 times as long as ECM. For larger factors this difference grows rapidly. From theory it can be said that finding a 20-digit factor requires about 500 times as much work as finding a 10-digit factor, finding a 30-digit factor requires about 160 times as much work as finding a 20-digit factor and finding a 40-digit factor requires about 80 times as much work as finding a 30-digit factor.

The default parameters are optimized for finding factors with about 15 -- 35 digits. This seems to be a sensible choice, since this is the most important range for the application of ECM. The function `FactorsECM` usually gives up when the input number n has two factors which are both larger than its third root. This is of course only a "probabilistic" statement. Sometimes -- but seldom -- the remaining composite has 3 factors, 4 factors should occur (almost) never.

The user can of course specify other parameters than the default ones, but giving timings for all possible choices is obviously impossible. The interested reader should follow the references given in the bibliography at the end of this manual for getting information on how many curves with which parameters are usually needed for finding factors of a given size. This depends mainly on the distribution of primes, respectively of numbers with prime factors not exceeding a certain bound.

For benchmarking purposes, the amount of time needed for trying a single curve with given smoothness bounds for a number of given size is suited best. A typical example is the following: one curve with $(Limit1, Limit2) = (100000, 10000000)$ applied to a 100-digit integer requires a total of 10 min 20 s, where 6 min 45 s are spent for the first stage and 3 min 35 s are spent for the second stage. The time needed for the first stage is approximately linear in $Limit1$ and the time needed for the second stage is a bit less than linear in $Limit2$.

4.3 Timings for the MPQS

The run time of `FactorsMPQS` depends only on the size of the input number, and not on the size of its factors. Rough timings are as follows: 90 s for a 40-digit number, 10 min for a 50-digit number, 2 h for a 60-digit number, 20 h for a 70-digit number and 100 h for a 75-digit number. These timings are much more precise than those given for ECM, but they may also vary by a factor of 2 or 3 depending on whether a good factor base can be found without using a large multiplier or not. A general rule of thumb is the following: 10 digits more cause 10 times as much work. For benchmarking purposes, precise timings for some integers are given: $38! + 1$ (45 digits, good factor base with multiplier 1): 2 min 22 s, $40! - 1$ (48 digits, not so good factor base even with multiplier 7): 8 min 58 s, cofactor of $1093^{33} + 1$ (61 digits, good factor base with multiplier 1): 1 h 12 min.

References

- [Bre89] D. M. Bressoud. *Factorization and Primality Testing*. Springer-Verlag, 1989. 8
- [Bre96] R. P. Brent. Factorization of the tenth and eleventh Fermat numbers, 1996. <ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb161tr.dvi.gz>. 11
- [Bre04] R. P. Brent. Factor tables, 2004. <http://web.comlab.ox.ac.uk/oucl/work/richard.brent/factors.html>. 5
- [Coh93] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993. 8
- [LN17] F. Lübeck and M. Neunhöffer. *GAPDoc (Version 1.6)*. RWTH Aachen, 2017. GAP package, <http://www.gap-system.org/Packages/gapdoc.html>. 2

Index

- CFRAC
 - shorthand for FactorsCFRAC, [11](#)
- Continued Fraction Algorithm (CFRAC), [11](#)
- continued fraction approximation, [11](#)
- ECM
 - shorthand for FactorsECM, [10](#)
- elliptic curve groups, [10](#)
- elliptic curve point, [10](#)
- Elliptic Curves Method (ECM), [10](#)
- FactInt
 - factorization of an integer, [7](#)
- FactIntInfo
 - setting the InfoLevel of InfoFactInt, [7](#)
- factor base, [11](#)
 - large factors, [11](#)
- Factors
 - FactInt's method, for integers, [5](#)
- FactorsCFRAC
 - Continued Fraction Algorithm, CFRAC, [11](#)
- FactorsECM
 - Elliptic Curves Method, ECM, [10](#)
- FactorsMPQS
 - Multiple Polynomial Quadratic Sieve, MPQS, [12](#)
- FactorsPminus1
 - Pollard's $p-1$, [8](#)
- FactorsPplus1
 - Williams' $p+1$, [9](#)
- FactorsTD
 - trial division, [8](#)
- first stage limit, [10](#)
- Gaussian Elimination, [11](#)
- Generalized Number Field Sieve, [4](#)
- InfoFactInt
 - FactInt's Info class, [7](#)
- information about factoring process, [7](#)
- Lagrange's Theorem, [9](#)
- MPQS
 - shorthand for FactorsMPQS, [12](#)
- Multiple Polynomial Quadratic Sieve (MPQS), [12](#)
- Pollard's $p-1$, [8](#)
- Pollard's Rho, [4](#)
- primality of the factors, [5](#)
- prime ideal, [4](#)
- projective coordinates, [10](#)
- RSA Factoring Challenge, [4](#)
- second stage limit, [10](#)
- sieving interval, [12](#)
- trial division, [8](#)
- Weierstrass model, [11](#)
- Williams' $p+1$, [9](#)