DOCUMENTATION FOR THE RATPOINTS PROGRAM

MICHAEL STOLL

1. Introduction

This paper describes the `ratpoints` program. This program tries to find all rational points within a given height bound on a hyperelliptic curve in the most efficient way possible.

2. History and Acknowledgements

This program goes back to an implementation of the 'quadratic sieving' idea by Noam Elkies that was around in the early 1990s. My own first contribution was to replace the `char` arrays that were used to store the sieving information by bit arrays, in 1995. Colin Stahlke then made use of the `gmp` library, so that points could be checked exactly, and implemented the selection of sieving primes according to their likely success rate, in 1998. After that, I successively put in numerous improvements (and some bug fixes). For details of how the program works, see Section 8 below.

Along with Noam Elkies and Colin Stahlke, I would like to thank John Cremona and Sophie Labour for bug reports and suggestions for improvements.

3. Availability

The `ratpoints-2.1.3` package can be downloaded from my homepage, see [rat].

Date: March 10, 2011.

4. Installation

This section describes the installation procedure under Linux.

4.1. Extract the archive.

```
> tar xzf ratpoints-2.1.3.tar.gz
```

This sets up a directory `ratpoints-2.1.3` containing the various files that belong to the installation.

4.2. Build the program and library.

Do

```
> cd ratpoints-2.1.3
> make all
```

This will build the library `libratpoints.a` and the executable `ratpoints`.

By default, the program will use primes up to 127. If you intend to do computations with large height bounds or with curves that you expect to have many points, it may make sense to increase the range of primes. This can be achieved via

```
> make all PRIME_SIZE=8
```

perhaps after a

```
> make distclean
```

to remove the files that were generated previously. The `PRIME_SIZE` argument can be given any value from 5 to 10; otherwise it is taken to be 7. The precise meaning is that the program will work with primes $< 2^s$, when $\texttt{PRIME\_SIZE} = s$.

The program now uses SSE instructions if available. If you don't want this, do

```
> make distclean
> make all CCFLAGS=-UUSE_SSE
```

4.3. Run a test.

Run

```
> make test
```

in the working directory. This will build an executable `rptest` and then run (and time) it. Finally, the output (which was written to a file `rptest.out`) is compared against `testbase`, which contains the output of a sample run. The two should be identical.

4.4. Install.

If you like, you can install the library, executable and header file on your system.

```
> su
> make install
```

The executable is copied to `/usr/local/bin/`, the library to `/usr/local/lib/`, and the header file to `/usr/local/include/`. You can change the `/usr/local` prefix by giving the option `INSTALL_DIR=...` to `make install`.

4.5. Debugging.

In case you found a bug and would like to find out where it comes from, or if you just want to see exactly what the program is doing, you can do

```
> make debug
```

in the working directory. This will build an executable `ratpoints-debug`, which, when run, will dump loads of output on the screen, so it is best to send the output to a file, which you can then study at leisure.

4.6. Cleaning up.

In order to get rid of the temporary files, do

```
> make clean
```

To remove everything except the files from the archive, use

```
> make distclean
```

4.7. Requirements.

You need a C compiler (like `gcc`, which is the default specified for the `CC` variable in `Makefile`; if necessary, it can be changed there).

In addition to the standard libraries, the program also requires the `gmp` (GNU multi-precision) library [gmp].

4.8. List of files.

The archive contains the following files.

- `Makefile`
- `ratpoints.h` — the header file for programs using ratpoints
- `rp-private.h, primes.h` — header files used internally
- `gen_find_points_h.c, gen_init_sieve_h.c` — short programs that write additional header files depending on the system configuration
- `sift.c, init.c, sturm.c, find_points.c` — the source code for the ratpoints library
- `main.c, rptest.c` — the source code for the `ratpoints` and `rptest` executables, respectively
- `testdata.h, testbase` — data for the test run

- `ratpoints-doc.pdf` — this documentation file
- `gpl-2.0.txt` — the GNU license that applies to this program.

5. How to use `ratpoints`

5.1. Basic operation.

Let

$$C : y^2 = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

be your curve, and let H be the bound for the denominator and absolute value of the numerator of the x-coordinate of the points you want to find. The command to search for the points is then

> ratpoints 'a_0 a_1 $\ldots a_{n-1}$ a_n' H

The first argument to `ratpoints` is the list of the coefficients, which have to be integers, are separated by spaces, and are listed starting with the constant term. There is no bound on the size of the coefficients; they are read as multi-precision integers.

The degree n of the polynomial is limited to `RATPOINTS_MAX_DEGREE` (which is defined in `ratpoints.h`), which is set to 100 by default.

The program will give an error message when the polynomial (considered as a binary form of the smallest even degree $\geq n$) is not squarefree (e.g., if you specify two leading zero coefficients).

The following subsections discuss how to modify the standard behavior. This is achieved by adding options after the two required arguments. Some of the options have arguments, others don't; the ordering of the options and option-argument pairs is arbitrary (except for options with opposite effect, where the last one given counts, and for the specification of the search intervals, which must be in order).

5.2. Early abort.

By default, `ratpoints` will search the whole region that you specify and print all the points it finds. If you just want to find *one* point (for example when dealing with 2-covering curves of elliptic curves), you can tell the program to quit after it has found one point via the `-1` option, e.g.,

```
> ratpoints '-18 116 48 -12 30' 60000000 -1
```

If you don't want to see points at infinity, specify the `-i` option. The two options can be combined: `-i -1` will stop the program after it has found one finite point.

5.3. Changing the output.

By default, `ratpoints` prints some general information before and after the points, which are given in the form `(`x `:` y `:` z`)`, one per line. Here, $(x : y : z)$ are the coordinates of the point considered as a point in the $(1, \lceil n/2 \rceil, 1)$-weighted projective plane, which is the natural ambient space for the curve. The coordinates are integers with x and z coprime and $z > 0$, or $z = 0$ and $x = 1$ (for points at infinity).

There are various ways to change this behavior.

- To suppress all output except the points, use `-q` (for *quiet*).
- To add some more output explaining what the program is doing, use `-v` (for *verbose*). This has no effect if the `-q` option is also given.
- To suppress printing of the points, use `-z`.
- If you only want to list the x-coordinates of point pairs rather than individual points, use `-y`.
- There are four options that influence the format in which the points are printed:
  `-f` *format* `-fs` *string-before* `-fm` *string-between* `-fe` *string-after*
  The arguments to all of them are strings; `"\n"`, `"\t"`, `"\\"` and `"\%"` are recognized and do what you expect. The *format* string can contain markers `%x`, `%y`, `%z` that will be replaced with the x, y, and z-coordinate of the point, respectively. The defaults are empty for *string-before*, *string-between* and *string-after*, and `"(%x : %y : %z)\n"` for *format* when `-y` is not specified; otherwise `"(%x : %z)\n"`.

  The effect is as follows. Before any point is printed, *string-before* is output. Then every point is printed according to *format*. Between any two points, *string-between* is output, and after the last point, *string-after* is output.

  As an example, consider the effect of
  `-f "[%x,%y,%z]" -fs "{" -fm "," -fe "}\n"`

5.4. Restricting the search domain.

There are two ways to restrict the domain of the search. The first is to restrict the range of denominators considered. This is done via
`-dl` $d_{\min}$ `-du` $d_{\max}$.
For example, to only look for integral points, you can say `-du 1`. By default, the lower limit is 1 and the upper limit is H.

The other way is to restrict the search to a union of (closed) intervals. If you only want to search for points in

$$[l_1, u_1] \cup [l_2, u_2] \cup \cdots \cup [l_k, u_k] ,$$

you can specify this in the form
`-l` l_1 `-u` u_1 `-l` l_2 `-u` u_2 $\ldots$ `-l` l_k `-u` u_k .
The first `-l` option is optional; l_1 defaults to $-\infty$. Similarly the last `-u` option is optional, and u_k defaults to $+\infty$. The number k of intervals is bounded by `RATPOINTS_MAX_DEGREE` (usually, 100).

5.5. Setting the parameters for the sieve.

It is possible to change the number of primes that are used in the various stages of the algorithm. This is done by the following options.
`-p` M `-N` N `-n` n

This sets the number of (odd) primes that are considered for the sieve to M, the number of primes that are actually used for the sieve to N, and the number of primes used in the first sieving stage to n. The program will, if necessary, reduce M, N, and n (in this order) to ensure that $n \le N \le M \le$ `RATPOINTS_NUM_PRIMES`. The latter is usually 30 (the number of odd primes $< 2^7$), but will be 53 (the number of odd primes $< 2^8$) if `PRIME_SIZE` is set to 8, etc.

There are two more parameters that can be set.

`-F` D

sets the maximal number of 'forbidden divisors'. If the degree is even and the leading coefficient is not a square, then the denominator of any x-coordinate of a rational point cannot be divisible by a prime p such that the leading coefficient is a non-square mod p. If the leading coefficient is divisible by p, but not a p-adic square, the denominator cannot be divisible by some power of p. The program constructs a list of such forbidden divisors against which to check the denominators, and this option specifies how many of these should be used.

The other option is

`-S` S or `-s` .

In the first form, it sets the number of refinement (interval halving) steps in the isolation of the real components to S. This part of the program computes a Sturm sequence for the polynomial and uses it in order to find a union of intervals that contains the intervals of positivity of the polynomial. This is done by a successive subdivision of the interval $]-\infty, +\infty[$. The number S sets the recursion depth. S can be omitted; then it is given a default value. The option `-s` skips the Sturm sequence computation completely. This also has the effect of removing most of the check for squarefreeness.

5.6. **Switching off optimizations.**

The options `-k` and `-j` can be used to prevent the program from reversing the polynomial, which it usually does if this will lead to faster operation (`-k`), and to prevent the program from using the Jacobi symbol test on the denominators (`-j`). This last test extends the 'forbidden divisors' method described in Subsection 5.5 above by computing the Jacobi symbol (l/d), where l is the leading coefficient and d is the odd and coprime-to-l part of the denominator. If the symbol is -1, the denominator need not be considered. The use of these options may be questionable, unless you want to see how much performance is gained by using these optimizations.

5.7. **Switching off exact testing of points.**

The option `-x` will prevent the program from checking the potential points that survive the sieve whether they really give rise to rational points. This implies that the points that are output may not actually be points. It also effectively sets the `-y` option, since the y-coordinates are not computed.

5.8. Overriding previous options.

The options -I, -Y, -Z, -S, -K, -J, -X can be used to cancel the effect of the corresponding lower-case option (and thereby restore the default behavior), when this occurs earlier in the list of options. This may be useful if you want to set a default behavior that is different from what the program does out-of-the-box (e.g., in a shell script), but want the caller to be able to override this change.

6. How to use the library

It is possible to use the ratpoints machinery from within your own programs. The library libratpoints.a provides the following functions.

```
long find_points(ratpoints_args*,
                 int proc(long, long, const mpz_t, void*, int*),
                 void*);

void find_points_init(ratpoints_args*);

long find_points_work(ratpoints_args*,
                      int proc(long, long, const mpz_t, void*, int*),
                      void*);

void find_points_clear(ratpoints_args*);
```

The passing of arguments to these functions is via the ratpoints_args structure, which is defined as follows.

```
typedef struct { mpz_t *cof; long degree; long height;
                 ratpoints_interval *domain; long num_inter;
                 long b_low; long b_high; long sp1; long sp2;
                 long array_size;
                 long sturm; long num_primes; long max_forbidden;
                 unsigned int flags; ...}
        ratpoints_args;
```

The dots at the end stand for additional fields that are used internally and are not of interest here. When calling find_points, the cof field must point to an array of size degree + 1 of properly initialized gmp integers; these are the coefficients of the polynomial. The program can alter the values of the integers in this array. To prevent this, set the RATPOINTS_NO_REVERSE bit in flags; this may result in a loss of performance, though.

height gives the height bound. It is an error for the degree or the height bound to be nonpositive; in this case the function returns the value RATPOINTS_BAD_ARGS.

The field domain must contain a pointer to an array of ratpoints_interval structures of length at least num_inter plus degree. This array gives (in its first num_inter entries) the intervals for the search region. The type ratpoints_interval is just

```
typedef struct {double low; double up;} ratpoints_interval;
```

the meaning of this should be clear. In `ratpoints`, this is set via the `-l` and `-u` options. Usually, you do not want to restrict the range of x-coordinates; then you set `num_inter = 0` (but you still have to fill `domain` with a valid pointer to at least `degree` intervals, unless you set `sturm` to a negative value!) The program may alter the values in the array *domain* points to, unless `sturm` has a negative value (which may lead to a performance loss).

`b_low` and `b_high` carry the lower and upper bounds for the denominator; if non-positive, they are set to 1 and the height bound, respectively. In `ratpoints`, these fields are set by the `-dl` and `-du` options.

`sp1` and `sp2` specify the number of primes to be used in the first sieving stage and in both sieving stages together, respectively. If negative, they are set to certain default values. In `ratpoints`, these fields are set by the `-n` and `-N` options. Similar statements are true for `num_primes` (option `-p`), `sturm` (options `-s`, `-S`) and `max_forbidden` (option `-F`). The field `array_size` specifies the maximal size (in long words) of the array that is used in the first sieving stage. If non-positive, it is set to a default value. The various default values are defined at the beginning of the header file `ratpoints.h`, where also the maximal degree of the polynomial is set.

The `flags` field holds a number of bit flags.

- `RATPOINTS_NO_CHECK` — when set, do not check whether the surviving x-coordinates give rise to rational points (set by the `-x` option to `ratpoints`).
- `RATPOINTS_NO_Y` — only list x-coordinates (in the form $(x : z) \in \mathbb{P}^1(\mathbb{Q})$) instead of actual points (with a y-coordinate); this is set by the `-y` option to `ratpoints`.
- `RATPOINTS_NO_REVERSE` — when set, do not allow reversal of the polynomial (set by the `-k` option to `ratpoints`).
- `RATPOINTS_NO_JACOBI` — when set, prevent the use of the Jacobi symbol test (set by the `-j` option to `ratpoints`).
- `RATPOINTS_VERBOSE` — when set, causes the procedure to print some output on what it is doing (set by the `-v` option to `ratpoints`).

There are some other flags that are used internally. One of them might be of interest:

- `RATPOINTS_REVERSED` — when set after the function call, this indicates that the polynomial has been reversed (and the contents of the `cof` array have been modified).

The main vehicle for passing information back to the caller is the `proc` function argument together with the pointer `info`. This function

```
int proc(long x, long z, const mpz_t y, void *info, int *quit)
```

is called whenever a point was found. `x`, `y` and `z` are the coordinates of the point (where `y` is a gmp integer). `info` is the pointer that was passed to `find_points`; this can be used to store information that should persist between calls to `proc`. If `*quit` is set to a non-zero value, this indicates that `find_points` should abort the point search and return immediately; otherwise the search continues. The return value is taken as a weight for counting the points; usually it will be 1.

The usual framework for using `find_points` is as follows.

```
(...)
#include "ratpoints.h"

(...)

mpz_t c[RATPOINTS_MAX_DEGREE+1];  /* The coefficients of f */
ratpoints_interval domain[2*RATPOINTS_MAX_DEGREE];
                     /* This contains the intervals representing the
                        search region */

/**************************************************************
 * function that processes the points                         *
 **************************************************************/

typedef struct {...} data;

int process(long x, long z, const mpz_t y, void *info0, int *quit)
{ data *info = (data *)info0;

  (...)

  return(1);
}

/**************************************************************
 * main                                                       *
 **************************************************************/

int main(int argc, char *argv[])
{
  long total, n;
  ratpoints_args args;

  long degree        = 6;
  long height        = 16383;
  long sieve_primes1 = RATPOINTS_DEFAULT_SP1;
  long sieve_primes2 = RATPOINTS_DEFAULT_SP2;
  long num_primes    = RATPOINTS_DEFAULT_NUM_PRIMES;
  long max_forbidden = RATPOINTS_DEFAULT_MAX_FORBIDDEN;
  long b_low         = 1;
  long b_high        = height;
```

```
long sturm_iter    = RATPOINTS_DEFAULT_STURM;
long array_size    = RATPOINTS_ARRAY_SIZE;
int no_check       = 0;
int no_y           = 0;
int no_reverse     = 0;
int no_jacobi      = 0;
int no_output      = 0;

unsigned int flags = 0;

data *info = malloc(sizeof(data));

/* initialise multi-precision integer variables */
for(n = 0; n <= degree; n++) { mpz_init(c[n]); }

(...)

{ /* set up polynomial */
  long k;
  for(k = 0; k < 7; k++) { mpz_set_si(c[k], ...); }

  args.cof           = &c[0];
  args.degree        = 6;
  args.height        = height;
  args.domain        = &domain[0];
  args.num_inter     = 0;
  args.b_low         = b_low;
  args.b_high        = b_high;
  args.sp1           = sieve_primes1;
  args.sp2           = sieve_primes2;
  args.array_size    = array_size;
  args.sturm         = sturm_iter;
  args.num_primes    = num_primes;
  args.max_forbidden = max_forbidden;
  args.flags         = flags;

  info->... = ...;
  (...)

  total = find_points(&args, process, (void *)info);
  if(total == RATPOINTS_NON_SQUAREFREE)
  { ... }
  if(total == RATPOINTS_BAD_ARGS)
  { ... }
```

```
    (...)

  }

  /* clean up multi-precision integer variables */
  for(n = 0; n <= degree; n++) {mpz_clear(c[n]); }

  return(0);
}
```

If points are to be searched on many curves, then it is slightly more efficient to use the sequence

```
  args.degree = degree; /* this information is needed */
  find_points_init(&args);

  for( ... )
  { ...
    total = find_points_work(&args, process, (void *)info);
    ...
  }

  find_points_clear(&args);
```

This avoids the repeated allocation and freeing of memory.

For practical examples, see `main.c` (the code that wraps `find_points` for the command line program `ratpoints`) or `rptest.c` (which runs `find_points` on some test data).

7. Fine-tuning the parameters

For large computations, it may be a good idea to try to find the best (or at least, a good) combination of parameters for the given kind of data. The default values are chosen for optimal performance on an `Intel(R) Core(TM)2 CPU T7200@2.00GHz`, for random genus 2 curves with small coefficients and a height bound of $2^{14} - 1$. For your machine and input data, other values may be better. You can replace the `testdata.h` file with your own collection of test data (and edit `rptest.c` if necessary to adapt the degree and height settings) and then time `./rptest -z` (the `-z` option suppresses the output) for various combinations of the parameter settings. `rptest` accepts most of the optional parameters of `ratpoints`, in particular the `-p`, `-N`, `-n`, `-F` and `-S` parameters, so you can easily change the parameters used on the command line. In addition, there is an option `-h` H to change the default height bound.

Once you have found a good set of parameter values for your application, you can hard-code them as defaults into `ratpoints` by changing the definitions in `ratpoints.h` (and then `make distclean all test`), or you can use them to fill the `ratpoints_args` structure for your call to `find_points`.

As a number to compare against, on my laptop with the CPU above, `make test` takes about 6.6 seconds. To put this into perspective, note that this means that some 24 points are tested on average per CPU cycle.

8. Implementation

8.1. **Overview.**

Let $F(x, z)$ be the binary form of even degree corresponding to the polynomial on the right hand side of the curve equation. The basic idea is to let run b from 1 to the height bound H, for each b, let a run from $-H$ to H, and for each coprime pair (a, b) check if $F(a, b)$ is a square.

Of course, in this form, this would take a very long time. To speed up the process, we try to eliminate quickly as many pairs (a, b) as possible before the actual test. This can be done by 'quadratic sieving' modulo several primes: if $F(a, b)$ is a square, it certainly has to be a square mod p, and so we can rule out all (a, b) that do not satisfy this condition. Another ingredient is to represent (for a fixed b) the various values of a by bits and treat all the bits in a long word (32 or 64, as the case may be, or 128 when SSE instructions are used) in parallel. For this, we organize the sieving information for each prime p into p arrays of p words each, one such array for every b mod p, such that the jth bit in this array is set if and only if $F(j, b)$ is a square mod p.

For each 'denominator' b, we then set up an array of words whose bits represent the range $-H \leq a \leq H$ (aligned so that $a = 0$ corresponds to the 0th bit of a word); the bits are initially set. Then for each of the sieving primes p, we perform a bit-wise *and* operation between this array and the sieving information at p. In a first stage, this is done on the whole array; after this first stage, each remaining ('surviving') non-zero word in the array is subject to tests with more primes. If some bits are still set after this second stage of sieving, the corresponding pairs (a, b) (if coprime) are then checked exactly.

In the following subsections, we discuss a number of improvements that were made.

8.2. **Sorting the primes.**

This idea is due to Colin Stahlke. We do not just take the first so many primes in increasing order for the sieving, but we first compute the number of points the curve has mod p for a number of primes p and then sort the primes according to the fraction of x-coordinates that give points. We then take those primes for the sieving that have the smallest fraction of 'surviving' x-coordinates. In this way, we need fewer sieving primes to achieve a comparable reduction of point tests.

8.3. **Using connected components.**

We note that we can only have points when $F(a, b)$ is non-negative. If we can determine intervals on which $f(x) = F(x, 1)$ is negative, then we do not have to look for points in these intervals. The necessary computations can be performed exactly, by computing a Sturm sequence for f and counting the number of sign changes at various points, see [Coh, Thm. 4.1.10]. This can in particular tell us whether F is negative definite, in which case we have already proved that there are no rational points in the curve. If there are real zeros, we use a subdivision method in order to find a collection of intervals containing the projections of the connected components of the curve over $\mathbb{R}$.

8.4. **Using 2-adic information.**

John Cremona suggested that in some cases, one can determine beforehand that all points will have odd 'numerators' a, and so we can pack the bits more tightly by only representing odd numbers. This approach can be extended. We first find all solutions mod 16 (higher powers of 2 would be possible, but not very likely to give an improvement). Then for every residue class mod 16 of the denominator b, we can find the residue classes mod 16 of potential numerators a. If there are none, then we can eliminate b as a denominator altogether. If all potential a's are even or odd (this will always be the case for even denominators), we can restrict the sieving to such numerators.

8.5. **Elimination of denominators.**

Depending on the equation, certain denominators can be excluded. We have seen an instance of this in the previous subsection, but we can also work with odd primes.

8.5.1. *Odd degree and ±monic.* If the polynomial has odd degree and leading coefficient ± 1, then the denominator has to be a square. This reduces the time complexity tremendously.

8.5.2. *Odd degree general.* In general, when f has odd degree, the denominator has to be 'almost' a square: it must be a square times a (squarefree) divisor of the leading coefficient, and there are further restrictions on the parity of the valuation at p, for p dividing the leading coefficient, when this valuation is sufficiently large. This gives the same type of time complexity as in the monic case, but with a larger constant.

8.5.3. *Even degree with non-square leading coefficient.* Here we can exclude denominators divisible by a prime p such that the leading coefficient is a non-square mod p. We can also in some cases rule out denominators divisible by a certain power of p when p is a prime that divides the leading coefficient (if the leading coefficient is not a p-adic square). While computing the sieving information, we make a list of such primes and prime powers, which we then use later to eliminate denominators. In addition, we use the necessary condition that the Jacobi symbol $\left(\frac{l}{b'}\right)$ must be $+1$, where l is the leading coefficient and b' is the part of b coprime to $2l$.

8.6. Reversing the polynomial.

The exclusion of denominators described above is more or less effective, depending on the situation; it is the better the earlier the case was described. Since the height of the points does not change under the transformation $(x, y) \to (1/x, y/x^{\lceil n/2 \rceil})$, we can as well search on the reversed polynomial $F(z, x)$. This will result in a speed-up if the reversed polynomial belongs to a 'better' class than the original.

8.7. Some general remarks.

Modern processors are very fast when doing basic things like moving data around between registers or simple arithmetic operations (addition, subtraction, shift, . . . , comparison). Even memory access can be fast when there is no cache miss. On the other hand, integer division is a slow process. It turned out that quite some improvement of the performance was possible by removing as many instances of integer division operations as reasonably possible.

For example, we use gcd and Jacobi symbol routines that rely (almost) entirely on differences and shifts. We compute the residue classes of b modulo the various sieving primes by addition of the difference from the last b and then correcting by subtracting p a number of times if necessary, and we implement most of the testing of 'forbidden divisors' of b using bit arrays similar to those used for sieving the numerators.

8.8. Change log.

Version 2.0 was released January 9, 2008.

Version 2.0.1 was released July 7, 2008. It fixes a bug that prevented the '`-1`' option to work properly.

Version 2.1 was released March 9, 2009. It makes use of the SSE instructions, so that the sieve can work on 128 bits in parallel (instead of on 64 or 32). On my laptop (with an Intel Core2 processor), `make test` runs about 25% faster than before.

Version 2.1.1 was released April 14, 2009. It fixes a bug that in some cases prevented an early abort when `*quit` was set by the callback function. Thanks to Robert Miller for the bug report and the fix. In addition, it is now checked that `__WORDSIZE == 64` before SSE instructions are used (in `rp-private.h`), since the program then assumes that a bit array consists of two `unsigned long`s. In this context, `__SSE2__` has been replaced by a new macro `USE_SSE`. A further fix eliminates unncessesary copying of sieving information when SSE instructions are not used (introduced in version 2.1). This was almost always harmless, but could have resulted in memory corruption in the extreme case that all the information had to be computed for all the primes.

Version 2.1.2 was released May 27, 2009. It fixes some memory leaks. Thanks to Robert Miller.

Version 2.1.3 was released September 21, 2009. The library function `find_points` should now work without any bound on the degree of the polynomial. The degree bound for the `ratpoints` executable is now set to 100 by default (specified by `RATPOINTS_MAX_DEGREE`, used to be 10).

A bug in `rptest.c` (pointed out to me by Giovanni Mascellani and Randall Rathbun) that could lead to a segmentation fault was fixed on March 10, 2011.

References

[Coh] H. Cohen, *A course in computational algebraic number theory,* Springer GTM **138**, second corr. printing, 1995.

[gmp] The GNU Multiple Precision Arithmetic Library, `http://gmplib.org/`

[rat] The `ratpoints-2.1.3` package. Available at `http://www.mathe2.uni-bayreuth.de/stoll/programs/ratpoints-2.1.3.tar.gz`

Mathematisches Institut, Universität Bayreuth, 95440 Bayreuth, Germany.

E-mail address: `Michael.Stoll@uni-bayreuth.de`