Verifying `ratpoints` in Lean: a blueprint

Michael Stoll

October 31, 2026

Contents

1 Overview 4
- 1.1 What `ratpoints` computes . . . 4
- 1.2 The four parts . . . 5
- 1.3 The specification . . . 6
- 1.4 The main theorems . . . 7
- 1.5 How the proofs go . . . 9
- 1.6 What one has to trust . . . 9
- 1.7 What is not verified . . . 10
- 1.8 How to read this blueprint . . . 11

2 The specification 13
- 2.1 The binary form . . . 13
- 2.2 The points . . . 14
- 2.3 The search region . . . 15
- 2.4 The order of `ratpoints` . . . 16
- 2.5 The options, the refusals and the outcome of a search . . . 18
- 2.6 The reference of the delivery . . . 19
- 2.7 The brute-force reference . . . 19

3 The main theorems 21
- 3.1 The search . . . 21
- 3.2 The points that a search finds . . . 22
- 3.3 The refusals . . . 23
- 3.4 The delivery and the options of the output . . . 24
- 3.5 What the program prints . . . 25
- 3.6 The search as a filter of the reference . . . 26
- 3.7 The blocks, the lag and the threads . . . 27
- 3.8 What comparator certifies . . . 28

4 The set-up 31
- 4.1 The order of the checks . . . 31
- 4.2 The squarefree binary form . . . 33
- 4.3 The reversal . . . 33
- 4.4 The remainder chain . . . 34
- 4.5 The points at infinity . . . 35
- 4.6 The exit by a prime . . . 36

5 The search region and the Sturm step **37**
5.1 The doubles . . . 37
5.2 The search intervals . . . 38
5.3 The bounds of the numerators . . . 39
5.4 The Sturm step . . . 40
5.5 Sturm's theorem and the coverage . . . 41
5.6 The region that the search uses . . . 43
5.7 Deviations . . . 43

6 The choice of the moduli and the blocks **44**
6.1 The table of a prime and the loop of the tables . . . 44
6.2 Prime powers and products . . . 45
6.3 The cost model . . . 46
6.4 The choice of the moduli and of the cut points . . . 47
6.5 The blocks of the run . . . 48
6.6 The correction of the moduli . . . 49
6.7 Why the choice does not change the output . . . 50

7 Tables and rows **51**
7.1 The tables as `ratpoints` computes them . . . 51
7.2 The closed form of a row . . . 52
7.3 The tables of rows and the builders . . . 53
7.4 The rows of a run . . . 55
7.5 Where the model differs from the C code . . . 56

8 The denominators **57**
8.1 The 2-adic information . . . 57
8.2 Odd degree: squares and divisors . . . 58
8.3 Even degree: the forbidden divisors . . . 59
8.4 The Jacobi test . . . 60
8.5 The shapes of the loop . . . 61
8.6 The order of the denominators . . . 62
8.7 Where the model differs from the C code . . . 63

9 The sieve of one denominator **64**
9.1 The sieve in the C code and in the model . . . 64
9.2 Bit arrays of words . . . 65
9.3 Reductions by a reciprocal . . . 65
9.4 The packing of the numerators . . . 66
9.5 The first stage . . . 67
9.6 The second stage and the extraction . . . 68
9.7 The test of a candidate and the exact check . . . 68
9.8 One denominator . . . 69
9.9 The stages compute the plain sieve . . . 70
9.10 Where the model differs from the C code . . . 70

10 Delivery and the program **72**
10.1 The callback and the delivery . . . 72
10.2 The stops . . . 73
10.3 The options of the output . . . 74
10.4 What a search gives back . . . 75
10.5 The report of the library . . . 75
10.6 The program . . . 76

11 The link to the C code **79**
11.1 The record . . . 79
11.2 The exclusions . . . 80
11.3 The deviations . . . 81
11.4 The differential tests . . . 82
11.5 Other values of the constants . . . 83
11.6 Continuous integration . . . 84

A The correspondence with the C code **85**

Chapter 1

Overview

`ratpoints` is a C program, built around a library function, that finds the rational points of bounded height on the curves $y^2 = F(x, z)$, where F is a binary form of even degree with integer coefficients, by sieving. This blueprint describes a Lean 4 development that verifies it: an executable model of `ratpoints` in Lean, which follows the C code component by component; proofs that the model meets a short specification; and a link from the model to the C code. This chapter states the specification and the main theorems in words, each statement linked to the node where it is stated precisely, with its Lean names; it says how the proofs go, what one has to trust and what is not verified, and how the rest of the blueprint is organized.

1.1 What `ratpoints` computes

The input is a list $c = [c_0, c_1, \dots, c_n]$ of integers, the coefficients of the polynomial $f = \sum_i c_i x^i$, constant term first, and a height bound H. Let d be the degree of f, the largest index with $c_d \neq 0$, and $D = 2\lceil d/2 \rceil$ the even number at or just above it. The curve is $y^2 = F_c(x, z)$ with the binary form

$$F_c(x, z) = \sum_{i=0}^{d} c_i \, x^i z^{D-i}$$

of even degree D. Its points lie in the weighted projective plane with the weights $(1, D/2, 1)$, where $(x : y : z)$ and $(-x : (-1)^{D/2} y : -z)$ are the same point. `ratpoints` lists the points $(a : y : b)$ with integer coordinates, normalized so that $\gcd(a, b) = 1$ and $b > 0$, or $(a, b) = (1, 0)$ for a point at infinity, whose height $\max(|a|, b)$ is at most H; a point with $y \neq 0$ comes with both signs of y. We call a the numerator and b the denominator of the point, after the rational number a/b. For instance, `ratpoints '-2 0 0 1' 130 -q` lists the points of height at most 130 on $y^2 = x^3 - 2$, among them $(129 : 3830 : 100)$, as $3830^2 = 100\,(129^3 - 2 \cdot 100^3)$.

The search is done by the library function `find_points`. It takes its arguments in a structure `ratpoints_args` (`ratpoints.h:362-400`): the coefficients and the height bound; flags that switch off the exact check of the points, the second point $(a : -y : b)$ of each pair, the reversal of the polynomial and the Jacobi test; a restriction of the search to intervals for x/z and to a range of the denominators; the depth of a Sturm step; and parameters that tune the sieve. It returns no list of points. It hands each point that it finds to a function of its caller, the *callback*, together with the caller's data, adds up the values that the calls return, and stops as soon as a call asks it to (`ratpoints.h:436-437`). The program `ratpoints` reads the coefficients, the height bound and

its options from the command line, calls the library with a callback that prints the points in a format chosen by the options, and prints a report around them.

Between the input and the points the C code does a good deal of work, and the chapters on the model follow it. It checks the input and refuses some binary forms that are not squarefree. It may replace the polynomial by its reversal, which exchanges x and z, where it expects the reversed curve to be searched faster. A Sturm step, unless it is switched off, refuses the other binary forms that are not squarefree and narrows the search intervals to those on which f may take a nonnegative value. It decides which denominators can carry a point: by the 2-adic information, by the prime factors of the leading coefficient when d is odd, and by forbidden divisors and a Jacobi symbol when d is even. It ranks small primes and some of their powers and products by a cost model computed in doubles and chooses the moduli of the three stages of its sieve; for each modulus and residue, a row of bits says which numerators can give a square value of the form modulo the modulus. For each denominator it and-s these rows into bit arrays of numerators, packed by a stride that the 2-adic information allows, with vector instructions; it tests the survivors for common factors with the denominator and against further primes, and checks the rest exactly. The loop over the denominators runs in blocks, between which `ratpoints` may correct its choice of the moduli by what the sieve has counted, and which it may hand to a pool of threads.

The C code is that of `ratpoints` 3.1.1, from its repository `https://github.com/MichaelStollBayreuth/ratpoints`. Its manual, `ratpoints-doc-3.1.pdf`, describes the program, its options and the library function. Every line number in this blueprint refers to one commit of the repository, the one that the record of the correspondence pins (Chapter 11); the link to the manual points to that commit too.

1.2 The four parts

The development has four parts, in separate folders. The specification and the model import neither each other nor the proofs, and the proofs import both; `scripts/check_layout.py` checks the imports.

(1) The *specification* (`Ratpoints/Spec`) states what `ratpoints` computes: the binary form, the normalized points of height at most H, the search region that the caller may give, Sturm's theorem as a proposition, and the list of the points in the order in which `ratpoints` finds them. It consists of four short files that use Mathlib and nothing of the model, so that they can be read on their own (Chapter 2).

(2) The *model* (`Ratpoints/Model`) is an executable version of `ratpoints` in Lean. It mirrors the hardware-independent structure of the C code function by function: the same loops, tables, bit arrays of 64-bit words and index arithmetic. It is plain code on Lean core and `Std`, without Mathlib and without contracts (its only proofs are a few decided facts about its constants, the termination of one recursion and the bounds of some array accesses), and forms a library of its own; the program `ratpoints-model` imports it alone and has the command line of `ratpoints` (Chapters 4 to 10).

(3) The *proofs* (`Ratpoints/Proof`) show that the model meets the specification. Each component of the model has a contract, and the main theorems put the contracts together (Chapter 3). Statements of general interest that Mathlib lacks, on homogenizations, Sturm sequences, sign changes, valuations and the like, are stated at Mathlib's generality in `Ratpoints/Mathlib`.

(4) The *link to the C code* (Chapter 11): a record of which Lean declaration models which lines of C, kept in paragraphs `C:` of the docstrings of the model and, for the pin, the excluded lines and the lock, in the folder `correspondence`, checked against the C code at the pinned commit; and differential tests that run `ratpoints-model` and the C program on the same inputs and compare what they print.

The model does without Mathlib so that the executable and the differential tests depend neither on Mathlib nor on the proofs. The specification does without the model so that one can read what the theorems claim without reading how `ratpoints` works.

1.3 The specification

The specification is the one part whose correctness no proof can check. Wherever there was a choice, it therefore copies what the C code computes, and Chapter 2 compares each of its clauses with `ratpoints` 3.1.1.

Points. The binary form F_c above is Definition 2.1. A triple (x, y, z) of integers is a *normalized point of height at most* H on the curve (Definition 2.4) when

$$y^2 = F_c(x, z), \qquad \gcd(x, z) = 1, \qquad z > 0 \text{ or } (x, z) = (1, 0), \qquad |x| \le H, \qquad z \le H\,.$$

The last four conditions say that the pair (x, z) is normalized and of height at most H, which is what the statement on the output without the exact check concludes. Remark 2.5 compares each clause with the C code; for instance, $\gcd(0, z) = z$, so that $x = 0$ occurs only with $z = 1$, as in the test of `ratpoints` for common factors.

Squarefreeness. The refusals speak about a second binary form, the polynomial made homogeneous of the even degree $N = 2\lceil n/2 \rceil$ that the length of the coefficient list allows,

$$\Phi_c = \sum_{i=0}^{n} c_i X^i Z^{N-i} \in \mathbb{Q}[X, Z]$$

(Definition 2.3), and about its squarefreeness in the sense of Mathlib. This is what `ratpoints` means by a polynomial that is "not squarefree as a binary form of even degree": Φ_c is squarefree exactly when f is squarefree and the leading zeros of c do not lower $\lceil n/2 \rceil$, the two conditions that the C code tests (Lemma 4.3). On every input that `ratpoints` accepts, Φ_c and F_c take the same values.

The search region. The caller may restrict the search to *search intervals* for x/z, closed intervals whose ends are C doubles, and to a range of the denominators from b_{low} to b_{high}. The specification reads a double by its 64 bits: from them it decides whether the double is a NaN and gives its value, a rational number or $\pm\infty$. A list of intervals is well formed when no end is a NaN, the lower end of each interval is at most its upper end, and each interval lies strictly below the next; this is the test by which `find_points` refuses bad arguments. A point lies in the search region when it lies at infinity, or when its denominator lies in the range and x/z lies in one of the intervals, if there are any (Definition 2.6). As the C code does no arithmetic on the ends that could decide a point, the specification can speak about their exact values (Remark 2.7).

Sturm's theorem. The Sturm step of `ratpoints` drops an interval when the numbers of sign changes of a remainder chain agree at its ends and f is negative just inside it. That no point is lost rests on Sturm's theorem, which Mathlib does not have, although it has the Sturm sequence and the sign changes of a list. The specification states the theorem as a proposition, for every squarefree real polynomial and every interval $(a, b]$ (Definition 2.8), and the statements that

need it take it as a hypothesis: those on the model only when the Sturm step is on, and the membership statement of the fast search.

The order. The list orderedPoints (Definition 2.9) gives the points in the order in which `ratpoints` finds them: for the polynomial that `ratpoints` searches, the reversal of c in three cases and c itself otherwise, its points at infinity, then its denominators in the order of the loop, the numerators of each in increasing order, $y > 0$ before $y < 0$, each point mapped back to the caller's curve after a reversal. The denominators increase unless the polynomial searched has odd degree and a leading coefficient ℓ with $|\ell| \neq 1$ that `ratpoints` can factor by trial division and that has fewer than maxDivisors $= 512$ squarefree divisors up to the largest denominator; then they are the numbers ek^2, divisor e by divisor (Definition 2.12 and Remark 2.10).

The interface of a search. Of the arguments in `ratpoints_args`, those that change the refusals, the order or what is delivered are the *options* of a search (Definition 2.11): the switch of the reversal, the depth of the Sturm step (negative for none), the search region, `RATPOINTS_NO_Y`, under which of each pair only the point with $y \geq 0$ is delivered, and `RATPOINTS_NO_CHECK`, under which the numerators that survive the three stages of the sieve and the test for common factors, and the candidate at infinity when there is a point at infinity, are delivered with $y = 0$ without the exact check. The *options of the sieve* only steer the choice of the moduli. A search refuses, for one of five reasons that mirror the errors 3, 5, 4, 8 and 9 of the program, or returns its *outcome*, what the program reads afterwards.

The two references. The reference of the delivery (Definition 2.13) is a fold of a few lines that calls the callback on the points of a list in turn, adding up the values that the calls return, until a call asks to stop; it models no C code and is the yardstick of the protocol theorem. The brute-force reference (Definition 2.14) tests every candidate of the box of height H and lists exactly the normalized points of height at most H, each once (Theorem 2.15); the proofs compare the search with it denominator by denominator. What one has to read to know what the main theorems say is listed in Section 3.8.

1.4 The main theorems

The model has three entry points, each defined from the one before (Definition 3.1): the search with a callback and a print channel for the report of the library, `Ratpoints.findPointsVerbose`, which the program runs; the search with a callback, `Ratpoints.findPointsWith`, which is the library function as it appears to a caller who does not ask for the report; and findPoints(c, H, s, o), which returns the list of the points or a refusal, the search with the callback that appends each point to a list and never stops the search. The theorems hold for every coefficient list c, every height bound H, every set s of options of the sieve and every set o of options of a search; they assume nothing about the options of the sieve, which the search normalizes before it chooses its moduli. Where Sturm's theorem is needed, it enters as the hypothesis

$$0 \leq o.\text{sturm} \;\rightarrow\; \text{SturmTheorem}\,,$$

which is void when the Sturm step is off.

Membership (Theorem 3.3). Assume Sturm's theorem when the Sturm step is on, and let the exact check be on. If findPoints(c, H, s, o) returns a list L, then a triple lies in L exactly when it is a normalized point of height at most H, with $y \geq 0$ under `RATPOINTS_NO_Y`, that lies in the search region of o. Indeed L is a permutation of the brute-force reference filtered by these two conditions.

No duplicates (Theorem 3.4). Every list that findPoints returns has no duplicates. This needs no hypothesis at all, neither Sturm's theorem nor the exact check.

The order (Theorem 3.5). Under the hypotheses of membership, L is the list orderedPoints of the specification, with the reversal allowed when o allows it and the search region is the whole box of height H, and with the largest denominator of the search as the bound of the denominators, filtered by `RATPOINTS_NO_Y` and the search region. As this list does not depend on s, the options of the sieve do not change the output with the exact check on; nor does the depth of the Sturm step, which the theorem also states in the form that the search with the step switched off returns the same list (this part is proved, not certified).

The refusals (Theorem 3.6). With the Sturm step on, findPoints refuses exactly when c has fewer than two or more than maxDegree $+ 1 = 101$ entries, $H = 0$ or $H \geq 2^{63}$ (not a C `long`), Φ_c is not squarefree, or the search intervals are not well formed. With the Sturm step off, the condition on Φ_c is replaced by the two tests that `ratpoints` makes without the remainder chain of the Sturm step, on the leading zeros and in the reversal check. A search that refused every input would contradict these statements, so that membership and the order are not vacuous.

The protocol (Theorem 3.7). For every callback and its data, the search with the callback refuses exactly when findPoints refuses, and otherwise gives the data and the count of the delivery of the list of findPoints to the callback by the reference of the delivery. So the calls of the callback are those for the points of that list, in their order, up to and including the first call that stops the search, and the statements on the list carry over to every callback, the callback of the program among them.

Without the exact check (Theorem 3.8). Under `RATPOINTS_NO_CHECK` every element of the list is a triple $(x, 0, z)$ whose pair (x, z) is normalized, of height at most H and in the search region; assuming Sturm's theorem when the Sturm step is on, every normalized point (x, y, z) of height at most H in the search region gives the element $(x, 0, z)$; and the list that the search finds with the exact check on and `RATPOINTS_NO_Y` set, with 0 in place of each y, is a sublist, so that the points come in the same order. Which other pairs pass the sieve depends on the moduli and is not specified.

What the program prints (Theorem 3.9). For a command line without `-v` that it reads, `ratpoints-model` prints the head of its report, the points of the list of findPoints that its callback keeps (under `-1` the first), in their order and in the format of the command line, between the strings that the command line sets, and the end of the report (under `-q` only the points and the strings). Without `-x`, and assuming Sturm's theorem when the Sturm step is on, the points kept are exactly the normalized points of height at most H in the search region of the command line, with the exact values of the ends that it gives, with $y \geq 0$ under `-y` and none at infinity under `-i`.

The blocks, the lag and the threads (Theorems 3.13, 3.14 and 3.15). The run delivers the points of its blocks in order, each block sifted with moduli that depend neither on the callback nor on the options of the output; a block reads only corrections of the moduli made at least adaptLag $+ 1$ blocks before it; and blocks sifted apart and delivered in their order give what the run delivers. So the output of the model, which sifts the blocks in turn on one thread, is that of `ratpoints` with any number of threads, provided that its pool of threads behaves as its code says (Section 3.7).

Comparator certifies thirteen statements: the two on the brute-force reference, membership, no duplicates, the two refusal statements, the order, the three statements without the exact check, the protocol (Section 3.8), and the two on the fast search of `ratpoints-lean` (below). The others are proved in the same library, but one has to read their Lean statements to know what they say: the forms of membership with the permutation and the finite set of the points, the part of the order theorem on the depth of the Sturm step, the theorems on the program, and the block, lag and thread lemmas.

1.5 How the proofs go

All the statements on the points come from one description of what a search delivers (Section 3.6). A test of the numerators over the denominators is *sound* when it lets through the numerator of every point. The search delivers the points at infinity and then, over the denominators of its loop in their order and the numerators of its search region in increasing order, what the exact check finds for the numerators that pass a sound test (Lemma 3.11). A sound test loses no point, and the exact check keeps only points; so with the exact check on, the list of a search is the brute-force reference, filtered by the region and by `RATPOINTS_NO_Y`, in the order of the loop and mapped back after a reversal (Lemmas 3.11 and 3.12). The test is fixed before the callback and the options of the output come into play, so that one test serves for all of them; this gives the protocol and the comparison of the searches with and without the exact check.

The chapters on the model supply the pieces: the refusals in the order of the program, and the reversal, which maps the points of the reversed curve bijectively onto those of the caller's curve (Chapter 4); the exact ranges of the numerators and the Sturm step, which keeps every point under Sturm's theorem (Chapter 5); the soundness of every sieve entry that the choice of the moduli can make and of every row that the run reads (Chapters 6 and 7); the tests on the denominators and their order (Chapter 8); the contract of the sieve of one denominator (Definition 9.11); and the block lemma, which puts the denominators together (Theorem 3.13). The two exits before the sieve, at the 2-adic information and at a prime modulo which the curve has no points, are taken only when the curve has no points at all (Lemmas 8.2 and 4.10).

Of the moduli the proofs need nothing but their soundness; so the theorems hold for every choice of the moduli and every correction between the blocks, and the cost model, computed in doubles, enters no proof (Chapter 6). The contract of each component with a loop is an `@[spec]` theorem, proved with `vcgen`, the program logic of Lean core, from loop invariants that are named definitions. The contracts of the sieve and of the rows use Lean core and `Std` only; those that reason modulo a prime, about valuations, Jacobi symbols or Sturm sequences use Mathlib.

1.6 What one has to trust

The specification. No proof can show that the specification describes `ratpoints`; one has to read it. Chapter 2 compares each clause with the C code, and three of the four files of the specification check examples of their definitions by `decide` against outputs of the C program. The specification is written with definitions of Lean core, Batteries and Mathlib, whose meaning one takes from these libraries: the greatest common divisor and the integer square root, `Squarefree` and `IsSquare`, the prime factors of a number, the sublists of a list, the extended rationals `WithBotTop`, and, in Sturm's theorem, the Sturm sequence `Polynomial.sturmSeq`, the sign variations `List.signVariations` and the cardinality `Set.ncard`. With the specification one reads the types of the model that the statements mention and the reference of the delivery (Section 3.8).

The kernel and comparator. Comparator, a checker for Lean proofs built into Lake, takes a challenge that restates the specification verbatim and states the thirteen certified theorems with `sorry`, and a solution that proves them by the theorems of the library. It accepts the solution only when each theorem states the same as in the challenge, the full bodies of the definitions that the statements reach included, the proofs use no axioms beyond `propext`, `Quot.sound` and `Classical.choice`, and, under `--paranoid`, six kernels accept them (Section 3.8). The project uses no tactic that adds an axiom, such as `native_decide` or `bv_decide`.

The fast program. The executable `ratpoints-lean` does not run the model but a search of its own, `Ratpoints.FastMask.fastPoints` (`Ratpoints/Fast/Search.lean`), built for speed in Lean:

it sieves the numerators of each class modulo 16 in chunks of bit masks held in byte arrays, with the rows of families of odd moduli, and passes the survivors to the exact check of the brute force. Without a range of the denominators it reverses the polynomial where `ratpoints` does (the model's reversal check) and maps the points back. For each denominator it searches only the numerators in the intervals that the Sturm step of the model keeps, and for even degree it skips the denominators with an odd divisor at which the Jacobi symbol of the leading coefficient is -1. The AND of a mask with a shifted row, the AND of a mask with a list of shifted rows and the next set bit of a mask are primitives in C (`Ratpoints/Fast/Mask.lean`); their Lean bodies are the reference definitions that the proofs see, and `lake exe masktest` compares the C code with them. The same sieve on natural numbers, `Ratpoints.Fast.fastPoints` (`Ratpoints/Fast/Sieve.lean`), lists the same points (`Ratpoints.FastMask.fastPoints_perm`). The two theorems of the fast search, `Ratpoints.FastMask.mem_fastPoints_iff` and `Ratpoints.FastMask.nodup_fastPoints`, say that for every coefficient list, height bound, range of the denominators and setting of its parameters it lists exactly the points of height at most H with their denominator in the range, assuming Sturm's theorem (only such points without it), each once; comparator certifies them with the eleven statements on the model and the brute force. The rest of this blueprint is about the model, whose program is `ratpoints-model`.

The executable. The statements are about the definitions of the model, and `ratpoints-model` is these definitions compiled by the compiler of Lean, with its runtime and the primitives of Lean core on machine words, arrays and floating-point numbers. That the executable runs the model is not certified. What is checked is that the model contains no `partial`, `unsafe`, `implemented_by` or `extern`, nor any experimental option, which could let the compiled code differ from the definitions that the proofs see, and that the program imports the model alone (Section 11.6). The theorems on the program speak about its pure core `Ratpoints.Cli.run`; the function `Ratpoints.Cli.main` around it writes the text once and, where `ratpoints` is ended by `SIGPIPE` when the reader of its output is gone, exits with the status 141 that the shell reports for that (Chapter 10).

Sturm's theorem. Membership, the order and the statement that every point appears without the exact check assume Sturm's theorem when the Sturm step is on, and so does the membership statement of the fast search, whose numerators rest on the Sturm step. It is a hypothesis, not an axiom: these statements hold outright with the step off, and with the step on they say something because the theorem, stated with Mathlib's Sturm sequence and sign variations, is true, the one mathematical fact taken on trust; a proof of it, in Mathlib or here, would discharge the hypothesis without changing a statement. No other certified statement needs it.

The cost model. The search ranks its candidate moduli by a cost model that computes in Lean's `Float`, as the C code computes in doubles, with the `log` and `pow` of the C library of the platform. No contract of the choice, of the blocks or of the correction reads the value of a `Float`: each states only what the natural numbers of the C code do, which holds whatever the doubles are, NaN included (Remark 6.6). So the theorems hold for every choice of the moduli that the search can make, and the floating-point arithmetic of the platform has to be trusted only for the agreement of the choice with that of the C program.

1.7 What is not verified

The link to the C code is not a proof. Lean has no semantics of C, and no theorem of this blueprint speaks about the C code. The link rests on three things (Chapter 11): the structure of the model, which mirrors the C code component by component, so that a reader can set each definition beside its lines of C; the record, in which every line of code of the ten C files lies in exactly one range, modelled by the declaration whose docstring names it or excluded with a

kind and a reason, each range locked with the hash of its text and its owners and checked by CI against the pinned commit, and in which paragraphs `Deviation:` say where the model departs from the C code (Sections 11.1 and 11.3); and the differential tests, which run both programs on the same command lines and compare the points in their order, the output byte for byte, the reports and the exit statuses: 5397 comparisons in the default suites and 13 and 92 in two further suites, none of which fails (Section 11.4). As glibc's `log` and `pow` are not correctly rounded and depend on the processor, the comparisons that depend on the choice of the moduli are made with both programs on one machine.

What the model leaves out. The record excludes, by kind, the pool of threads, the management of memory, the parts of the interface of the library that only another caller meets, such as a work space of the caller's own, the instrumentation for timing, counting and debugging, and lines without meaning for the model: includes, prototypes, hints to the compiler, the frames of the generators of tables, and macros, a field and a message that the C code never uses (Section 11.2); one more exclusion concerns other builds, below. The model allocates nothing that can fail, so it has no error for lack of memory. In place of the threads it has the block, lag and thread lemmas, whose hypotheses on the pool of threads are read off the C code (Section 3.7).

The other builds. The vector code of the C code becomes words: a bit array of the default build, a vector of 256 bits, is four consecutive 64-bit words. The builds of the C code differ in compile-time constants, such as the width of the bit arrays and the number of bit arrays that the first stage keeps in registers, which choose among the arms of the code. The model has these constants as definitions with the values of the default build, and both arms where the C code has two, but for the arm of `exact_scale` for a build without a 128-bit integer type, which the record excludes (`find_points.c:3239-3256`). No proof depends on the values of the constants beyond a few facts that are decided; a script checks this, outside CI, by building copies of the whole project with eleven other sets of values (Section 11.5). So the proofs cover the models of the builds that have `unsigned __int128`, while the tests run the default build.

The platform. The doubles of the search region are their bits, with a function for each operation that `ratpoints` applies to them, from the comparisons to `frexp`, `ldexp` and `nextafter`, and the program reads and prints numbers as glibc's `strtod` and `printf` do (Chapters 5 and 10). These functions model the platform, not `ratpoints`: they are checked against gcc and glibc on the cases at the edges, and the proofs show of them what the theorems need, but they are not proved to be what glibc computes.

What is compared but not proved. The report of the library under `-v` and the messages of the program, the numbers of the moduli used, and the pairs that pass the sieve under `RATPOINTS_NO_CHECK` beyond what Theorem 3.8 says, are compared with the C program by the differential tests. Nothing is proved about the work that a search does or about how well the choice of the moduli serves it.

1.8 How to read this blueprint

Chapter 2 states the specification, and Chapter 3 the main theorems, with the proofs in outline and what comparator certifies; a reader who wants only to know what is verified can stop there. Chapters 4 to 10 describe the model part by part, following a search from the input to the output: the set-up, the search region and the Sturm step, the choice of the moduli and the blocks, the tables and the rows, the denominators, the sieve of one denominator, and the delivery and the program. Chapter 11 describes the link to the C code, and Appendix A lists every recorded range of C.

The notation of this chapter is used throughout: $c = [c_0, \dots, c_n]$, d, D, N, f, F_c and Φ_c, the

height bound H, and b_{low} and b_{high} for the range of the denominators; a point $(a : y : b)$ has the numerator a and the denominator b.

Nodes. The definitions, lemmas, theorems and corollaries are the nodes of the dependency graph; each names its Lean declarations, which in the web version link to the API documentation. A node of the model describes one component: it names the main definition and its contract, says in words what the one computes and the other promises, precisely enough that a reader can check the Lean statement against it, and links to the C code that the definition models. Each statement is followed by its proof in outline. Remarks are not nodes; they compare the specification and the model with the C code. Most helpers appear in no node; the API documentation has them all.

The dependency graph. In the graph of the web version, a definition is a box and a lemma or theorem an ellipse; an arrow leads from a node to each node that uses it, dashed when the statement uses it and solid when only the proof does. Every node is green, and every statement dark green: its proof and the proofs of all the nodes it rests on are formalized.

Links to the C code. A link such as `find_points.c:4624-4632` shows the lines 4624 to 4632 of `find_points.c` at the pinned commit. Inside a node, every link is one of the ranges that the docstrings of its declarations record, as `scripts/correspondence.py` checks. The docstrings carry all the ranges and deviations of each declaration, and Appendix A lists every range with its owners, or with the reason for its exclusion.

Chapter 2

The specification

This chapter states what the main theorems of Chapter 3 speak about: the binary form of a coefficient list, the points of bounded height on the curve, the search region, Sturm's theorem as a proposition and the order in which `ratpoints` finds the points; and, from the model, the interface of a search (its options, its refusals and what it gives back), the reference of the delivery of the points to a callback, and the brute-force reference. The specification proper is the four files `Basic.lean`, `Region.lean`, `Sturm.lean` and `Order.lean` in `Ratpoints/Spec`, which use Mathlib and nothing of the model, so that they can be read on their own. The comparator challenge restates their definitions verbatim, all but the finite set of the points (Section 2.7), and imports the interface and the two references from the model, which uses no Mathlib, as it imports the search itself; Section 3.8 says what one has to read.

The specification is the one part whose correctness no proof can check. So wherever there was a choice, it copies what the C code computes, and each of its clauses is compared below with `ratpoints` 3.1.1 at the pinned commit. Where the library function `find_points` and the command-line program `ratpoints` differ, as in the order of the checks on the input, the specification and the model follow the program. The specification enters the chapters on the model (Chapters 4 to 10) only through the statements of Chapter 3.

Throughout, $c = [c_0, c_1, \dots, c_n]$ is a list of integers, the coefficient list of the input with the constant term first, so that $n+1$ is its length; d is its *degree*, the largest index with $c_d \neq 0$, or 0 if there is none; and

$$D = 2\lceil d/2 \rceil, \qquad N = 2\lceil n/2 \rceil$$

are d and n rounded up to even numbers. The leading zero coefficients of c *lower* $\lceil n/2 \rceil$ when $\lceil d/2 \rceil < \lceil n/2 \rceil$, that is, when $D < N$, or equivalently $d+2 \leq N$; otherwise $D = N$, or equivalently $N \leq d+1$. H is a natural number, the height bound.

2.1 The binary form

Definition 2.1 (The binary form)**.** The degree d of c is the length of c with its trailing zeros (the leading zero coefficients of the polynomial) dropped, less one (0 for a list of zeros). For integers x and z,

$$F_c(x, z) = \sum_{i=0}^{d} c_i \, x^i z^{D-i} \,,$$

the binary form of even degree D of the polynomial $f = \sum_i c_i X^i$.

Remark 2.2 (What the C code computes)**.** This is the form that the C code evaluates. The library drops the leading zero coefficients (`find_points.c:4624-4632`), so that it works with the degree d, and its exact check computes $\sum_{k\le d} c_k a^k b^{d-k}$ by Horner's rule in the numerator a, on the products $c_k b^{d-k}$ computed once for each denominator b, and multiplies by b once more when d is odd (`find_points.c:5291-5298`); the result is $F_c(a,b)$. The model evaluates the form in the same way, in the brute-force reference (Section 2.7) and in the exact check of the sieve (Chapter 9), and both evaluations are proved equal to F_c. The C code writes the degree of the form as `degree + (degree & 1)`, and the Lean definition, which copies it, as $d + (d \bmod 2)$; both are $D = 2\lceil d/2 \rceil$.

Definition 2.3 (The binary form of the refusals)**.**

$$\Phi_c = \sum_{i=0}^{n} c_i X^i Z^{N-i} \in \mathbb{Q}[X, Z]\,,$$

the polynomial made homogeneous of the even degree N that the length of its coefficient list allows. When $\lceil n/2 \rceil \le \lceil d/2 \rceil$, the two degrees agree, $D = N$, and $\Phi_c(x,z) = F_c(x,z)$ for all integers x and z.

The refusal theorems (Theorem 3.6) speak about Φ_c with Mathlib's `Squarefree`: no square of a non-unit divides it. The units of $\mathbb{Q}[X,Z]$ are the nonzero constants, so $4(X^2+Z^2)$ is squarefree (`ratpoints` accepts `4 0 4`), and 0 is not. Φ_c is squarefree exactly when $N \le d+1$ and f is squarefree (Section 4.2). The C code tests the first condition by the leading zeros (`find_points.c:4629-4631`) and the second in the remainder chain of the Sturm step, which runs only when `args->sturm` ≥ 0. With the step off, `ratpoints` refuses as not squarefree only the forms with $d+2 \le N$ and, when the reversal is allowed, the search region is the whole box and d is even, those with $c_0 = c_1 = 0$ (then $X^2 \mid \Phi_c$); it accepts other forms that are not squarefree, such as `1 0 2 0 1` or `0 0 1 1` under `-s`. Every input that `ratpoints` does not refuse has $N \le d+1$, so that there Φ_c and F_c agree.

2.2 The points

Definition 2.4 (Normalized point of height at most H)**.** A pair (x,z) of integers is *normalized and of height at most H* if

$$\gcd(x,z) = 1, \quad z > 0 \text{ or } (z = 0 \text{ and } x = 1), \quad |x| \le H, \quad z \le H\,.$$

A triple (x,y,z) of integers is a *normalized point of height at most H* on $y^2 = F_c(x,z)$ if $y^2 = F_c(x,z)$ and (x,z) is normalized and of height at most H. These points form a finite set points(c,H): $|x| \le H$, $0 \le z \le H$, and $|y| \le y^2 = F_c(x,z)$.

Remark 2.5 (The clauses compared with `ratpoints`)**.** *The normalization* is the one in which the program prints the points. As D is even, $(-x : (-1)^{D/2} y : -z)$ is the same point as $(x : y : z)$ in the weighted projective plane with the weights $(1, D/2, 1)$; as both signs of y are listed, the condition can pick $z > 0$, and $x = 1$ when $z = 0$, where $\gcd(x,0) = |x|$ forces $x = \pm 1$.

The points at infinity are therefore the triples $(1,y,0)$ with $y^2 = c_D$: the point $(1:0:0)$ for odd d, and $(1 : \pm\sqrt{c_d} : 0)$ for even d when c_d is a square. These are the points at infinity that the C code delivers (`find_points.c:4894-4899`, `find_points.c:5068-5105`) with the exact check on, without `RATPOINTS_NO_Y` and without a reversal; under `RATPOINTS_NO_CHECK` it delivers one triple with $y = 0$, under `RATPOINTS_NO_Y` only the one with $\sqrt{c_d}$, and after a reversal the

points at infinity of the polynomial searched, as the points $(0 : y : 1)$ of the caller's curve (`find_points.c:5079-5079`; Section 4.5).

Both signs of y are listed, as the program prints both; the option of the library that keeps only the point with $y \geq 0$ is an option of the search (Definition 2.11), not a part of the predicate. *The box is closed*: $|x| = H$ and $z = H$ are included, as in the C code, whose numerators run over $-H, \ldots, H$ and whose denominators over $1, \ldots, H$. *Coprimality* is `Int.gcd`, with $\gcd(0, z) = z$, so that $x = 0$ occurs only as $(0, y, 1)$, as in the test `relprime` of the C code (`sift.c:318-343`), which for the numerator 0 returns whether the denominator is 1. *The height bound applies to the points at infinity too* ($|1| \leq H$), so that for $H = 0$ there are no points at all; the program refuses $H = 0$, so this never shows, and it keeps the predicate uniform.

The pair alone is what the program prints when the exact check is off, as triples $(x, 0, z)$ that need not be points; the statement on that output concludes `Ratpoints.IsNormalizedPair` for each of them (Theorem 3.8). The predicates are decidable, so that `Basic.lean` checks facts taken from the output of the program by `decide`: for instance that $(129, 3830, 100)$, which `ratpoints '-2 0 0 1' 130 -q` prints, is a point of height at most 130 of $y^2 = x^3 - 2$.

2.3 The search region

The caller of `find_points` may restrict the search to *search intervals* $[l, u]$ for x/z, closed intervals whose ends are C `doubles`, and to a range of the denominators from b_{low} to b_{high}. The specification reads a double by its 64 bits only to decide whether it is a NaN and to give its value in $\mathbb{Q}$ extended by $\bot$ and $\top$ (Mathlib's `WithBotTop` $\mathbb{Q}$); every other predicate reads the values.

Definition 2.6 (The search region)**.** For the bits $w < 2^{64}$ of an IEEE binary64 double, with the exponent field $e = \lfloor w/2^{52} \rfloor \bmod 2^{11}$, the fraction $t = w \bmod 2^{52}$ and the sign $\sigma = -1$ for $w \geq 2^{63}$ and $\sigma = 1$ otherwise: w is a *NaN* when $e = 2047$ and $t \neq 0$, and the *value* of the double is

$$\text{doubleValue}(w) = \begin{cases} \sigma t \, 2^{-1074} & \text{if } e = 0 \text{ (the zeros and the subnormals)}, \\ \sigma(2^{52} + t) \, 2^{e-1075} & \text{if } 1 \leq e \leq 2046, \\ \bot \text{ for } \sigma = -1, \ \top \text{ for } \sigma = 1 & \text{if } e = 2047 \end{cases},$$

so that both zeros have the value 0. The intervals of a list over the values *are search intervals* when each $[l, u]$ has $l \leq u$ and each upper end lies below the lower end of the next interval; a list of pairs of doubles, each given by its bits w, is *well formed* when no end is a NaN and the values of the ends are search intervals. A rational number r *lies in* a list D of intervals over the values when $l \leq r \leq u$ for some $[l, u] \in D$. A triple (x, y, z) lies *in the search region* of the intervals D (over the values) and the range from b_{low} to b_{high} when

$$z = 0 \ \vee \ (b_{\text{low}} \leq z \ \wedge \ (b_{\text{high}} < 1 \vee z \leq b_{\text{high}}) \ \wedge \ (D = [\,] \vee x/z \text{ lies in } D)) \ :$$

a point at infinity lies in every region, and x/z may lie anywhere when there are no intervals. The search region is the *whole box* of the height bound H when there are no intervals, $b_{\text{low}} \leq 1$, and $b_{\text{high}} < 1$ or $H \leq b_{\text{high}}$. The *largest denominator* of the search is b_{high} when $1 \leq b_{\text{high}} \leq H$, and H otherwise.

The statements take the search intervals of the options, a list of pairs of doubles of the model (Definition 5.1), by the values of their ends where they speak about the points and by their bits where they test well-formedness (Definition 2.11). The value of a NaN is junk, an infinity, and nothing depends on it: the statements that read the values assume a successful search, which

refuses a NaN end. `Region.lean` checks its definitions by `decide +kernel`, which adds no axiom, on a few doubles and on intervals that `ratpoints` accepts and refuses: $[-0.0, 0.0]$ and $[0.0, -0.0]$ are accepted, and $[-1, -0.0]$ followed by $[0.0, 1]$ is refused.

Remark 2.7 (The region compared with `ratpoints`)**.** The ends of the intervals are C doubles (`ratpoints_interval`, `ratpoints.h:359-359`). The C code does no floating-point arithmetic on them that could decide a point: it compares them, converts between `long`s and doubles, takes doubles apart and scales them by powers of 2 (Chapter 5), and computes with them only in the estimate of the work in the choice of the moduli (Chapter 6). So the specification can speak about their exact values.

Well-formedness is the test of `find_points_work_1` that refuses intervals as bad arguments (`find_points.c:4851-4854`): an interval without $l \le u$, or one that does not begin above the end of the interval before it. Every comparison with a NaN is false, so that a NaN end is refused too; infinite ends, degenerate intervals $[a, a]$ and any number of intervals are accepted. On doubles that are not NaNs the comparisons of the C language agree with the order of the values, with -0.0 equal to 0.0; so the region is the same for -0.0 and 0.0, for the specification as for the C code.

Without intervals the library searches $[-h, h]$ for the double h of H, which holds every x/z of the box (`find_points.c:4831-4841`). It takes $b_{\text{low}} < 1$ as 1 and a b_{high} outside $1, \dots, H$ as H (`find_points.c:4659-4661`), which the region states without clamping; it delivers the points at infinity whatever the region, and forbids the reversal exactly when the region is not the whole box (`find_points.c:4686-4687`; Chapter 5).

Definition 2.8 (Sturm's theorem)**.** The proposition that for every squarefree real polynomial p and all reals $a \le b$

$$\#\{t \in \mathbb{R} : a < t \le b,\ p(t) = 0\} + V_p(b) = V_p(a)\,,$$

where $V_p(t)$ is the number of sign changes (`List.signVariations`) of the values at t of the Sturm sequence of p and its derivative (`Polynomial.sturmSeq p (derivative p)`), and the number of roots is `Set.ncard`.

The Sturm step of `ratpoints` narrows the search intervals to those on which f may take a value ≥ 0 (Chapter 5); that it keeps every point rests on Sturm's theorem, which Mathlib does not have, though it has the Sturm sequence and the sign changes of a list. The form for $(a, b]$ is the one that the bisection of the step needs, as it evaluates the sequence at midpoints that may be roots of f. The three certified statements that need it (membership, the order and the coverage when the exact check is off) take the hypothesis $0 \le \text{sturm} \to \text{SturmTheorem}$, so that they hold outright with the Sturm step off. It is a hypothesis, not an axiom; a proof of the theorem would discharge it without changing a statement.

2.4 The order of `ratpoints`

The file `Order.lean` defines the list of the points in the order in which `ratpoints` finds them, in terms of the polynomial, and the order theorem (Theorem 3.5) says that a search with the exact check on returns this list, filtered by noY and the search region, assuming Sturm's theorem when the Sturm step is on. It uses the prime factors of a number (`Nat.primeFactorsList`, in increasing order with multiplicity).

Definition 2.9 (The points in the order of `ratpoints`)**.** For the switch r of the reversal and the coefficient list c of degree d, the polynomial that `ratpoints` searches and whether it is the reversed one, $\text{searchedPoly}(r, c)$, is the first of the following that applies:

(1) $((c_d, c_{d-1}, \dots, c_0), \text{true})$ if r is true, d is even and $c_0 = 0$;

(2) $((c_d, c_{d-1}, \dots, c_0), \text{true})$ if r is true, d is even, c_d is a square and c_0 is not a square;

(3) $((0, c_d, c_{d-1}, \dots, c_1), \text{true})$ if r is true, d is odd, $|c_d| \neq 1$, $c_0 = 0$ and $|c_1| = 1$;

(4) (c, false) otherwise.

A natural number n is *factored by trial division* when the product of its prime factors from 1021 on, with multiplicity, is below 1021^2, or none of its prime factors exceeds 1021. The squarefree divisors of n at most B *in the order of binary counting*: for the distinct prime factors $p_1 < p_2 < \cdots$ of n, the products of the sublists of $(p_1, p_2, \dots)$ in the order of `List.sublists`, $1, p_1, p_2, p_1p_2, p_3, p_1p_3, \dots$, with p_1 the lowest bit, those at most B. The *denominators of the search* of c up to B, for a bound m of the number of divisors: when d is odd, $|c_d| \neq 1$, $|c_d|$ is factored by trial division and has fewer than m squarefree divisors at most B, the numbers $ek^2 \leq B$, divisor e by divisor in the order of binary counting and $k = 1, 2, \dots$ for each; otherwise $1, \dots, B$. The *points with the denominator* z: for $x = -H, \dots, H$ in turn, with $y = \lfloor \sqrt{F_c(x, z)} \rfloor$ (0 when $F_c(x, z) < 0$), those among (x, y, z) and $(x, -y, z)$ that are normalized points of height at most H, in this order, one triple when $y = 0$; the denominator 0 gives the points at infinity. The *points in the order of* `ratpoints`, orderedPoints(m, r, c, H, B): for $(c', \text{rev}) = \text{searchedPoly}(r, c)$, the points of c' with the denominator 0 and then with the denominators of the search of c' up to B, in turn, each (a, y, b) mapped to (b, y, a) for $a \geq 0$ and to $(-b, y, -a)$ for $a < 0$ when rev is set.

The binary form of a reversed list is that of c with x and z exchanged, except for even d with $c_0 = c_1 = 0$, which `ratpoints` refuses; so the map at the end is a bijection from the points of c' to those of c that preserves the box (Chapter 4). In the order theorem, m is maxDivisors (Definition 2.12), r the switch of the reversal when the search region is the whole box and false otherwise, and B the largest denominator: the search intervals and the least denominator only filter the list.

Remark 2.10 (The order compared with `ratpoints`)**.** searchedPoly is the reversal check of `find_points_work_1` (`find_points.c:4732-4803`) on the coefficient list; items (1) to (3) are its cases 1 to 3. In case (1) the check refuses the form as not squarefree when also $c_1 = 0$, since then $X^2 \mid \Phi_c$ (`find_points.c:4744-4749`), and otherwise reverses $c_0, \dots, c_d$, which lowers the degree to $d - 1$; in case (2) it reverses $c_0, \dots, c_d$, in case (3) $c_1, \dots, c_d$. The squares are those of GMP's `mpz_perfect_square_p`, which agrees with Mathlib's `IsSquare` on $\mathbb{Z}$: no negative integer is a square. After a reversal the C code hands each point (a, y, b) of the polynomial searched to the caller mapped as above (`find_points.c:5312-5316`).

The C code delivers the points at infinity first, then sieves the denominators in the order of its loop, and for each denominator delivers the points by increasing numerator, (a, y, b) before $(a, -y, b)$ (`find_points.c:5300-5327`). For odd d the loop departs from $1, \dots, B$ in two ways. When $|c_d| = 1$ it visits only the squares, in increasing order; the specification lists $1, \dots, B$, and the denominators that are not squares carry no point. When $|c_d| \neq 1$, `setup_us1` factors $|c_d|$ by trial division up to 1021, which succeeds exactly when $|c_d|$ is factored by trial division in the sense above, and makes the squarefree divisors up to b_{high} in the order of binary counting (`find_points.c:777-860`, `find_points.c:864-881`); when there are fewer than `MAX_DIVISORS`, the loop visits ek^2, divisor by divisor and k increasing (`find_points.c:3937-3940`), the one case in which the denominators do not increase. Chapter 8 proves that the loop of the model visits the denominators in this order.

`Order.lean` checks its definitions by `decide +kernel` on outputs of the C program, among them `ratpoints '-1 0 9' 10 -q`, which reverses the polynomial to `9 0 -1`, and `ratpoints '-6 5 -3 30' 12 -q`, which visits the denominator 6 before 5, in the order $1, 2, 3, 6, 5, 10, 15, 30$ of the divisors of 30, and on runs with search regions.

2.5 The options, the refusals and the outcome of a search

The main theorems speak about the search findPoints(c, H, s, o), which returns the list of the points or a refusal, and about the search with a callback (Definition 3.1). Their arguments and results are types of the model, whose fields mirror the fields of the arguments `ratpoints_args` of the library (`ratpoints.h:362-400`) and the bits of its flags (`ratpoints.h:402-424`).

Definition 2.11 (The options, the refusals and the outcome of a search)**.** The *options* of a search are those of the library that change the refusals, the order of the output or what is delivered, with the defaults of a run of the program without options: whether the set-up may reverse the polynomial, that is, whether the bit `RATPOINTS_NO_REVERSE` is clear (true by default); the depth `args->sturm` of the Sturm step, an integer, negative for none, by default defaultSturm $= 10$ (`RATPOINTS_DEFAULT_STURM`); the two options of the output noCheck and noY, false by default, the bits `RATPOINTS_NO_CHECK`, under which every numerator that passes the sieve is delivered with $y = 0$ without the exact check, and `RATPOINTS_NO_Y`, under which only the point with $y \geq 0$ of a pair is delivered; and the search region: the search intervals, a list of pairs of doubles for the array `domain` with its length `num_inter`, empty by default, and the least and the largest denominator, integers for the `longs` `b_low` and `b_high`, by default 1 and 0 (`main.c:348-348`, `main.c:352-352`, `ratpoints.h:403-405`); the program sets `b_high` to H, and the library takes a value below 1 as H, so that the two defaults give the same search. The statements read the search intervals by the values of their ends, domainValues, and by their bits, domainBits.

The *options of the sieve* are the values of the options of the program that tune the sieve, which change neither the points nor their order: the number of primes examined (`-p`), the cut points of the stages (`-n`, `-N`), the parameters of the choice of the moduli and of its correction (`-r -R -U -C -A -P -Q -W`), the number of forbidden divisors (`-F`), the array size, and whether the Jacobi test runs (`-j`, `-J`), as the program reads them into `ratpoints_args`, -1 for an absent option that takes a number (`main.c:337-347`, `main.c:349-350`, `ratpoints.h:365-367`).

The *refusals* are five, each with the error of the program: more than maxDegree$+1$ coefficients (error 3), fewer than two (error 5), the height bound 0 or one of 2^{63} or more, which is not a C `long` (error 4), a binary form that is not squarefree (error 8), and search intervals that are not well formed (error 9).

The *outcome* of a search that is not refused is what the library gives back to its caller: the data of the callback, the count, the three bits of the flags that the program reads after the search, `RATPOINTS_REVERSED`, `RATPOINTS_USE_SQUARES` and `RATPOINTS_USE_SQUARES1`, the search intervals as the search leaves them, and the numbers of the moduli used in the stages (`ratpoints.h:377-383`, `ratpoints.h:414-414`, `ratpoints.h:416-417`).

The search refuses in the order of the program, which differs from that of the library on a few inputs; Section 4.1 gives this order with its lines of C, and Theorem 3.6 says exactly which inputs are refused, with the Sturm step on and off. The options of the sieve are not options in the sense above: the main theorems hold for every value of them, with no hypothesis, because the search chooses its moduli from the curve and these values only steer the choice (Chapter 6). The search normalizes both kinds of options as `find_points_work_1` does (`Ratpoints.Options.normalize`, `Ratpoints.SieveOptions.normalize`), and the statements speak about the options as the caller gives them, as `find_points_work` restores the fields that it normalized.

Definition 2.12 (The bounds that the statements mention)**.** maxDegree = 100, the largest degree of a polynomial that `ratpoints` accepts, the value of `RATPOINTS_MAX_DEGREE` (`ratpoints.h:33-33`): a list of coefficients has at most maxDegree + 1 = 101 entries. maxDivisors = 512, the value of `MAX_DIVISORS` (`find_points.c:121-121`): the loop over the denominators of a polynomial of odd degree runs over the divisors of its leading coefficient times squares only when there are fewer than this many divisors up to the largest denominator. An integer is a C `long` when it lies in $[-2^{63}, 2^{63})$ (`rp-private.h:41-43`): `ratpoints` needs a `long` of 64 bits, and the height bound is one.

Like the other constants that the C code fixes when it is compiled, the two are definitions of the model with the values of the default build, and no proof depends on their values beyond $0 < \text{maxDegree} < 2^{57}$; the author checks this by building the project with other values of the constants (Chapter 11).

2.6 The reference of the delivery

The library function `find_points` returns no list of points. For each point that it finds it calls a function `process` of the caller with the coordinates, the caller's data `info` and a pointer to a flag `quit`; it adds the value that the call returns to a count `total`, which it returns, and it stops the search as soon as a call sets the flag (`ratpoints.h:436-437`).

Definition 2.13 (The reference of the delivery)**.** The callback is a function of x, z, y and the data, in the order of the C code, that returns what the call gives: the weight that it returns, an integer, whether it set `*quit`, and the data after the call (`ratpoints.h:436-437`, `ratpoints.h:441-442`). A *delivery* is the data of the callback, the count, the sum of the weights so far, and the flag of the stop; one call on a delivery adds the weight to the count and takes the flag and the data that the call leaves. The *delivery of a list* L of points to the callback, from a delivery r: for each point (x, y, z) of L in turn, while the flag is clear, the call with (x, z, y); the points after a call that sets the flag, and all of them when it is set at the start, are passed over.

This fold over a list is a reference of the model, as the brute-force reference is, and models no C code: the search delivers each point where it finds it and tests the flag where the C code tests `quit` (Chapter 10). By the protocol theorem (Theorem 3.7) the search with any callback refuses what findPoints refuses and otherwise gives the data and the count of the delivery of its list from the caller's data, the count 0 and the flag clear; findPoints is itself the search with the callback that appends each point to an array, with the weight 1, and never stops (`Ratpoints.appendPoint`). The weights are integers without a bound, for the `int`s that the callback returns.

2.7 The brute-force reference

Definition 2.14 (The brute-force reference)**.** The list, computed in the model, of the triples found by testing, in this order, the candidate $(1, 0)$ if $H \geq 1$, and then the pairs (x, z) with $1 \leq z \leq H$, $-H \leq x \leq H$ and $\gcd(x, z) = 1$, with z increasing and, for each z, x increasing. For each candidate, $v = F_c(x, z)$ is computed by Horner's rule in x on the products $c_k z^{d-k}$, computed once for each z, times z when d is odd; if $v \geq 0$ and $v = y^2$ for $y = \lfloor \sqrt{v} \rfloor$, the triple (x, y, z) is listed, followed by $(x, -y, z)$ if $y \neq 0$. The part of the list over one denominator $z \geq 1$, or over the candidate $(1, 0)$ for $z = 0$, is a function of its own, with which the proofs compare a search denominator by denominator.

The reference is the core of the script `verify-test4.py` of the C repository, written in the model, in Lean core without Mathlib. It evaluates the form as the exact check of the C code does (Remark 2.2), so that the candidate $(1, 0)$ needs no case of its own, and it computes the degree by the expression of degree, which the model cannot import. It takes no hypothesis on c or H. The proofs of the main theorems compare the list of the search with this reference, denominator by denominator (Chapter 3); for a polynomial that `ratpoints` neither reverses nor searches in the order of the divisors, and for the whole box with the options of the output clear, its list is that of `ratpoints`, in the same order.

Theorem 2.15 (The brute-force reference finds exactly the points)**.** *For every list c and every H: a triple lies in* bruteForce(c, H) *if and only if it is a normalized point of height at most H on $y^2 = F_c(x, z)$; the list has no duplicates; and its set of elements is* points(c, H)*.*

Proof. Horner's rule on the products, times z for odd d, is $F_c(x, z)$; v is the square of y exactly when $v \geq 0$, $v = \lfloor\sqrt{v}\rfloor^2$ and $y = \pm\lfloor\sqrt{v}\rfloor$. The candidate $(1, 0)$ gives the points with $z = 0$, since $\gcd(x, 0) = 1$ and the normalization force $x = 1$, and the pairs give those with $z > 0$. The triples of different candidates differ in x or in z, and $y \neq -y$ for $y \neq 0$, so that there are no duplicates. The third statement follows from the first. □

Comparator certifies the first two statements. The third is not in the challenge, because points is not restated there: comparator compares the full declarations that a certified statement reaches, so that the finiteness proof inside points would have to be reproduced term for term, and the first two statements say everything that the third says.

Chapter 3

The main theorems

This chapter states what is proved about the search of the model, the counterpart of the library function `find_points`: which points it finds, without duplicates and in which order, which inputs it refuses, what it delivers to a callback and without the exact check, and what the program `ratpoints-model` prints. Comparator certifies nine of these statements, the two on the brute-force reference and the two on the fast search of Chapter 1, thirteen in all (Section 3.8). The proofs in outline (Section 3.6) reduce them to what Chapters 4 to 10 prove part by part: that the test of the sieve lets every point through and that the parts compose to the search.

Throughout, c is a coefficient list of degree d (Definition 2.1), H a natural number, s the options of the sieve and o the options of a search (Definition 2.11). The search intervals of o are read by the values of their ends (`Ratpoints.domainValues`), except where their well-formedness is tested, which reads their bits (`Ratpoints.domainBits`). A triple *lies in the search region of* o when it lies in the search region of these values and of the range of the denominators from o.bLow to o.bHigh (Definition 2.6). The hypothesis

$$h_S\colon \quad 0 \le o.\text{sturm} \;\to\; \text{SturmTheorem}$$

assumes Sturm's theorem (Definition 2.8) only when the Sturm step is on, so that a statement with h_S holds outright for a search without the Sturm step.

3.1 The search

The model has three entry points, one inside the next: the search with a callback and a print channel, which the program runs; the search with a callback, the library function as a caller without `RATPOINTS_VERBOSE` sees it; and findPoints, which collects the points in a list. The statements speak about the last two.

Definition 3.1 (The search for the points)**.** The search with a callback and a print channel models the library function `find_points`, the function `find_points_work` that it calls and the start of `find_points_work_1` (`ratpoints.h:436-437`, `find_points.c:4523-4528`, `find_points.c:4624-4633`, among others). It makes the refusals of the program in its order (Lemma 4.2), first those of the coefficient list, the height bound and the leading zeros (errors 3, 5, 4 and 8), then those of the set-up (Definition 4.1), which may also return at once after the Sturm step with nothing delivered; otherwise it runs the search of the polynomial that the set-up hands on (Definition 3.2). A refusal comes with the data of the callback as the report has left it; otherwise the search returns its outcome (Definition 2.11). The print channel prints the report of the library under

`-v` into the data of the callback (Definition 10.5). The search with a callback is this search with the data dropped from a refusal and, by default, the print channel that prints nothing; findPoints(c, H, s, o) is the list of the array that it gives for the callback that appends each point and never stops (`Ratpoints.appendPoint`), from the empty array, or its refusal.

Definition 3.2 (The search of the polynomial handed on)**.** For the polynomial that the set-up hands on, with the flags of the check (the flag of the reversal and the two options of the output), the range of the denominators and the intervals that the Sturm step leaves, with its flag `sturm_empty`, the search sets up the tests on the denominators (`find_points.c:4900-4914`) and the 2-adic information, and returns with nothing delivered when no class modulo 64 admits a numerator and there is no point at infinity (`find_points.c:4954-4959`); it tests whether the coefficients are all `long`s, (Definition 2.12; `find_points.c:4814-4823`), which chooses how the tables reduce them modulo a prime; it chooses the moduli (Definition 6.7), with nothing delivered when the curve has no points modulo a prime (`find_points.c:4996-5006`); it delivers the points at infinity (Definition 4.9) and returns when the callback has stopped the search or the Sturm step has left no interval (`find_points.c:5100-5103`, `find_points.c:5105-5106`); otherwise it runs over the blocks of the denominators of the shape of the loop (Definition 8.10) with the correction of the moduli (Definition 6.11, `find_points.c:5137-5139`). Each point is mapped back to the caller's curve when the flag of the reversal is set (Definition 4.4). The outcome has the bits of the flags, the search intervals that the Sturm step leaves (those that this search is given), and the numbers of the moduli of the last block that the run sifts (`find_points.c:5210-5213`), 0 after a return before the sieve.

3.2 The points that a search finds

Theorem 3.3 (Membership)**.** *Assume h_S and that the exact check is on,* o.noCheck = false*, and let* findPoints(c, H, s, o) *return the list L. A triple $p = (x, y, z)$ lies in L if and only if p is a normalized point of height at most H on $y^2 = F_c(x, z)$ (Definition 2.4), $y \geq 0$ if* o.noY *is set, and p lies in the search region of o. Indeed L is a permutation of the brute-force reference* bruteForce(c, H) *filtered by the last two conditions; and when neither option of the output is set, the set of the elements of L is* points(c, H) *filtered by the search region of o.*

Proof. By Lemma 3.12(3), L is what the brute-force reference of the polynomial searched finds at infinity and over the denominators of its search up to the largest, mapped back after a reversal and filtered by the two conditions. Mapped back, these are a permutation of the points of bruteForce(c, H) at infinity or with a denominator up to the largest (Lemmas 8.12 and 4.6), and the search region holds no point with a larger one; the reference finds exactly the points (Theorem 2.15). □

Theorem 3.4 (No duplicates)**.** *For all c, H, s and o, if* findPoints(c, H, s, o) *returns the list L, then L has no duplicates.*

Proof. By Lemma 3.12(1), L is the list of a search for a sound test. With the exact check on, Lemma 3.11(2) makes it a sublist of the brute-force reference of the polynomial searched, mapped back; the denominators of the search have no duplicates (Lemma 8.12), and the reference of a reversed polynomial, mapped back, is a permutation of that of c (Lemma 4.6), which has no duplicates (Theorem 2.15). With the exact check off, Lemma 3.11(3). □

No duplicates needs no hypothesis at all: neither Sturm's theorem nor the exact check.

Theorem 3.5 (The order of the points)**.** *Assume h_S and that the exact check is on,* o.noCheck = false. *Let* findPoints(c, H, s, o) *return the list L, let r be true when o allows the reversal and its search region is the whole box of H, and B the largest denominator of the search,* o.bHigh *if* $1 \le o.\text{bHigh} \le H$ *and H otherwise (Definition 2.6). Then L is*

$$\text{orderedPoints}(\text{maxDivisors}, r, c, H, B)$$

(Definition 2.9) with the points left out that have $y < 0$ when o.noY *is set or do not lie in the search region of o. Moreover, the search with the Sturm step off and the other options as in o returns L too.*

Proof. Lemma 3.12(3) gives L as in the proof of Theorem 3.3. The normalized options have the switch r and the largest denominator B (Lemma 5.4); the reversal check with r hands on searchedPoly(r, c) (Lemma 4.6); the denominators of its search from 1 to B are those of searchDenominators (Lemma 8.12); and over a denominator $1 \le z \le H$, or the candidate $(1, 0)$, the reference finds the points with the denominator z, or 0. For the last statement, the search with the step off refuses only where the search with the step on refuses too (Theorem 3.6), and otherwise both lists are given by the first statement, which does not mention the depth. □

So the list is the same for every value of the options of the sieve, which change the moduli and the work but not the points or their order; and with the exact check on, the depth of the Sturm step does not change it either. This last statement is proved, not certified.

3.3 The refusals

The refusals of the search are those of the program (Chapter 4). Two theorems say exactly which inputs they are, with the Sturm step on and off; so a search that refused every input would not meet the statements of this chapter.

Theorem 3.6 (The refusals)**.** *Let $n + 1$ be the length of c and $N = 2\lceil n/2 \rceil$. With the Sturm step on ($0 \le o.$*sturm*),* findPoints(c, H, s, o) *refuses if and only if*

(1) *c has fewer than two or more than* maxDegree $+ 1$ *entries,*

(2) *$H = 0$ or $H \ge 2^{63}$,*

(3) *the binary form Φ_c (Definition 2.3) is not squarefree, or*

(4) *the search intervals of o are not well formed (Definition 2.6).*

*With the Sturm step off ($o.$*sturm < 0*), it refuses if and only if (1), (2) or (4) holds, or $d+2 \le N$, or o allows the reversal, its search region is the whole box of H, d is even and $c_0 = c_1 = 0$.*

Proof. The search refuses in the order of Lemma 4.2, a natural number being a `long` exactly when it is below 2^{63}. The leading zeros lower $\lceil n/2 \rceil$ exactly when $d + 2 \le N$, and then $Z^2 \mid \Phi_c$; the reversal check refuses exactly in the case stated, and then $X^2 \mid \Phi_c$ (Lemmas 4.3 and 4.5); malformed search intervals are refused (Lemma 5.4); and the Sturm step, when it is on, refuses exactly when the remainder chain is none (Lemma 5.8), which after the tests before it means that Φ_c is not squarefree (Lemma 4.8). As each refusal of the form implies that Φ_c is not squarefree, the disjunctions come out as stated. □

Without the Sturm step the search refuses only some of the forms that are not squarefree, as `ratpoints` does under `-s`, and searches the others; the statements on the points hold for these searches too.

3.4 The delivery and the options of the output

The library delivers each point to the callback where it finds it and stops when a call sets the flag `*quit` (Chapter 10). By the protocol this comes to a fold of the list of findPoints (Definition 2.13), so that the statements on the list carry over to every callback.

Theorem 3.7 (The protocol of the delivery)**.** *For all c, H, s and o, every callback and its data: the search with the callback, with the print channel that prints nothing, refuses exactly when* findPoints(c, H, s, o) *refuses, with the same refusal; and when* findPoints(c, H, s, o) *returns the list L, the search with the callback returns an outcome whose data and count are those of the delivery of L to the callback from its data, the count* 0 *and the flag of the stop clear. So the calls of the callback are those for the points of L, in their order, up to and including the first call that stops the search, and the count is the sum of the weights that these calls return.*

Proof. Both searches make the same refusals, which do not depend on the callback, and otherwise search the same polynomial over the same range of the denominators and the intervals that the Sturm step leaves, or both return at once after the Sturm step (Lemma 4.2). The intervals are well formed and the range lies in $1, \dots, H$ (Lemmas 5.4 and 5.8), so that Lemma 3.11(1) gives the data and the count of the search with the callback as those of the delivery of the list of the search with the appending callback. The proof rests on the contract of the search, with one test of the sieve for every callback, not on an argument about the control flow. □

The protocol speaks of the data and the count only. The rest of the outcome is described in Section 10.4; the numbers of the moduli used depend on where the callback stops the search. The report under `-v`, which comes through the print channel, is compared with that of the C program and not proved (Chapter 11).

Under `RATPOINTS_NO_CHECK` the library delivers, each with $y = 0$ and mapped back after a reversal, the candidate at infinity of the polynomial searched when that polynomial has points at infinity (odd degree or a square leading coefficient), and every numerator that passes the sieve, the test for common factors and the third stage, whether or not it gives a point. Which candidates pass depends on the moduli and is not specified; the theorem below says that each is a triple $(x, 0, z)$ with (x, z) normalized, of height at most H and in the search region, and that they include the points, in the order in which the search with the exact check finds them.

Theorem 3.8 (The output without the exact check)**.** *Let the exact check be off,* o.noCheck = true*, and let the search* findPoints(c, H, s, o) *return the list L.*

(1) *Every element (x, y, z) of L has $y = 0$ and the pair (x, z) normalized and of height at most H (Definition 2.4), and lies in the search region of o.*

(2) *Assume h_S. For every normalized point (x, y, z) of height at most H in the search region of o, the triple $(x, 0, z)$ lies in L.*

(3) *If the search with the same s and o with* noCheck *cleared and* noY *set returns the list L', then L' with each y replaced by* 0 *is a sublist of L.*

Proof. By Lemma 3.12(1), L and L' are the lists of searches with the same sound test over the same region, which differ only in the flags of the check. Item (1): the pair by Lemma 3.11(3); the region, without a reversal, by Lemma 3.11(4), since the region that the search uses lies in that of o (Definition 5.11, Lemma 5.4), and after a reversal, which comes only for the whole box (Lemma 4.5), since every normalized pair of height at most H lies in it. Item (3) is Lemma 3.11(5). Item (2): $(x, |y|, z)$ lies in L' by Theorem 3.3, and item (3) puts $(x, 0, z)$ into L. □

Items (1) and (3) do not need Sturm's theorem. The two searches of item (3) choose the same moduli and make the same corrections, which depend neither on the callback nor on the options of the output (Chapter 6).

3.5 What the program prints

The program `ratpoints-model` runs the search with the callback of `main.c`, which prints the points (Chapter 10). For a command line that it reads as c, H, s, o and the settings of the output, it searches with the options s' of s with the array size that `main.c` sets and the options o' of o with the search intervals of the command line joined (Definition 10.7).

Theorem 3.9 (What the program prints)**.** *Let the program read a command line as above, and call the points of a list that the callback of the program keeps (Definition 10.9), under* `-l` *only the first of them, its* counted points *(`Ratpoints.Cli.countedPoints`).*

(1) *Without* `-v`*: if* findPoints(c, H, s', o') *returns the list L and the search with the callback of the program the outcome r, the program prints the head of its report, the string before the points, the counted points of L (none under* `-z`*) in the format, separated by the string between the points, the string after the points and the end of the report for r, with the status* 0*; the count of r is the number of the counted points of L.*

(2) *Under* `-q`*, which clears* `-v`*, the program prints, whenever* findPoints(c, H, s', o') *returns L, the same without the head and the end of the report; under* `-q -l` *without* `-z` *the one point printed is the first point of L that the callback keeps.*

(3) *The callback keeps a point unless it lies at infinity under* `-i`*, or $z \neq 0$, the command line gives search intervals and x/z does not lie in their exact intervals. When every argument of the command line has fewer than 2^{56} characters, the joined intervals are well formed (Definition 2.6), so that the library does not refuse them.*

(4) *Assume h_S and that the exact check is on. If* findPoints(c, H, s', o') *returns L, a triple lies in L and is kept by the callback exactly when it is a normalized point (x, y, z) of height at most H (Definition 2.4) with $y \geq 0$ under* `-y` *and $z \neq 0$ under* `-i`*, and, for $z \neq 0$, with z in the range of the denominators of* `-dl` *and* `-du` *and x/z in the exact intervals of the command line when there are such intervals.*

Proof. The delivery of any list to the callback of the program prints its counted points in turn and counts them, stopping under `-l` at the first point kept (`Ratpoints.Cli.deliver_process`); by the protocol (Theorem 3.7) the search with the callback gives the data and the count of the delivery of L; without `-v`, which `-q` clears (`Ratpoints.Cli.verbose_eq_false_of_input_eq_ok`), the print channel prints nothing. Items (3) and (4) rest on an invariant of the reading of the intervals (`Ratpoints.Cli.Intervals.ReadsWF`): the exact ends are those of increasing and disjoint intervals, and the doubles of each interval hold it for $|x| \leq H$. For arguments of fewer than 2^{56} characters (`scan_end` stops accumulating the digits of an exponent at `LONG_MAX/20`) the doubles of the lower ends after the first are the largest doubles at most their exact ends (Lemma 10.8), so that the join makes intervals that are increasing and strictly apart. Item (4) is then membership (Theorem 3.3) on the joined intervals with the test of the callback. □

3.6 The search as a filter of the reference

The theorems come from one description of what a search delivers, in terms of a test of the sieve of which the proofs use two properties: it lets the numerator of every point through, and it depends neither on the callback nor on the options of the output. In the search it is the 2-adic word of the class of the denominator (Chapter 8), the rows of the moduli of its block (Chapters 6 and 7) and the test for common factors (Chapter 9); the exact check then decides, so that the list of a search is a filter of the brute-force reference, up to the order and the region.

Definition 3.10 (The list of a search)**.** For the coefficient list c of its degree, H, the denominators from b_{low} to b_{high}, intervals D of values, a test $S(b, a)$ of the numerators a over the denominators b, and the flags of the check: the points at infinity with the flags (Definition 4.9), followed, for the denominators b of the search from b_{low} to b_{high} in the order of its loop (Definition 8.10), by the points with the flags (Definition 10.3) of the numerators $a = -H, \ldots, H$ with a/b in one of the intervals of D and $S(b, a)$. The test is *sound* when, over each denominator b of the search, it holds for every numerator $a \in [-H, H]$ over which the brute-force reference finds a point (a, y, b).

Lemma 3.11 (The list of a sound test)**.** *For the list of a search of c with the denominators from b_{low} to b_{high}, intervals D, a test S and the flags of the check:*

(1) *Let $1 \le H < 2^{63}$, $1 \le b_{\mathrm{low}}$, $b_{\mathrm{high}} \le H$, and let the search intervals be well formed. For the normalized options of the sieve and either value of* `sturm_empty` *there is a sound test S such that the search of c of its degree (Definition 3.2), with the print channel that prints nothing, gives, for every callback, every setting of the flags of the check and every delivery whose flag of the stop is clear, the data and the count of the delivery of the list of the search for S, the values of the search intervals (none when* `sturm_empty` *is set) and these flags. So the search with any callback gives the data and the count of the delivery of the list that the search with the appending callback gives. The run over the blocks has the corresponding contract, for sound arguments (*`Ratpoints.Sieve.IsSoundRun`*), a well-formed first decision of the choice and the bounds of well-formed search intervals.*

(2) *If S is sound, $d \ge 1$ and the exact check is on, the list is what the brute-force reference finds over the candidate $(1, 0)$, followed by what it finds over the denominators of the search and the numerators $a \in [-H, H]$ with a/b in D, each point mapped back when the flag of the reversal is set, without those with $y < 0$ under* noY*.*

(3) *With the exact check off, $1 \le H$, $1 \le b_{\mathrm{low}}$ and $b_{\mathrm{high}} \le H$, whatever D and S, every element is $(x, 0, z)$ with (x, z) normalized and of height at most H, and the list has no duplicates.*

(4) *Without the flag of the reversal and for $1 \le b_{\mathrm{low}}$, every element has $z = 0$, or $b_{\mathrm{low}} \le z \le b_{\mathrm{high}}$ and x/z in D.*

(5) *For the same D, S and flag of the reversal, the list with the exact check on and* noY *set, each y replaced by 0, is a sublist of the list with the exact check off, for either value of* noY*.*

Proof. (1): by cases on the 2-adic information and the result of the choice of the moduli, which depend neither on the callback nor on the flags; the test is chosen after them. At either exit the curve has no points (Lemmas 8.2, 6.8 and 4.10), and the test that lets nothing through is sound. Otherwise the result of the choice is what the run needs (Lemma 6.8), and after the points at infinity the run delivers, by the block lemma (Theorem 3.13) and the contract of one denominator (Definition 9.11), the points with the flags of the numerators in the ranges of the intervals (Lemma 5.6) that pass the class word of b and survive the entries of its block. This test

is sound, since the shape lets the denominator of every point through in the order of the search (Lemmas 8.11 and 8.12), the class word admits the numerator of every point (Lemma 8.2), and every entry is sound (Lemma 7.10). (2): a sound test leaves out only numerators over which the reference finds nothing; (2) and (5) then hold numerator by numerator and at infinity (Lemma 10.4). (3): the survivors are $(a, 0, b)$ with $\gcd(a, b) = 1$, $|a| \le H$ and b in the range, distinct over distinct denominators (Lemma 8.12), and the map of the reversal is injective on normalized pairs (Lemma 4.6). (4) is read off the definition. □

Lemma 3.12 (The list of a search for the points)**.** *Let* findPoints(c, H, s, o) *return the list* L*, let* c' *and* rev *be the list and the flag that the reversal check of the normalized options hands on, and let the normalized range of the denominators run from* b'_{low} *to* b'_{high}*.*

(1) *The predicate* `Ratpoints.IsSearchedPoly` *of what a successful search guarantees after its refusals (Lemma 4.2) holds for* c' *and* rev*, and there is a sound test* S *over that range such that* L *is the list of the search of* c' *for* S *and the region that the search uses (Definition 5.11), with the flag* rev *and the options of the output of* o*, or the empty list when that region is none; for every choice of the two options of the output, with the other options unchanged,* findPoints *returns the list for the same* S *and region with that choice.*

(2) *With the exact check on,* L *is what the brute-force reference of* c' *finds over the candidate* $(1, 0)$ *and over the denominators of its search from* b'_{low} *to* b'_{high} *with the numerators in the region that the search uses, mapped back after a reversal, without the points with* $y < 0$ *under* noY*.*

(3) *If moreover* h_S *holds,* L *is what the reference of* c' *finds over the candidate* $(1, 0)$ *and over the denominators of its search from* 1 *to* b'_{high} *with all the numerators* $-H, \dots, H$*, mapped back after a reversal, without the points with* $y < 0$ *under* noY *and those outside the search region of* o*.*

Proof. (1): after the refusals the search of c' runs over the normalized range, which lies in $1, \dots, H$, and over well-formed intervals (Lemmas 4.2, 5.4 and 5.8), so that Lemma 3.11(1) applies with the appending callback; the options of the output enter only the flags of the check, which the test precedes. (2) follows by Lemma 3.11(2). (3): under h_S, the region that the search uses holds, among the numerators of the points of a denominator, exactly those with a/b in the caller's intervals, and when the step returns at once the polynomial searched has no point at infinity (Definition 5.11); the denominators from b'_{low} on are those from 1 that are at least b'_{low} (Lemma 8.12); and a reversal comes only for the whole box (Lemma 4.5). □

3.7 The blocks, the lag and the threads

The run sifts the denominators in blocks, each with the moduli of the decision of the correction in force for it, and the correction may append a decision after each block (Chapter 6). A decision is the first block for which it holds, with the ends s_2 and s_3 of the second and third stages from that block on; the decision in force for block i is the last whose first block is at most i (Definition 6.9). The block lemma is a step of the proofs above (Lemma 3.11). The lag lemma and the thread lemma are not used by them: they state the property on which the pool of threads of the C code relies, so that the output of the model, which sifts the blocks in turn, is also that of `ratpoints` with threads. None of the three is a statement of the challenge.

Theorem 3.13 (The block lemma)**.** *Let the run satisfy the hypotheses of Lemma 6.10 other than those on block* i *and its decision, the sieve entries having what* `Ratpoints.Sieve.IsRunEntries`

asks (their tables of inverses, the reciprocals of their moduli in their order, the row offsets of the classes with a pattern for all of them, and prime entries where they count no factor; they need not be sound), let the word of the classes mark those with a nonzero pattern, and let the decision of the choice be well formed. Then the run delivers the points of its blocks in order, block i those of its denominators that pass the tests of the shape, each with the first s_3 entries of the decision of block i, which depends neither on the callback nor on the flags of the output. So, when the loop of the shape visits no denominator twice, there is for each denominator b a number $s_3(b)$, independent of the callback and the flags, such that the run delivers, over the denominators of the loop that pass its tests and in its order, the points with the flags of the numerators in the ranges of b that pass the class word of b and survive the first $s_3(b)$ entries.

Proof. The invariant of the loop over the blocks (`Ratpoints.Sieve.SieveRunLoopInvariant`): the delivery is that of the points of the blocks done, and until a stop the state of the correction is its closed form after their folds and the store fits all the entries. Each block delivers its points by Lemma 6.10, with a well-formed decision by Lemma 6.12; a stop in a block ends the run before any decision that its fold appends is read, and the delivery after a stop is that of the whole list (Lemma 10.2). The form per denominator follows, as the blocks, concatenated, visit the denominators of the loop in its order (Lemma 6.10). □

Theorem 3.14 (The lag lemma)**.** *For the arguments of a run and every block i, the decision in force for block i among the decisions after the folds of the first k blocks is the decision of block i for every $k \geq i - \mathrm{adaptLag}$: block i reads only decisions made at the folds of blocks at least $\mathrm{adaptLag} + 1$ before it.*

Proof. The decision in force for a block, the last whose first block is at most it, is unique, and decisions appended for later blocks change neither it nor its index. After k folds the decisions are those after any $m \leq k$ folds followed by decisions for the blocks from $m + \mathrm{adaptLag} + 1$ on (Lemma 6.12); take $m = i - \mathrm{adaptLag}$. □

Theorem 3.15 (The thread lemma)**.** *Under the hypotheses of Theorem 3.13, let each block i be sifted alone with the appending callback into a slot, from the empty array, with the decisions after the folds of any $k_i \geq i - \mathrm{adaptLag}$ blocks and a store of its own that fits all the entries. Then delivering the slots in the order of the blocks to a callback whose flag is clear, with a return after the slot in which a call stops the search, gives what the run delivers.*

Proof. By the lag lemma (Theorem 3.14) the decision of each slot is that of its block, and the points of a block do not depend on the store that it starts from as long as the store fits (Lemma 6.10); so slot i holds the points of block i, and the delivery of the slots in order is the delivery of their concatenation (Lemma 10.2), which is what the run delivers (Theorem 3.13). □

The pool of threads of the C code meets these hypotheses: a thread starts block i only once the blocks before block $i - L$ are folded, L being `RP_ADAPT_LAG` (`find_points.c:4292-4296`); the tables of rows that the threads share only gain complete rows, published once built (`init.c:180-180`, `init.c:241-241`, `init.c:391-391`, `init.c:443-443`), so that they fit all the entries whenever a thread reads them (`find_points.c:3160-3160`, `find_points.c:3533-3535`); and the calling thread delivers the slots in the order of the blocks. The threads themselves stay outside the model (Chapter 11), and `ratpoints-model` accepts `-t` and ignores it.

3.8 What comparator certifies

Comparator, a checker for Lean proofs built into Lake, takes a challenge, a file that states theorems with `sorry`, and a solution, a file that proves them, and checks the solution independently of

the build of the repository. The files are in the folder `comparator`: the challenge `Challenge.lean`, the solution `Solution.lean`, and `challenges.json`, which lists the thirteen statements and the permitted axioms; `scripts/run-comparator.sh` runs the check.

The challenge may import only Mathlib, the model and the fast search, which the layout check enforces. It restates verbatim the definitions of the four files of the specification with their decidability instances (Chapter 2), all but the finite set points(c, H) (Section 2.7), and imports exactly the Mathlib modules among the public imports of these files, so that the restated definitions elaborate to the same terms as there. It imports the whole model (`Ratpoints.Model`), of which the statements mention the search with and without a callback and the brute-force reference; the options, the options of the sieve and the doubles of the search intervals; the outcome, the answer of a callback, the delivery and its reference; and the constants maxDegree and maxDivisors. It imports the fast search (`Ratpoints.Fast`) too, of which the statements mention `Ratpoints.FastMask.fastPoints` and its parameters `Ratpoints.Fast.Tuning`. A definition of the model or of the fast search elaborates without Mathlib, to other instance terms, and is never restated: they count as the code that the executables run, while the specification is reproduced, so that a library that specified something else would be rejected.

Eleven of the statements are statements of this blueprint, each named as its theorem in the library with `challenge_` in place of `Ratpoints.` (the challenge spells out the maps of the search intervals to their values and bits, which the library abbreviates): the two statements of the brute-force reference (Theorem 2.15), the first statements of membership and of the order (Theorems 3.3 and 3.5), no duplicates (Theorem 3.4), the two refusal theorems (Theorem 3.6), the three statements without the exact check (Theorem 3.8) and the first statement of the protocol (Theorem 3.7). They hold for every coefficient list, height bound, options of a search and options of the sieve, so for every choice of the moduli and every correction that the search makes; Sturm's theorem enters as the hypothesis h_S of membership, the order and item (2) without the exact check. The other two are the theorems of the fast search, `Ratpoints.FastMask.mem_fastPoints_iff`, which assumes Sturm's theorem too, and `Ratpoints.FastMask.nodup_fastPoints` (Chapter 1). The solution proves each statement by its theorem in the library.

`lake comparator` builds and exports both files in a sandbox, without loading the `.olean` files of the project into its own process, and accepts the solution when each of its theorems states the same as the theorem of the challenge, comparing the full bodies, not only the types, of every definition that the statement reaches; when the proofs use only the permitted axioms, `propext`, `Quot.sound` and `Classical.choice`, so that a proof by `native_decide` or `bv_decide` would fail (the layout check forbids both in the project); and when the kernel accepts them. With `--paranoid` it checks the solution with six kernels: Lean's kernel in its default and in its paranoid mode, `lean4lean`, `nanoda`, `con-leche` and `con-ron` (the flag `enable_nanoda` of `challenges.json` does not switch `nanoda` off under `--paranoid`). On 2026-10-31, `lake comparator --paranoid` accepted the thirteen statements with all six.

What one has to read, then, is the challenge: the specification as restated there and, of the model, the types that the statements mention and the reference of the delivery, a fold of a few lines. Comparator does not certify that the executable `ratpoints-model`, which imports only the model and is compiled by the compiler of Lean, runs the model: the layout check keeps `partial`, `unsafe`, `implemented_by`, `extern` and experimental options out of the model's own code (Section 11.6), while the primitives of Lean core that the compiler implements natively, on machine words, arrays, thunks and `Float`, remain trusted (Section 1.6); no proof reads the value of a `Float` (Remark 6.6). Nor does comparator say anything about the C code (Chapter 11), or about the theorems outside the challenge (the forms of membership with the permutation and the set of the points, the depth of the Sturm step, the program theorem, the block, lag and thread lemmas): they are proved in the same library, but one has to read their Lean statements

to know what they say.

Chapter 4

The set-up

This chapter describes what a search does with its input before the sieve: it checks the coefficient list and the height bound, drops the leading zero coefficients, refuses a binary form that is not squarefree, may reverse the polynomial, delivers the points at infinity, and ends when the curve has no points modulo one of the primes it examines. The refusal theorems (Theorem 3.6) rest on it, and so do the statements on the points delivered, since after a reversal `ratpoints` searches another polynomial and maps its points back. The rest of the Sturm step, which begins with the remainder chain of this chapter, is in Chapter 5, the tests on the denominators that follow the set-up in Chapter 8, and the loop over the primes in which the exit fires in Chapter 6.

Throughout, $c = [c_0, \dots, c_n]$ is the coefficient list, constant term first; d is its degree, the largest index with $c_d \neq 0$; $D = 2\lceil d/2 \rceil$ and $N = 2\lceil n/2 \rceil$ are d and n rounded up to even numbers; $F_c(x, z) = \sum_{i \le d} c_i x^i z^{D-i}$ is the binary form of the points (Definition 2.1), $\Phi_c = \sum_{i \le n} c_i X^i Z^{N-i}$ the binary form of the refusals (Definition 2.3), and $f = \sum_i c_i X^i \in \mathbb{Q}[X]$ the polynomial that the C code works with.

4.1 The order of the checks

The program refuses more than maxDegree + 1 coefficients with its error 3, fewer than two with error 5, and a height bound below 1 or one that is not a `long` with error 4, before it calls the library (`main.c:362-380`). The library refuses a degree above maxDegree (`find_points.c:4557-4558`), and then:

(1) it drops the leading zero coefficients and refuses the form as not squarefree when that lowers $\lceil n/2 \rceil$, that is when $d + 2 \le N$, so that $Z^2 \mid \Phi_c$ (`find_points.c:4624-4632`); this catches every list of two or more entries of degree 0, so that the refusal of a degree 0 as bad arguments that follows (`find_points.c:4633-4633`) is never met from the program; then it refuses a height bound below 1 (`find_points.c:4658-4658`);

(2) it turns the reversal off when the caller gives search intervals or a range of the denominators other than $1, \dots, H$ (`find_points.c:4686-4687`), and runs the reversal check, which refuses the form when d is even and $c_0 = c_1 = 0$, so that $X^2 \mid \Phi_c$, and may reverse the polynomial (`find_points.c:4732-4803`);

(3) it takes the default interval $[-H, H]$, its ends rounded outward to doubles, when the caller gives none, and otherwise refuses as bad arguments search intervals without $l \le u$, not

increasing and strictly apart, or with a NaN end (`find_points.c:4831-4855`), which no command line causes;

(4) with the Sturm step on, it refuses the form when the remainder chain of the polynomial searched meets a zero remainder, then restricts the search intervals, and returns when nothing is left to search and there is no point at infinity (`find_points.c:4857-4892`);

(5) it sets up the tests on the denominators and chooses the moduli, each with an exit when it finds that the curve has no points (`find_points.c:4954-4963`, `find_points.c:4996-5012`);

(6) it delivers the points at infinity (`find_points.c:5068-5105`) and sieves.

The program reports the refusal as not squarefree with its error 8, and the other negative codes of the library with its error 9, but for a lack of memory, error 10 (`main.c:256-261`). All refusals come before the first call of the callback. Where the program and the library differ, the model follows the program: it refuses the coefficient list and the height bound before the leading zeros (so that `5 0` with $H = 0$ is error 4, as in the program), and a height bound of 2^{63} or more, which the library cannot be given, with error 4. These first checks are in the search with a callback (Definition 3.1); the rest of the set-up is one definition. Appendix A lists how else the components differ from the C code; the main deviations of the set-up are noted in the sections below.

Definition 4.1 (The set-up of a search)**.** The refusals (Definition 2.11) carry the errors 3, 5, 4, 8 and 9 of the program; an integer is a `long` when it lies in $[-2^{63}, 2^{63})$. The set-up of `find_points_work_1`, whose state is declared at `find_points.c:4577-4585`, for the list c of degree d and the options o, normalizes the options (`Ratpoints.Options.normalize`), so that the reversal is allowed only when o allows it and its search region is the whole box; runs the reversal check (Definition 4.4), which may refuse or hand on a list c' of degree d'; takes the search intervals of o, or the default one, refusing malformed ones (`Ratpoints.searchIntervals`); runs the Sturm step on c' (`Ratpoints.sturmRegion`), which with the step on refuses the form when the remainder chain of c' is none, returns at once with nothing delivered when it keeps no interval and c' has no point at infinity, and otherwise narrows the intervals; and runs the search of c' (`Ratpoints.searchPoints`) with the flag of the reversal and the options of the output as the flags of the check. Whether the coefficients of c' are all `long`s, which the C code tests in the set-up and the model tests in the search of c', where it is used (Definition 3.2), chooses how the tables reduce them modulo a prime: by the helper `mod` of the C code (`Ratpoints.Sieve.modLong`) or by the Euclidean remainder, which agree (Chapter 7).

Lemma 4.2 (The set-up in the order of the program)**.** *With the print channel that prints nothing, the search with a callback refuses, in this order, more than* maxDegree $+ 1$ *coefficients, fewer than two, $H = 0$ or $H \geq 2^{63}$, $\lceil d/2 \rceil < \lceil n/2 \rceil$ (error 8), the refusal of the reversal check with the switch of the normalized options, malformed search intervals, and the refusal of the Sturm step; otherwise it returns the outcome of the search of the list c' that the check hands on, or, when the Sturm step returns before the search, the data of the callback with the count* 0*, the flag of the reversal and no interval. The refusals do not depend on the callback. A natural number is a* `long` *exactly when it is below 2^{63}. The predicate* IsSearchedPoly *collects what the refusals leave for a successful search, from which the main theorems start (Chapter 3): the reversal check of the normalized options hands on c' with its degree d'; $1 \leq d' \leq$* maxDegree*; $1 \leq H < 2^{63}$; the caller's search intervals are well formed (Definition 2.6); and with the Sturm step on, the remainder chain of c' is not none.*

Proof. Unfolding the definitions gives the order; a print channel that prints nothing leaves the data as they are, and every refusing branch returns the same error for any callback. A natural number is never below -2^{63}. After the refusals c has at least two entries and the leading zeros did not lower $\lceil n/2 \rceil$, so $d \geq 1$; the reversal check returns the degree of its list, never raises it and keeps it at least 1 (Lemma 4.5). □

4.2 The squarefree binary form

As $\Phi_c = Z^{N-d}\,\Phi^0$, where Φ^0 is the homogenization of f in degree d, which Z does not divide, Φ_c is squarefree exactly when $N \leq d+1$ and f is squarefree (Lemma 4.3): the first condition is the test of the leading zeros, the second is what the remainder chain tests (Section 4.4). Mathlib has nothing on the squarefreeness of a homogenization; the facts that are needed are proved at Mathlib's generality in `Ratpoints/Mathlib`. The search itself needs no squarefreeness: the statements on the points hold for every polynomial that the set-up hands on, also for a form that is not squarefree, which the search accepts when the Sturm step is off.

Lemma 4.3 (Squarefreeness in the terms of the C code)**.**

(1) *Over a domain and for $m \geq \deg p$, the homogenization of degree m of a polynomial p is squarefree if and only if p is squarefree and $m \leq \deg p + 1$. The homogenization of degree m of the reflection of p in degree m is that of p with the variables exchanged; so, for $m \geq \deg p$, the homogenization of degree m of p is squarefree exactly when the reflection of p in degree m is, provided that m exceeds the degree of the reflection by at most one.*

(2) *The polynomial f of the list c has the degree d of c, and Φ_c is its homogenization of degree N; so Φ_c is squarefree if and only if f is squarefree and $N \leq d+1$. In particular Φ_c is not squarefree when the leading zeros lower $\lceil n/2 \rceil$, nor when $c_0 = c_1 = 0$, whatever the degree.*

Proof. (1) The homogenization of degree m is that of degree $\deg p$ times $Z^{m-\deg p}$. If $m \geq \deg p + 2$, then Z^2 divides it, and if $a^2 \mid p$, the square of the homogenization of a divides it. Conversely, if A^2 divides it and $m \leq \deg p + 1$, then A is homogeneous, as a factor of a homogeneous polynomial over a semiring without zero divisors (`MvPolynomial.IsHomogeneous.of_mul_left`), $Z \nmid A$, and the square of $A(X,1)$ divides p, so $A(X,1)$ is a constant, and then so is A. The reflection compares coefficients, and the exchange of the variables is a ring automorphism, which keeps squarefreeness. (2) The coefficient of X^i in f is c_i, so f has the degree $d \leq N$; Φ_c and the homogenization agree term by term, and (1) applies. A drop of $\lceil n/2 \rceil$ means $d+2 \leq N$, and $c_0 = c_1 = 0$ gives $X^2 \mid f$. □

4.3 The reversal

When the caller allows it and the search region is the whole box, `ratpoints` may replace the polynomial by its reversal, in three cases in which it expects the search of the reversed polynomial to be faster. The reversal exchanges x and z in the binary form; the C code maps each point of the reversed curve back to the caller's curve before it delivers it, so that the output holds the same points in another order, which the specification of the order follows: it names the polynomial searched, as the reversal check chooses it (Definition 2.9). The model makes a new list where the C code reverses its array in place, and writes back to the caller the flag of the reversal, not the reversed coefficients.

Definition 4.4 (The reversal check and the map of its points)**.** For the switch of the reversal and the list c of degree d: if the switch is off, c itself; otherwise, for even d with $c_0 = 0$, the refusal (not squarefree) if also $c_1 = 0$, and else the list $c_d, \dots, c_1, c_0$ with the degree $d - 1$ (case 1); for even d with c_d a square and c_0 not, the list $c_d, \dots, c_0$ (case 2); for odd d with $|c_d| \neq 1$, $c_0 = 0$ and $|c_1| = 1$, the list $c_0, c_d, \dots, c_1$ (case 3); and otherwise c itself; with a flag that says whether the list is reversed (`find_points.c:4744-4756`, `find_points.c:4759-4770`, `find_points.c:4775-4790`). The squares are those of `mpz_perfect_square_p`: 0 and the squares of the positive integers. The map of a reversed search sends a point (a, y, b) of the reversed curve to (b, y, a) for $a \geq 0$ and to $(-b, y, -a)$ for $a < 0$; the exact check applies it to every point it finds when the flag of the reversal is set (Chapter 9), and the points at infinity use its image of the candidate $(1, 0)$ (Definition 4.9).

Lemma 4.5 (The reversal exchanges x and z)**.** *The check refuses exactly when its switch is on, d is even and $c_0 = c_1 = 0$; with the switch of the normalized options, exactly when the options allow the reversal, their search region is the whole box (Definition 2.6), d is even and $c_0 = c_1 = 0$; and it reverses only for the whole box. For d the degree of c: the degree returned is that of the list returned, at most d, and at least 1 when $d \geq 1$. If the check reverses, the list c' it returns is the reflection of c in degree D, $c'_i = c_{D-i}$ for $i \leq D$ and $c'_i = 0$ beyond, and $F_{c'}(x, z) = F_c(z, x)$ for all integers x and z. If the leading zeros of c do not lower $\lceil n/2 \rceil$, then Φ_c is squarefree if and only if the polynomial of the list handed on is, reversed or not.*

Proof. By the cases of the check. In each case of a reversal the list built is the reflection, with $c_{d+1} = 0$ in case 3; the new leading coefficient, c_1, c_0 or c_1, is not zero, so the reversed list has the degree returned and the same D, and $F_{c'}(x, z) = \sum_i c_{D-i}\, x^i z^{D-i} = F_c(z, x)$. Without a reversal Lemma 4.3 gives the last claim; after one, the polynomial of c' is the reflection of f in degree $D = N$, of degree $D - 1$ or D, and the reflection does not change whether the homogenization of degree D is squarefree (Lemma 4.3). □

Lemma 4.6 (The points of a reversed search)**.** *The map of a reversed search keeps the normalized pairs (x, z) of height at most H (Definition 2.4) and is injective on the triples with $z > 0$ or $(x, z) = (1, 0)$. For lists c and c' with $F_{c'}(x, z) = F_c(z, x)$ for all x and z, it sends the normalized points of height at most H of c' to points of c, and the brute-force reference of c', mapped, is a permutation of that of c. The reversal check hands on the polynomial and the flag of the specification of the order (Definition 2.9). The negation of an integer of absolute value below 2^{63} is a `long`, so the map stays among the `long`s for $H < 2^{63}$.*

Proof. $F_c(-b, -a) = F_c(b, a)$, since D is even. For $a \neq 0$ the image has $z = |a| > 0$ and $|x| = b$; for $a = 0$ the normalization forces $b = 1$, and the image $(1, y, 0)$ is a point at infinity; so the box goes onto the box. On the triples with $z > 0$ or $(x, z) = (1, 0)$ the map is its own inverse; both references list their points once (Theorem 2.15), so the mapped list is a permutation. □

4.4 The remainder chain

Definition 4.7 (The remainder chain)**.** For the coefficients $c_0, \dots, c_d$ of f, with $d \geq 1$ and $c_d \neq 0$, the rows of integers that `sturm.c` computes (`sturm.c:99-163`): $s_0 = f$, $s_1 = f'$, and for $k = 2, \dots, d$ the row s_k from the remainder t of s_{k-2} by s_{k-1}: starting from $t = s_{k-2}$, while $\deg t \geq \deg s_{k-1}$, t becomes $ut - vX^{\deg t - \deg s_{k-1}} s_{k-1}$, with u and v the leading coefficients of s_{k-1} and t divided by their gcd and signed so that $u > 0$, and its zero leading coefficients are dropped; the result is none when t becomes zero. Otherwise s_k is $-t$ divided by the gcd of its coefficients, and the chain ends when s_k is constant. The contract, proved with the program logic of Lean core: the loops end, and the result is a chain (IsChain: rows as just described, each with a nonzero leading

coefficient, their sizes decreasing from s_1 on) whose last row is constant, or none, and then some chain has a last row of positive degree that reduces the row before it to zero. A chain has at most $d+1$ rows, the size of the table of the C code.

Lemma 4.8 (The chain detects a repeated factor)**.** *Three consecutive rows of a chain satisfy $A\,s_k = Q\,s_{k+1} - g\,s_{k+2}$ with integers $A, g > 0$ and some $Q \in \mathbb{Q}[X]$. For $d \geq 1$ and $c_d \neq 0$, the chain is none if and only if f is not squarefree over $\mathbb{Q}$. So for a list c of at least two entries whose leading zeros do not lower $\lceil n/2 \rceil$, the chain of the list that the reversal check hands on, with the degree it returns, is none if and only if Φ_c is not squarefree.*

Proof. The relation is read off the arrays coefficient by coefficient. In a commutative ring, for units a and b, $ax = qy - bz$ makes y and z coprime exactly when x and y are; so coprimality passes down the chain from (f, f') to its last two rows, which a constant last row makes coprime and a last row of positive degree that reduces the row before it to zero makes not coprime. Over $\mathbb{Q}$, f and f' are coprime exactly when f is separable, that is squarefree. The last claim follows with Lemma 4.5: the degree handed on is at least 1, and Φ_c is squarefree exactly when the polynomial handed on is. □

The C code keeps the rows, not only the verdict, because the Sturm step goes on with them: the relation with positive A and g and the decreasing degrees make the rows positive multiples of the Sturm sequence of f and f', whose sign changes the step counts (Chapter 5). The model keeps each row as the array of its coefficients up to its degree, where the C code keeps a table of GMP integers with an array of degrees.

4.5 The points at infinity

The points at infinity of the polynomial searched are the triples $(1, y, 0)$ with $y^2 = c_D$: for odd d the one point $(1 : 0 : 0)$, for even d the points $(1 : \pm\sqrt{c_d} : 0)$ when c_d is a square, and none otherwise. After a reversal they are the points $(0 : y : 1)$ of the caller's curve.

Definition 4.9 (The points at infinity)**.** For the callback, the list c of degree d, the flags of the check and a delivery whose flag of the stop is clear: when d is odd or c_d a square, with the candidate $(a, b) = (1, 0)$, or $(0, 1)$ when the flag of the reversal is set, one call with $(a : 0 : b)$ under noCheck or for odd d, and otherwise the call with $(a : s : b)$ for $s = \lfloor\sqrt{c_d}\rfloor$ and then, unless that call stopped the search or noY is set, the call with $(a : -s : b)$; nothing otherwise (`find_points.c:4894-4896`, `find_points.c:5068-5074`, `find_points.c:5078-5099`). The contract: on a delivery whose flag is clear these calls are the delivery (Definition 2.13) of the list of $(1, 0, 0)$, or of $(1, s, 0)$ and, unless noY, $(1, -s, 0)$, each mapped by the map of the reversal when its flag is set, and of the empty list when there are no points at infinity; without noCheck and for odd d or $c_d \neq 0$, this list is that of the brute-force reference over the candidate $(1, 0)$, mapped after a reversal and without the point with $y < 0$ under noY.

Like the C code, the model has no test $s \neq 0$, which would matter only for even d and $c_d = 0$ (the point $(1, 0, 0)$ twice), excluded by the refusals. The C code computes the flag `lcfsq` before the reversal check and clears it in case 2; the model tests c_d on the polynomial searched, which agrees for even degree, the only one that reads the flag. The search delivers the points at infinity after the choice of the moduli and returns when a call stops it there (Definition 3.2).

4.6 The exit by a prime

While it chooses the moduli (Chapter 6), `ratpoints` computes for each prime p it examines the table of the residues x with $f(x)$ a square modulo p, with a last entry for the points at infinity modulo p (Chapter 7). When no entry is true, `examine_prime` returns -1 (`find_points.c:1762-1762`) and the search ends with no points (`find_points.c:2613-2614`, `find_points.c:2994-2998`, `find_points.c:4996-5012`). The model's table with the exit, `Ratpoints.Sieve.examinePrime?`, is then none, and otherwise the reference table `Ratpoints.Sieve.examinePrime`, for every prime below maxPrimeEven $= 2^{\text{maxBitsInPrime}}$, the bound of the moduli of the sieve, below which lie all the primes that the choice examines.

Lemma 4.10 (A curve without points modulo p has none)**.** *Let p be a prime and d the degree of c. Integers a and y and a natural number b with $\gcd(a,b) = 1$ and $y^2 = F_c(a,b)$ make some entry of the table of p true: for $p \nmid b$ the entry of the residue ab^{-1}, and for $p \mid b$ the entry at infinity, since then c_d is a square modulo p when d is even. So if $p <$* maxPrimeEven *and the table of p with the exit is none, c has no normalized point of any height; this holds in particular for the prime at each position below* numPrimes *of the table.*

Proof. For $p \nmid b$ the row of b admits a (Lemma 6.4), so the entry of ab^{-1} is true. For $p \mid b$, $b = 0$ included, $p \nmid a$ and $y^2 \equiv c_D a^D \bmod p$, so the entry at infinity is true: for even d because c_d is then a square modulo p, for odd d always. For $0 < p <$ maxPrimeEven the table with the exit is none exactly when no entry is true (Lemma 7.1). □

When the exit fires the curve has no points at all, so it changes no output. After a reversal it never fires: each case leaves the caller's curve a point of height 1, $(0 : 0 : 1)$ in cases 1 and 3 and the points at infinity in case 2.

Chapter 5

The search region and the Sturm step

The caller of `find_points` may restrict the search to *search intervals* for x/z, closed intervals whose ends are C `doubles`, and to a range of the denominators, from b_{low} to b_{high}. Chapter 2 specifies this search region (Definition 2.6); this chapter models what the C code does with it: the doubles, the default interval and the refusal of bad intervals, the exact bounds of the numerators of each denominator, and the Sturm step of `sturm.c`, which narrows the search intervals to those on which f may take a value ≥ 0; and it proves, assuming Sturm's theorem (Definition 2.8), that the Sturm step keeps every point and that its intervals fit the arrays that the C code gives it. The remainder chain of f and f', with which the Sturm step begins, is Chapter 4's; the main theorems (Chapter 3) use this chapter through the region that the search uses (Section 5.6).

`ratpoints` does no floating-point arithmetic on the ends of its intervals (Chapter 2). So the model holds a double as its bits, with a function for each operation that the C code applies to it, and computes everything else exactly; only the value of a double and well-formedness, the test of `RATPOINTS_BAD_ARGS`, read the bits.

5.1 The doubles

Definition 5.1 (The doubles of the C code)**.** A C `double` by its 64 bits of IEEE 754 binary64, in a structure with no equality of its own (equality of the bits is not `==`); and the operations that `ratpoints` applies to doubles, as gcc and glibc compute them with rounding to nearest, ties to even: the comparisons, false when either side is a NaN and otherwise the order of a key (the bits without the sign bit, negated when it is set); negation and `fabs`; the exact value of a finite double; the double nearest to $\pm m\,2^k$ and to $\pm n/d$; the conversions from and to `long`, and to `unsigned long`; and `frexp`, `ldexp` and `nextafter` towards $\pm$`HUGE_VAL`, which pass a signalling NaN on as it is, where glibc quiets it (`ratpoints` passes them no NaN). These functions model the platform and hold no C range.

Each operation is checked when the model is compiled, on the cases at the edges (zeros, subnormals, the largest finite doubles, infinities, a NaN, ties, 2^{53} and 2^{63} with their neighbours), against the bits that a C program built with gcc and glibc prints; for the roundings the proofs show what the theorems need, a faithful rounding.

Lemma 5.2 (The values of the doubles)**.** *Write $v(x)$ for the value of the bits of a double x (Definition 2.6).*

(1) *On doubles that are not NaNs,* `<`*,* `<=` *and* `==` *are the order and the equality of the values; the value of a finite double is its exact value.*

(2) *The double nearest to $\pm m\,2^k$ is exact for $0 < m < 2^{53}$, $k \geq -1074$ and $m\,2^k < 2^{1024}$, and* `(double)n` *for $|n| \leq 2^{53}$; for a finite $x \neq 0$,* `frexp` *writes x as $g\,2^e$ with $1/2 \leq |g| < 1$, and* `ldexp(x, n)` *is $x\,2^n$ when $2^{-1022} \leq |x|\,2^n < 2^{1023}$; and a double $0 \leq x < 2^{63}$ converts to the* `long` *$\lfloor x \rfloor$, one $0 \leq x < 2^{64}$ to the* `unsigned long` *$\lfloor x \rfloor$.*

(3) *The next double above x, not a NaN or $+\infty$, is at most every double above x; dually below.*

(4) *The roundings of $\pm n/d$ and of $\pm m\,2^k$ are never a NaN, and for $n, m \neq 0$ a finite result is a faithful rounding: the value lies strictly between the values of its two neighbours.*

(5) *For $1 \leq H < 2^{63}$, the double h of H, moved up to the next double when $h < 2^{63}$ and the* `long` *of h is below H, is finite and $H \leq v(h)$.*

Proof. The value is strictly increasing in the key; the comparisons and the neighbours follow. A significand $M < 2^{53}$ with the exponent q of its last place has the value $\pm M\,2^q$: item (2). For item (4) the rounded significand lies within half a unit of the exact quotient, the neighbours one unit away, also across a change of the exponent; item (5) follows. □

5.2 The search intervals

The library normalizes the range of the denominators and the depth `sturm` of the Sturm step in place and restores those of the caller afterwards (`find_points.c:4542-4545`, `find_points.c:4560-4563`); the model returns the normalized options instead. The search intervals and the Sturm step come after the reversal check (Chapter 4).

Definition 5.3 (The normalized options and the search intervals)**.** For the options of a search (Definition 2.11) and the height bound $H \geq 1$, the *normalized* options: the least denominator 1 for one below 1; the largest denominator H for one below 1 or above H; the depth of the Sturm step capped at DBL_MANT_DIG + 2 = 55, beyond which the ends of the intervals of the step would not all be doubles; and the switch of the reversal turned off, for the private bit `RATPOINTS_NO_REVERSE_AUTO`, when the caller gives search intervals or the normalized range is not $1, \ldots, H$ (`find_points.c:4659-4661`, `find_points.c:4678-4689`). The *search intervals* of a search: for none given, the default interval $[-h, h]$, where h is the double of H, moved up to the next double when $h < 2^{63}$ and the `long` of h is below H; otherwise the caller's intervals, or none, for `RATPOINTS_BAD_ARGS`, when an interval has not $l \leq u$ or does not begin above the end of the one before it, compared as doubles, so that a NaN end is refused too (`find_points.c:4831-4855`).

The search turns the answer none into the refusal badArgs; the test of a null pointer (`find_points.c:4830-4830`) is excluded as part of the interface of the library.

Lemma 5.4 (The normalized options and the search intervals)**.**

(1) *The normalized switch of the reversal is the switch and the whole box (Definition 2.6); the normalized largest denominator is* denominatorBound(b_{high}, H)*; a denominator $1 \leq z \leq H$ lies in the normalized range exactly when $b_{\text{low}} \leq z$ and ($b_{\text{high}} < 1$ or $z \leq b_{\text{high}}$), as in the search region; the normalized depth is at most* 55 *and negative exactly when the depth is.*

(2) *The search intervals are none exactly when the caller's intervals are not well formed; they are the caller's intervals when these are well formed and not empty; and for* $1 \le H < 2^{63}$ *they are well formed and, when the caller gives none, hold* a/b *for every* $b > 0$ *and* $|a| \le H$.

Proof. Item (1) by cases on the clamps. Item (2): the loop of the validation tests each index, and the chain of the specification, read index by index, is that test in the order of the values (Lemma 5.2); the default interval by item (5) of that lemma. □

5.3 The bounds of the numerators

For a denominator b the numerators of an interval $[l, u]$ are $\lceil bl \rceil, \dots, \lfloor bu \rfloor$ within the height bound. The C code computes them exactly from the doubles of the ends, on `long`s and in 128 bits where those do not suffice, and the model on the same words; the bounds are made once for the run (`find_points.c:3869-3870`).

Definition 5.5 (The exact bounds of the search intervals)**.**

(1) An *end* x (`find_points.c:3264-3271`, `find_points.c:3272-3291`, `find_points.c:3292-3299`): its sign, $x < 0$ as in C, so that -0.0 is not negative; for $|x| < 2^{63}$, $|x| = m\,2^{-s}$ with m odd or $s = 0$, except that s is capped at 127, where $m\,2^{-s} \ne |x|$ but $b\,m\,2^{-s}$ and $b|x|$ both lie strictly between 0 and 1 for $1 \le b < 2^{63}$; otherwise $m = 2^{63}$ and $s = 0$; and the rounding $r = 2^s - 1$ when the end wants $b|x|$ rounded up (a lower end with $x \ge 0$, an upper end with $x < 0$), else $r = 0$. Its *scaled* value at b, $\min(2^{64} - 1, \lfloor (bm + r)/2^s \rfloor)$, is computed on two words (`find_points.c:3231-3238`).

(2) The *form* (M, C, sh) of an end (`find_points.c:3304-3318`), for which $(bM + C) \gg sh$ fits a `long` for every $b \le b_{\text{high}}$: $(0, \mp H, 0)$ for an end at or beyond the height bound on its side ($x \le -H$ at a lower end, $x \ge H$ at an upper end); $(\pm m, C, s)$, with $C = 2^s - 1$ at a lower end and 0 at an upper end, when $s < 64$ and $m \le \lfloor (2^{63} - 1 - C)/b_{\text{high}} \rfloor$; and none otherwise.

(3) The *bounds* of the search intervals (`find_points.c:3323-3340`): the two ends of each interval, which is taken inline, with the forms of its ends, when $H \le 2^{63} - 65$ and both ends have a form, and otherwise gets the forms of an empty interval.

(4) The *bits* of an interval at b for a class of the numerators (`find_points.c:3410-3434`): $lo = \lceil bl \rceil$ and $hi = \lfloor bu \rfloor$ from the forms; the code 2 for $lo > H$; for $hi < -H$ the code 1 if the interval is taken inline, and otherwise what `interval_bits_exact` computes from the scaled ends (`find_points.c:3381-3406`); and otherwise the code 0 with the bits of the first and the last numerator of the class in $\max(lo, -H), \dots, \min(hi, H)$.

(5) The bounds of an interval *represent* a range $lo, \dots, hi$ at b when, for every class of a packing (Chapter 9: stride 2^k with $k <$ numStrides and $k \le 6$, residue $a_0 < 2^k$), their bits have the code 2 only for a range above H, the code 1 only for one below $-H$, and otherwise the bits of the numerators of the class in $\max(lo, -H), \dots, \min(hi, H)$. The bounds of a run *give* a list of ranges at b when there is one for each interval, represented by its bounds, each with $lo \le hi + 1$, increasing and strictly apart.

This is what the loop of the sieve of one denominator over the intervals needs (Chapter 9, `find_points.c:3471-3488`): the code 2 ends the denominator, the code 1 goes on to the next interval and the code 0 sieves the bits, so that the numerators of a denominator come in increasing order.

Lemma 5.6 (The bounds give the ranges of the numerators)**.** *The* range of the numerators *of b in an interval $[l, u]$ over the values runs from $\lceil bl \rceil$, clamped to $-H, \dots, H+1$, to $\lfloor bu \rfloor$, clamped to $-H-1, \dots, H$, an infinite end going to the end of its clamp.*

(1) *For $b > 0$ and $|a| \leq H$, a lies in one of the ranges of the intervals of D exactly when a/b lies in one of the intervals. For intervals with $l \leq u$ each range has $lo \leq hi + 1$, and for search intervals (Definition 2.6) each range ends below the start of the next.*

(2) *For a double x that is not a NaN, $1 \leq b < 2^{63}$ and $H < 2^{63}$, the end made of x, at either side, gives the same clamped numerators as x; and for $H < 2^{63}$ and $1 \leq b \leq b_{\text{high}} \leq H$ the bounds of an interval represent the range of the ends made of its doubles, on both paths of* `interval_bits`*.*

(3) *So, for well-formed intervals, $H < 2^{63}$, $b_{\text{high}} \leq H$ and $0 < b \leq b_{\text{high}}$, the bounds of the run give at b the ranges of the values of the intervals (Definition 5.5).*

Proof. Item (1): the ceiling and the floor of bx against an integer; the ranges are apart since $\lfloor bu_i \rfloor \leq bu_i < bl_{i+1} \leq \lceil bl_{i+1} \rceil$. Item (2): the product on two words with its carry gives the scaled end exactly for $s \leq 127$ (`Ratpoints.Sieve.toNat_exactScale`); the mantissa is exact by Lemma 5.2, and an end capped at $s = 127$ or beyond 2^{63} gives the clamped numerators of the double; a form $(bM + C) \gg s$ is $\lceil bm\, 2^{-s} \rceil$ for $C = 2^s - 1$ and the floor for $C = 0$, a `long` by the bound on m. Item (3): the loop of `exact_bounds` is a map. □

5.4 The Sturm step

With sturm ≥ 0, `_ratpoints_compute_sturm` narrows the search intervals before the sieve. From the remainder chain $s_0 = f, s_1 = f', \dots, s_k$ (Chapter 4) it counts the sign changes at $\pm\infty$; when f has no real root and is negative it stops; otherwise it bisects $[-\infty, +\infty]$ to the depth of the step, keeping the intervals on which f may be ≥ 0, and intersects them with the search intervals. The model keeps the `long` numerators and exponents of the ends, as `Int64`, and the doubles of the C code.

Definition 5.7 (The Sturm step)**.** The *bisection*, the nested function `iterate` of gcc (`sturm.c:197-208`, `sturm.c:228-233`, `sturm.c:261-267`), works on intervals whose ends are $n/2^e$, with `long`s n and e, or $\mp\infty$, written $\mp$LONG_MAX with the exponent 0. It evaluates the rows of the chain at $n/2^e$ by Horner's rule in integers, which computes $2^{e \deg s}$ times the value of the row s, and counts their sign changes, zeros skipped. An interval with the counts c_l, c_r at its ends and the signs σ_l, σ_r of f just inside them is dropped when $c_l = c_r$ and $\sigma_l < 0$; it is kept when $c_l = c_r$ and $\sigma_l > 0$, or at the depth of the step, its ends converted to doubles, $\mathrm{ldexp}((\mathrm{double})\, n, -e)$, and appended to the intervals kept, or joined to the last of them when that ends at its lower end (`==` of doubles); and otherwise it is split at its midpoint m, the left half bisected first, both halves with the count and the sign of f at m, except that at a root m the left half gets the count plus 1 and $-\mathrm{sign}\, f'(m)$, the right half $\mathrm{sign}\, f'(m)$. The midpoint is 0 for $[-\infty, +\infty]$; -1 or $2n_r$ for $[-\infty, n_r]$ and 1 or $2n_l$ for $[n_l, +\infty]$, as the finite end is 0 or not; and otherwise half the sum of the ends over the larger denominator. An interval is *well formed* at the depth δ when it has one of the four shapes that the bisection meets (the whole line; an infinite interval whose finite end, of exponent 0, is 0 or on its side of 0; two finite ends, equal exponents being 0), its exponents lie between 0 and δ, its finite numerators over the larger exponent are at most $2^\delta/4$ in absolute value from the depth 1 on, and its ends increase.

The *intersection* with the caller's intervals goes by two cursors: of the two current intervals, compared as doubles, the one that begins first is skipped when it ends before the other begins;

otherwise their intersection is noted and the cursor of the one that ends first moves on (sturm.c:289-322). The *Sturm step* on f of degree $d \geq 1$ with $c_d \neq 0$ (sturm.c:86-97, sturm.c:177-190, sturm.c:269-273): none when the remainder chain is none; no interval when the counts at $\mp\infty$ agree, so that f has no real root, and $c_0 < 0$; otherwise the intervals that the bisection keeps from $[-\infty, +\infty]$, started with the counts and the signs of f at $\mp\infty$, intersected with the caller's intervals. The *Sturm step of the search*, on the polynomial that the reversal check hands on, at the normalized depth (find_points.c:4879-4892): with the step off, the intervals as they are; the refusal notSquarefree when the chain is none; the intervals left, when there are some; and when there are none, none for an even d whose leading coefficient is not a square, so that the search returns at once, there being no point at infinity either, and otherwise no intervals with the flag `sturm_empty`, by which the search delivers the points at infinity only.

The proofs read f also as the real polynomial `Ratpoints.Sturm.chainPoly` with the coefficients $c_0, \ldots, c_d$.

Lemma 5.8 (What the Sturm step computes)**.**

(1) *Over an ordered field, a list of nonzero polynomials whose last entry is a unit and whose entries from the third on are* positive multiples *(c p with $c > 0$) of the negated remainders of the two before is, entry by entry, a list of positive multiples of the Sturm sequence of its first two entries; so the rows of a chain that ends with a constant are, over $\mathbb{R}$, positive multiples of the Sturm sequence of f and f'.*

(2) *Horner's rule gives the sign of the value of a row at $n/2^e$ when $e \geq 0$ and e times the size of the row is below 2^{63}. For a chain that ends with a constant, the counts of the step at $\pm\infty$ are those of the Sturm sequence of f and f'; if also $d < 2^{57}$ and f is squarefree, the signs and the count of the step at $n/2^e$, $0 \leq e \leq 55$, are the signs of f and f' and the count $V_f(n/2^e)$ of Sturm's theorem (Definition 2.8).*

(3) *The whole line is well formed at the depth 0, and the halves of a well-formed interval at a depth below 55 are well formed at the next depth, without overflow of the* `long`*s; the doubles of the ends have the values $n\,2^{-e}$, and $\pm 2^{63}$ for* ±LONG_MAX*. So the bisection to a depth of at most 55 keeps intervals with $l \leq u$, increasing and strictly apart.*

(4) *The step is none exactly when the chain is none. The intersection of two well-formed lists holds every value that lies in both. At a depth of at most 55, for well-formed intervals of the caller, the step and the Sturm step of the search leave well-formed intervals inside the given ones, none when* `sturm_empty` *is set; the Sturm step of the search refuses exactly when it is on and the chain is none.*

Proof. Item (1): the chain has $As_{j-2} = Qs_{j-1} - gs_j$ with integers $A, g \geq 1$ (Chapter 4); the rest by induction along the Sturm sequence, which commutes with the map from $\mathbb{Q}$ to $\mathbb{R}$. Item (2): positive multiples have the signs of the values and of the leading coefficients, and a squarefree f and f' never vanish together, which the count with the zeros skipped needs. Item (3): the cases of the midpoint, whose numerators stay at most 2^{53}, so that the doubles are exact (Lemma 5.2). Item (4): the loop of the intersection as a recursion on the two lists. □

5.5 Sturm's theorem and the coverage

The specification states Sturm's theorem as a proposition (Definition 2.8), in the form for an interval $(a, b]$ whose ends may be roots, which the bisection needs, as it evaluates the chain at

midpoints that may be roots of f. The general facts behind the proofs (signs near a point and near $\pm\infty$, sign variations of positive multiples, squarefree polynomials over a perfect field) are stated at Mathlib's generality under `Ratpoints/Mathlib`.

Lemma 5.9 (What Sturm's theorem says about an interval)**.** *For a real polynomial p let $V_p(t)$ be the number of sign changes at t of the Sturm sequence of p and p', $V_p(+\infty)$ that of the leading coefficients of its entries, and $V_p(-\infty)$ that of the leading coefficients each times $(-1)^{\deg q}$ for the entry q. For an interval with ends $l < r$ in $\mathbb{R} \cup \{\pm\infty\}$, the* count *at l is $V_p(l)$, and at r it is $V_p(r)$, plus 1 at a root; the* sign *at a finite l is that of $p(l)$, or of $p'(l)$ when $p(l) = 0$, at a finite r that of $p(r)$, or of $-p'(r)$ when $p(r) = 0$, and at $\pm\infty$ that of the leading coefficient of p, times $(-1)^{\deg p}$ at $-\infty$; for a squarefree p these are the signs of p just inside the ends. Assuming Sturm's theorem, for a squarefree p:*

(1) *the counts at the ends agree exactly when p has no root strictly between them;*

(2) *when they agree and p is negative just inside l, p is negative on (l, r) and just inside r;*

(3) *when they differ, or p is positive just inside l, p is positive somewhere in (l, r);*

(4) *when $V_p(-\infty) = V_p(+\infty)$ and $p(0) < 0$, p is negative everywhere.*

Proof. Sturm's theorem on $(l, r]$ counts the roots, a root at r giving the 1; at $\pm\infty$ the counts and the signs are those at points far enough out. Without a root, a sign just right of l holds on (l, r) by the intermediate value theorem; a root in (l, r) gives a point of positive value, since p changes its sign at a simple root. □

Lemma 5.10 (The Sturm step keeps every point)**.** *Let f have degree $1 \le d < 2^{57}$ with $c_d \neq 0$, and the depth be at most 55. When the chain does not fail, f is squarefree. Assuming Sturm's theorem:*

(1) *The* invariant *of the bisection is a well-formed interval whose counts and signs are those of f at its ends (Lemma 5.9). The bisection of an interval with the invariant keeps what the intervals kept before it hold and every $x \in \mathbb{Q}$ with $f(x) \ge 0$ between the values of the doubles of its ends, at a finite end only when f is positive just inside it.*

(2) *For well-formed intervals of the caller, the step leaves intervals that hold every $x \in \mathbb{Q}$ of the caller's intervals with $|x| \le 2^{63}$ and $f(x) \ge 0$. So the Sturm step of the search, for d the degree of c and when it does not refuse, puts every a/b of the given intervals with $b > 0$, $|a| \le 2^{63}$ and $F(a, b) \ge 0$ into an interval that it leaves,* `sturm_empty` *clear; this needs Sturm's theorem only when the step is on.*

(3) *The bisection keeps at most $1 + \lfloor d/2 \rfloor$ intervals, the size of the array* `ivlocal` *of the C code, and the step leaves at most $\lfloor d/2 \rfloor$ intervals more than the caller gives, within the room for $1 + \lfloor d/2 \rfloor$ more that* `sturm.c` *asks of the caller's array.*

Proof. Item (1) by the induction principle of the bisection: a dropped interval holds no x with $f(x) \ge 0$ (Lemma 5.9(2)); at a split the step computes the signs and the count of f at the midpoint (Lemma 5.8), so that both halves have the invariant and one of them keeps each x that the interval keeps. Item (2): the whole line has the invariant and keeps every $|x| \le 2^{63}$; at the exit $f < 0$ (Lemma 5.9(4)); and $F(a, b) = b^{2\lceil d/2 \rceil} f(a/b)$. Item (3): each kept interval has a point with $f > 0$ (Lemma 5.9(3)), $f < 0$ somewhere between two of them, and f changes its sign at most d times; an intersection has fewer intervals than its two lists together. □

5.6 The region that the search uses

The main theorems are stated over the search region of the caller (Definition 2.6); their proofs pass through the region that the search uses, the intervals that the Sturm step leaves.

Definition 5.11 (The region that the search uses)**.** A definition of the proofs, not of the model: for the polynomial c of degree d that the reversal check hands on, H and the options, none when the search intervals are none or the Sturm step of the search refuses or returns at once; no interval when it sets `sturm_empty`; and otherwise the values of the intervals that it leaves. For well-formed intervals of the caller:

(1) every x in the region lies in the caller's intervals, or the caller gives none;

(2) if the chain does not fail when the step is on, the region is none only for an even d whose leading coefficient is not a square;

(3) assuming Sturm's theorem when the step is on, for $0 < d < 2^{57}$ the degree of c, $1 \le H < 2^{63}$ and a chain that does not fail when the step is on: every a/b with $b > 0$, $|a| \le H$ and $F(a, b) \ge 0$ in the caller's intervals, or anywhere when there are none, lies in the region, which is not none.

Items (1) and (2) follow from Lemmas 5.4 and 5.8, item (3) from Lemma 5.10 and the default interval. With the exact check on, and assuming Sturm's theorem when the step is on, the depth of the Sturm step does not change the list of points that a search returns; Theorem 3.5 states this beside the order of the points.

5.7 Deviations

Two deviations of this part bear on what the proofs cover (Chapter 11 lists all of them by kind). The model of `exact_scale` forms the 128-bit product, sum and shift on two 64-bit words, for the `unsigned __int128` of the arm that the default build compiles; the other arm, for a build without a 128-bit type (`find_points.c:3239-3256`), is excluded. And the `long`s are `Int64`s, where a wrap would stand for an overflow of the C code, which the bounds on the forms and on the ends of the bisection exclude; Lemma 5.10(3) bounds the arrays of the Sturm step.

Chapter 6

The choice of the moduli and the blocks

The sieve of one denominator (Chapter 9) removes a numerator when the row of one of its moduli does not admit it. This chapter says what the moduli are, how the search chooses them, as `ratpoints` does in `sieving_info`, and how it runs over the denominators in blocks, between which it may correct its choice, as `adapt_primes` does. A modulus is a prime of the table of `ratpoints`, a power of such a prime or a product of such powers; its rows come from tables computed from the curve (Chapter 7). The search ranks the candidates by a model of what each costs and what it says, computed in doubles.

None of this changes the points that the search finds or their order, whatever the values of the doubles: the theorems hold for every choice that the search makes (Section 6.7).

6.1 The table of a prime and the loop of the tables

Write $f = \sum_{i \le d} c_i x^i$ and $F(x,z) = \sum_{i \le d} c_i x^i z^{D-i}$ with $D = 2\lceil d/2 \rceil$ (Definition 2.1). For a point (a, y, b) with $\gcd(a,b) = 1$ and a prime $p \nmid b$, $F(a,b) \equiv b^D f(ab^{-1}) \bmod p$, and b^D is the square of a unit, so that $f(ab^{-1})$ is a square modulo p, zero included; for $p \mid b$, $p \nmid a$. These two facts are all that the rows of a prime rest on.

Definition 6.1 (The table of a prime)**.** The primes of the sieve are the odd primes below 1024 in increasing order (`primes.h:29-40`), which `decide` checks; `ratpoints` chooses its moduli among the numPrimes of them below $2^{\text{maxBitsInPrime}}$, the 53 primes up to 251 for the 8 bits of the default build, whose Makefile passes `PRIME_SIZE` = 8 in place of the header's own default 7 (`rp-private.h:321-323`, `rp-private.h:325-361`). The table of a prime p has $p+1$ entries: entry $x < p$ says whether $f(x)$ is a square modulo p, and entry p whether there are points at infinity modulo p, that is, whether d is odd or c_d is a square modulo p; it is kept with p and the inverses modulo p (`rp-private.h:542-555`). The row of the denominator b admits the numerator a when, for $r = b \bmod p$ and $x = a \bmod p$, the entry at $xr^{-1} \bmod p$ is true if $r \neq 0$, and $x \neq 0$ if $r = 0$. This admission test is what the bit of a in the row of r says in ratpoints.

`ratpoints` computes this table in `examine_prime`, by Horner's rule with Barrett's reduction; that it is the table of Definition 6.1 is shown in Chapter 7.

Definition 6.2 (The loop of the tables)**.** The loop of `sieving_info` over the primes of the table while fewer than the number to look at, the normalized value of `-p`, have been looked

at (find_points.c:2601-2605). The examination of a prime (find_points.c:2606-2626) gives its table, the coefficients modulo p, its density (the share of the numerators that its rows let through) and a code: -1, the exit, when the curve has no points modulo p (Chapter 4); 0 when the prime says nothing; 2 when it says only that numerator and denominator are coprime; 1 otherwise. A prime that says something becomes a candidate with its key for the first stage, and its table of rows goes into the store; while fewer than their bound have been found, the coefficients modulo p give the forbidden divisor of p (Chapter 8). When the caller left the number of primes to the library, the loop looks at one more prime, at most numPrimes in all, while the first two stages would be short of candidates that say more than coprimality (find_points.c:2694-2705).

The contract: for options that look at no more than numPrimes primes, an exit means that the curve has no points of any height; otherwise at most numPrimes primes are to be looked at, each table that the loop keeps, that of a prime that says something, is that of Definition 6.1 for one of the first numPrimes primes, every candidate is a prime of the table with the sieve entry of its table, the forbidden divisors are sound, and the store holds the tables of rows of the primes.

6.2 Prime powers and products

ratpoints also sieves with odd composite moduli from 9 up to compositeMax $= 64$, products of at most three powers of primes looked at. The tables of a power $q = p^e$ of a prime, which examine_power computes (Chapter 7), test the numerator against f when $p \nmid b$, and the denominator against the reversed polynomial $\tilde{f}(t) = \sum_k c_k t^{D-k}$ when $p \mid b$, since then $p \nmid a$ and $F(a,b) \equiv a^D \tilde{f}(ba^{-1}) \bmod q$. No Chinese remainder theorem is needed for a product: its entry only intersects the conditions of its factors, and each factor needs only that its modulus divides the product.

Definition 6.3 (The sieve entry of a modulus)**.** The tables of a prime power $q = p^e$ are the bit arrays fsq, bit $x < q$ set when $f(x)$ is a square modulo q, and gsq, bit t set when $p \mid t$ and $\tilde{f}(t)$ is a square modulo q, with the inverses of the units (rp-private.h:531-534). Its row of b admits a, for $r = b \bmod q$ and $x = a \bmod q$, when bit xr^{-1} of fsq is set if $p \nmid r$, and when $p \nmid x$ and bit rx^{-1} of gsq is set if $p \mid r$. A factor is a prime with its table or a prime power with its tables. The sieve entry of a modulus m holds its factors, the reciprocal of m, with which the sieve reduces, the fields of the packing (Chapter 9), the reference to its table of rows and the kind of its builder (Chapter 7), its density, and the count of its factors, 0 for a prime, 1 for a prime power and their number for a product (rp-private.h:542-555). Its row of b admits a when the row of every factor admits $a \bmod m$ at $b \bmod m$; a numerator survives a list of entries for b when the row of b of every entry admits it, the interface between the sieve and the check.

The composite moduli that the choice offers (Ratpoints.addModuli) are the odd m from 9 up to compositeMax $= 64$ and below $2^{\text{maxBitsInPrime}}$ whose table the run can pay for and that factor into at most three powers of primes looked at, each among the first 64 primes of the table and each saying something (find_points.c:2346-2349); the entry of one that the choice takes is made from the codes of its factors (Ratpoints.makeModulus), with the inverse of rbaLength modulo m by the extended Euclidean algorithm (find_points.c:2414-2426, find_points.c:2434-2470).

Lemma 6.4 (The rows of a modulus let every point through)**.** *Let b be a natural number and a, y integers with $\gcd(a,b) = 1$ and $y^2 = F(a,b)$.*

(1) *For a prime p, the row of b in the table of p admits a.*

(2) *For a prime p and $e \geq 1$ with p^e within the words of the bit arrays, the row of b in the tables of p^e admits a.*

(3) *A factor is sound for m when its row of b* mod m *admits a* mod m *for every such (a, y, b), its inverses serve the scaling of the rows by a unit modulo m, its builder has what it needs, and its tables are those of Definition 6.1 or of* `examine_power`*. A prime of the sieve that divides m is a sound factor for m, and so is a power $p^e \mid m$ of one, for $e \geq 1$ and $0 < m < 2^{\text{maxBitsInPrime}}$.*

(4) *A sieve entry is sound when its modulus is positive and its row of b admits a for every such (a, y, b). An entry of an odd modulus $1 < m < 2^{\text{maxBitsInPrime}}$ with sound factors whose moduli multiply to m, the reciprocal and the bias of m and an inverse $x < m$ of* rbaLength *modulo m is an entry of m as the choice makes it (*IsModulusEntry*), whatever its density, count, reference and builder; such an entry is sound.*

Proof. The entry of the table of p at $t < p$ is true whenever $f(t)$ is a square modulo p. For $p \nmid b$, b is a unit modulo p^e, and in any commutative ring $F(kb, b) = b^D f(k)$, with b^D the square of the unit $b^{D/2}$; so $f(ab^{-1})$ is a square. For $p \mid b$, $p \nmid a$, which is all that the table of p asks; modulo p^e, $F(a, b) = a^D \tilde{f}(ba^{-1})$, by the same identity with a and b exchanged. Zero counts as a square, so a high power of p dividing $F(a, b)$ needs no case of its own. A factor whose modulus divides m reads b mod m and a mod m as it reads b and a, and an entry admits a when every factor does. □

6.3 The cost model

`ratpoints` ranks the candidate moduli by what each costs per word of numerators swept, divided by what it says: a modulus of density r says $-\log r$.

Definition 6.5 (The cost model)**.** A candidate modulus carries its density, key and cost, its mask of primes, its sieve entry and the codes of its factors (`find_points.c:147-149`). Its cost per word swept, in units of an `and` of the first stage, is a part for every word, the cost of its entry in the list of the moduli of a denominator and, in the first two stages, the costs of its set-up and, for a positive cost of a row, of its table of rows, spread over the U words and the V denominators of the run (`find_points.c:1220-1235`); its key is that cost over $-\log r$ (`find_points.c:1241-1247`), with a variant for the first stage (`find_points.c:1256-1270`). The rule of the first stage always takes the first candidate and then the next while the survivors per word that it removes, weighted by what a surviving word costs further on, exceed a target times its cost (`find_points.c:1583-1589`); the second stage adds as many moduli as an offset, 11 by default, scaled by $1/(1 + u_0/U)$ (`find_points.c:1619-1626`). The run is estimated from 64 samples of each segment of the loop over the denominators (`find_points.c:1939-1944`), and the cost of an exact check from the bit lengths of the coefficients and of H (`find_points.c:1285-1318`). Every expression is computed in `Float`, in the order of the operations of the C code; there is no contract.

Remark 6.6 (Why no proof reads a double)**.** The doubles are values of Lean core's `Float`, IEEE 754 binary64 with the arithmetic of the platform, as the C code, compiled without fused multiply-adds and without `-ffast-math`, computes them; `log` and `pow` are the functions of the platform's C library. No contract of the choice, the blocks or the correction reads the value of a `Float`: each states only what the natural numbers of the C code do, the cut points staying within the pool and the entries from the end of the second stage on counting no factor, which holds whatever the doubles are, NaN included. So the theorems hold for every choice that the cost model may make, and nothing is proved about the quality of the choice. Since glibc's `log` and `pow` are not correctly rounded and glibc chooses their variant by the processor, the choice of `ratpoints` depends on the machine; the model calls the same routines of the same library (on

glibc an older symbol version of them, with the same values on every argument that occurs here), and the differential tests compare the choices of the two programs on one machine (Chapter 11).

6.4 The choice of the moduli and of the cut points

In `sieving_info` (`find_points.c:2522-3103`) the first stage takes candidates in the order of their keys, and so does the second, after the rest is keyed again by the cost of a word that survives the first; each drops every candidate that shares a prime with one taken, so that the moduli of the first two stages are pairwise coprime. The third stage takes the prime of least density left while each pays for itself; the primes left after it are the reserve of the correction (Section 6.6).

Definition 6.7 (The choice of the moduli)**.** The choice for the curve, with the options of the sieve as the search normalizes them (Definition 2.11), takes these steps:

(1) the tuning, from `-r`, `-R`, `-U` and `-C` or the defaults (`find_points.c:2569-2575`);

(2) the first estimate, the loop of the tables with its exit, which returns the prime, and the further forbidden divisors (`find_points.c:2522-2532`);

(3) the estimate and the keys again, the composite moduli with their entries (`add_moduli`, Section 6.2) and the sort (`find_points.c:2808-2812`, `find_points.c:2816-2823`, `find_points.c:2826-2826`);

(4) the first stage, the value of `-n` or as many as its rule wants, and the second, the value of `-N` or the offset, each taking candidates in their order (`find_points.c:2490-2508`, `find_points.c:2843-2846`, `find_points.c:2864-2871`);

(5) the third stage, up to the value of `-P` more or while each prime pays for itself, looking at further primes, with the exit, when no prime left pays (`find_points.c:2973-2975`, `find_points.c:3009-3016`);

(6) the end: the third stage and the reserve sorted by density from s_2 on, the entries of all the candidates, their reciprocals and the store with the tables of rows of every entry (`find_points.c:3024-3034`).

The result holds the cut points $s_1 \le s_2 \le s_3$, the ends of the three stages, the number of primes looked at and the estimates of the run (`ratpoints.h:395-397`), with the sieve entries of all the moduli, the reserve included, their reciprocals, the forbidden divisors, the store of the tables of rows and the switch of the checked loop.

Lemma 6.8 (The choice is what the run needs)**.** *For options that look at no more than* numPrimes *primes, the checked loop only for even degree, and any values of the doubles: an exit of the choice means that the curve has no points of any height; otherwise the result is what the run needs:*

(1) *every sieve entry is sound (Lemma 6.4), with the tables of inverses of its factors and the fields that the packing reads;*

(2) *every entry that counts no factor has a single factor, a table of its modulus* $1 < p \le 2^{32}$ *whose table of inverses gives every nonzero residue below* p *a nonzero residue below* p*, and the reciprocal of* p*: the shape that the third stage of the sieve needs;*

(3) *the reciprocals are those of the moduli of the entries, in their order, and the store fits all the entries, the reserve included;*

(4) *the first decision, the ends s_2 and s_3 from block 0 on, is well formed: $s_1 \le s_2 \le s_3 \le$ the number of the entries, and every entry from s_2 on counts no factor;*

(5) *the forbidden divisors are sound (Chapter 8), and the checked loop is on only for even degree.*

Proof. The loop of the tables gives the state of Definition 6.2. The loops of `add_moduli` keep a pool (ChoicePoolWF) of primes with their entries and of composites whose codes name tables that the choice holds and whose factors multiply to the modulus, so that the entry made from such codes at a position is an entry of its modulus, with the reference and the builder of the position (IsChoiceEntry, Lemma 6.4). The stages keep every property of the candidates that does not read their key and cost and leave the cut points within the pool; the third stage exits only at a prime modulo which the curve has no points. The doubles enter only through values that no conjunct reads. □

So the result of the choice needs no well-formedness of its own. The search normalizes the options of the sieve before the choice (`Ratpoints.SieveOptions.normalize`), which bounds the number of primes looked at by numPrimes, so that the theorems need no hypothesis on the options.

6.5 The blocks of the run

The run cuts the steps of the loop of the shape (Chapter 8) into about 1024 blocks of consecutive steps and sifts each block with the moduli of the decision of the correction in force for it; the model sifts the blocks in turn, as `ratpoints` does without threads. No proof depends on the values of the constants of the blocks and of the correction but for the positivity of a block length that the build fixes (`Ratpoints.blockLen_pos`).

Definition 6.9 (The blocks of the run)**.** The steps of a shape form segments (`find_points.c:3812-3837`): the k of the squares k^2, for each divisor s of the divisor shape the k of sk^2, both within $[b_{\text{low}}, b_{\text{high}}]$, or the words of 64 denominators, or the denominators. The blocks have length $\lfloor \text{steps}/1024 \rfloor + 1$, at least 16, or the length that the build fixes (`find_points.c:3838-3842`), and cut each segment into consecutive pieces (`find_points.c:3843-3850`). A decision of the correction is the first block that it is for, with the ends s_2 and s_3 from that block on; the decision in force for block i is the last whose first block is at most i (`find_points.c:3908-3910`). Block i (`find_points.c:3889-3898`) hands each denominator of its steps that passes the tests of the shape to the sieve of one denominator, with the sieve entries cut by the decision in force into the first two stages, the first ending at s_1, and the third; it returns when the callback stops the search, and reports its counters with its decision.

Lemma 6.10 (A block delivers the points of its denominators)**.** *The plan of the blocks is well formed, with blocks of at least one step, and the denominators of the blocks, concatenated in order, are those of the shape from b_{low} to b_{high} in the order of its loop. When the first decision is for block 0, the decision found for block i is the one in force. Let the callback not have stopped the search, the pieces have at least one bit array, the sieve entries be fit for the run (of the shape that the third stage needs where they count no factor, with the tables of inverses of their factors, their reciprocals in order and the row offsets of the classes; soundness is not needed here), the word of the classes have bit j set exactly when class j admits a numerator, the decision in force be well formed, the forbidden divisors be as the shape of the words needs them, i be below the number of the blocks, $1 \le b_{\text{low}}$, $b_{\text{high}} < 2^{63}$, $H < 2^{63}$, and the store fit all the entries. Then block i keeps*

the store so and delivers, in order, over the denominators of its steps that pass the tests of the shape, the points that the exact check finds among the numerators that survive the first s_3 entries of its decision, for every choice of the words of the classes and the ranges of the numerators that the packing and the bounds give; and when the callback has not stopped the search within it, its report is a closed form in the arguments of the run, which reads neither the callback nor the flags of the output.

Proof. The plan is arithmetic of the quotients by the length, which is positive. The loop of each shape, for a step with a contract on each denominator, delivers the concatenation of their lists in order; a stop of the callback ends it with the delivery of a prefix, which is that of the whole list (Lemma 10.2). The step is the sieve of one denominator (Definition 9.11) with the entries $E_{:s_2}$ and $(E_{s_2:})_{:s_3-s_2}$, which concatenate to $E_{:s_3}$. Building rows never removes one, so that the store keeps fitting the entries. □

6.6 The correction of the moduli

The sieving thread counts, for each block, the words that the first stage sweeps, the bit arrays that survive it, the numerators extracted, those prime to the denominator, those that pass the third stage and the calls of the sieve, all before the exact check, so that the counts depend neither on the flags of the output nor, in a block in which the callback does not stop the search, on the callback. From the rates that the run has shown, `adapt_primes` fits the survivors per word as a linear function of the product of the densities; with `-A 2` or more (not by default) it walks the end s_2 of the second stage up through the entries beyond it, or down, above s_1 and not past a composite modulus; and without `-P` it computes the end s_3 of the third stage again from the new s_2. A decision made at the fold of block j is for the blocks from $j + \text{adaptLag} + 1$ on, with the lag $\text{adaptLag} = 32$ of `RP_ADAPT_LAG` (`find_points.c:1379-1381`).

Definition 6.11 (The correction of the moduli)**.** The counters of the run, the sums of the counters of the blocks folded (Definition 9.1), those from the extracted numerators on restarted when a block sifted with another s_2 is folded (`find_points.c:1363-1367`). The correction (`find_points.c:1383-1388`), when the run has swept enough words, sets the next one at twice the words swept and computes the rates, the walk of s_2 under `-A 2` or more and the end of the third stage, every double in the order of the C code. The fold of block j (`find_points.c:4114-4127`) adds its counters to the state of the correction and appends a decision for the blocks from $j+\text{adaptLag}+1$ on when the correction changes an end. The run starts from the decision of the choice for block 0, sifts each block with the decisions made before it and folds its report, returns after the block in which the callback stopped the search, and gives the ends s_2 and s_3 of the last block folded, the moduli used (`find_points.c:5152-5160`).

Lemma 6.12 (The correction keeps the decisions well formed)**.** *For any values of the doubles: when the latest decision and the decision with $s_3 = s_2$ for the end s_2 under which the counters of the run were counted are well formed, so is the decision that the correction makes. A state of the correction is well formed when its first decision is for block 0 and its decisions, the latest one and the one for the end of the counters are well formed; the fold of a block whose decision is one of the state keeps it so, adds one block folded and appends decisions only for the blocks from* $\text{folded} + \text{adaptLag} + 1$ *on. The state after the folds of the first k blocks, each with its report in closed form (Lemma 6.10), and the decision of each block are functions of the arguments of the run that read neither the callback nor the flags of the output, well formed when the decision of the choice is.*

Proof. The walk up ends at most at the number of the entries and passes only entries from the old s_2 on; the walk down ends at least at s_1 and passes only entries that count no factor; the third stage ends between the new s_2 and the number of the entries. The contracts of these loops state only what the natural numbers do, also where the walks compare costs that may be NaN. The closed form follows by induction on the number of folds. □

6.7 Why the choice does not change the output

By the block lemma (Theorem 3.13) and Lemma 6.12, the run delivers, over each denominator b, the points that the exact check finds among the numerators that survive the first $s_3(b)$ entries, with $s_3(b)$ independent of the callback and the flags (`Ratpoints.Sieve.exists_run_sieveRun_eq_deliver`); every entry is sound (Lemma 6.8), so every point survives, whatever the choice and the corrections. The choice changes only which candidates reach the exact check, and so the output under `RATPOINTS_NO_CHECK`, the numbers of the moduli used and the report under `-v` (Chapter 10). The statement under `RATPOINTS_NO_CHECK` that compares its output with that of the exact check (Theorem 3.8) compares two searches with the same options of the sieve and the same region, which differ only in the flags of the output (the second has the exact check on and `NO_Y` set); neither the choice nor the decisions read these flags or the callback.

With threads (`-t`) `ratpoints` sifts the blocks apart in a pool, which the model leaves out (Chapter 11). The lag and thread lemmas (Theorems 3.14 and 3.15) prove that blocks sifted apart, block i once the blocks before block $i - \mathrm{adaptLag}$ are folded and each with a store of its own, and delivered in their order give what the run delivers; that the pool of the C code works so is read off the C code, not proved.

Chapter 7

Tables and rows

This chapter describes how `ratpoints` computes the tables of its moduli and how it builds and keeps the rows that the sieve reads (Chapter 9). Chapter 6 says what the tables say and why their *admission tests* let the numerator of every point through: the row of a denominator b in the sieve entry of a modulus admits a numerator a when the tables allow $F(a, b)$ to be a square. Each builder is proved to build the *closed form* of its row, the array of words whose bits are the values of the admission test, so that every row of a run lets every point through. The contracts of the tables, which reason modulo p and p^e, use Mathlib, the others Lean core and `Std` only; the proofs use that the constants rbaPack and chunk (Chapter 9) are positive, never their values.

7.1 The tables as `ratpoints` computes them

`examine_prime` (`find_points.c:1683-1826`) computes the table of a prime p (Definition 6.1) by Horner's rule in `unsigned longs` with Barrett's reduction, on a fixed schedule (`find_points.c:1628-1655`, `find_points.c:1716-1747`): below maxPrimeEven $= 2^{\text{maxBitsInPrime}}$ an accumulator below p survives hornerSteps $= 64/\text{maxBitsInPrime} - 1$ steps $s \mapsto sa + c$ within 64 bits, and `ratpoints` examines no larger prime. For a factor $q = p^e$ with $e \geq 2$, `examine_power` (`find_points.c:2254-2331`) computes the tables of Definition 6.3.

Lemma 7.1 (The table of a prime as `ratpoints` computes it)**.** *Barrett's reduction of u modulo p with the reciprocal m is $r = u - \lfloor um/2^{64} \rfloor p$, and $r - p$ when $r \geq p$ (`find_points.c:1639-1650`); the value of f at a modulo p is Horner's rule from c_d on the coefficients reduced modulo p, with Barrett's reduction at the end and, for $d >$ hornerSteps, after every* hornerSteps *steps (`find_points.c:1726-1731`, `find_points.c:1737-1744`); the table is made of these values and handed back with the coefficients modulo p, or none, the exit, when no entry of it is true (`find_points.c:1683-1697`). Then:*

(1) *for $p > 0$ and $u < 2^{64}$, Barrett's reduction with $m = \lfloor (2^{64} - 1)/p \rfloor$ gives $u \bmod p$;*

(2) *for $p <$ maxPrimeEven, $a < p$ and coefficients below p, the schedule gives $f(a) \bmod p$, and the accumulator stays below $p^{\text{hornerSteps}+1} < 2^{64}$;*

(3) *for $0 < p <$ maxPrimeEven the computed entry, the prime with its table and its inverses, is that of Definition 6.1, with the coefficients modulo p, or none exactly when no entry of that table is true.*

Proof. Write $2^{64} - 1 = mp + t$ with $t < p$; then $u/p - um/2^{64} = u(1 + t)/(p\,2^{64})$ lies in $[0, 1)$, so the quotient is $\lfloor u/p \rfloor$ or one less. The invariant of Horner's loop: the accumulator is congruent modulo p to the value of the polynomial of the coefficients read, and below p^{k+1}, where $k \le$ hornerSteps counts the steps since the last reduction, and $p^{64/\text{maxBitsInPrime}} \le 2^{64}$ since $p < 2^{\text{maxBitsInPrime}}$, whatever the degree. For (3), both ways of reducing the coefficients agree, and the loop over the residues makes the entries of the reference table by (2). □

Lemma 7.2 (The tables of a prime power)**.** *The tables of $m = p^e$ as* `examine_power` *makes them from the coefficients reduced modulo m (`find_points.c:2267-2282`), in bit arrays of* modWords $= \lceil \text{maxPrimeEven}/64 \rceil$ *words, bit x in bit $x \bmod 64$ of word $\lfloor x/64 \rfloor$ (`rp-private.h:523-523`): the bits of the squares $x^2 \bmod m$, $x < m$, set in a bit array* `sq` *(`find_points.c:2283-2284`); the bits of* `fsq` *and* `gsq` *set where the values of f and of the reversed polynomial, by Horner's rule, are marked in* `sq`*; and the inverses by the extended Euclidean algorithm* `modinv` *(`find_points.c:2241-2252`), 0 for the residues divisible by p (`find_points.c:2302-2303`). Then:*

(1) *for $1 < m$ and x prime to m,* `modinv` *gives the inverse of x modulo m in $[0, m)$; for a prime p and $e \ge 1$, every entry of the inverses is below p^e, and the entry of a unit $x < p^e$ is its inverse;*

(2) maxPrimeEven $\le 64\,$modWords*, and for $m \le 64\,$*modWords *a bit array of* modWords *words gets the bit of every square modulo m set;*

(3) *for a prime p, $e \ge 1$ and $p^e \le 64\,$*modWords*, the tables of p^e have these inverses, bit $x < p^e$ of* `fsq` *set whenever $f(x)$ is a square modulo p^e, and bit $t < p^e$ of* `gsq`*, for $p \mid t$, set whenever the reversed polynomial of the even degree $D = 2\lceil d/2 \rceil$ takes a square value at t; only this direction is needed, since a further bit loses no point.*

Proof. The Euclidean loop runs on remainders r_i and cofactors s_i from $r_0 = m$, $r_1 = x \bmod m$, $s_0 = 0$, $s_1 = 1$, with the invariant $r_i \equiv s_i x \bmod m$, $\gcd(r_0, r_1) = \gcd(m, x)$, $r_0 > 0$, cofactors of opposite signs with $|s_0|\,r_1 + |s_1|\,r_0 = m$; it ends since r_1 decreases, with $r_0 = 1$, and keeps the cofactors in $[-m, m]$, the range of their `longs`. For (3), a value that is a square modulo m is $y^2 \bmod m$ for some $y < m$, so that its bit in `sq` is set by (2), and the loops over the residues and the multiples of p set the bit of the residue. □

7.2 The closed form of a row

A row of the modulus m has $m + \text{chunk} - 1$ bit arrays of rbaPack words; bit t of word j stands for the number $64j + t$ (`init.c:252-258`), and the last chunk $- 1$ bit arrays are copies of the first ones, for the chunked first stage. Which numerator a bit stands for is the business of the packing (Chapter 9).

Lemma 7.3 (The closed form of a row)**.** *The* closed form *of the row of a residue r modulo m, for an admission test* adm*, which says whether the row of r admits a number N, is the array of $(m + \text{chunk} - 1)\,$*rbaPack *words whose word j has bit t set exactly when $t < 64$ and* adm(r) *admits $64j + t$. With the admission test of a sieve entry, of the table of a prime or of the tables of a prime power, it is the row that* `ratpoints` *builds; it has no C code of its own. Then:*

(1) *bit t of word $j < (m + \text{chunk} - 1)\,$*rbaPack *is $t < 64$ and* adm$(r)(64j + t)$*;*

(2) *for $m > 0$ and a test that does not change when a multiple of m is added to the number, word j is word $j \bmod m\,$*rbaPack*, so that the closed form of a row of a sieve entry of positive modulus repeats its first m bit arrays after them;*

(3) *a word* holds $n \le 64$ *bits of a pattern* g *of period* m *from* s *on when its bit* $t < n$ *is* $g((s+t) \bmod m)$ *and its other bits are clear; if* $g(N \bmod m) = \mathrm{adm}(r)(N)$ *for every* N*, an array of* $(m + \mathrm{chunk} - 1)\,\mathrm{rbaPack}$ *words whose word* j *holds the 64 bits of* g *from* $64j$ *on is the closed form; the contracts of the builders conclude by this.*

Proof. Two words are equal when their bits are. For (2), the numbers $64j + t$ and $64(j \bmod m\,\mathrm{rbaPack}) + t$ differ by a multiple of m, and the test of a sieve entry reduces the numerator modulo its modulus first. For (3), both bits are $g((64j + t) \bmod m)$. □

7.3 The tables of rows and the builders

A sieve entry of `ratpoints` (rp-private.h:542-555) holds the array `sieve` of the rows of its residues, `NULL` for a row not built, and the pointer `init` to its builder. A prime has its row of 0 from the table `sieves0` that the build generates (gen_find_points_h.c:102-134, find_points.c:1820-1821) and the others absent; a prime power has one entry, made when a modulus first needs it, and a product an entry of its own that points to the entries of its factors (find_points.c:2414-2475). The builders are generated for each prime from `CODE_INIT_SIEVE1` for $p < 64$ (init.c:118-183) and from `CODE_INIT_SIEVE2` for $p > 64$ (init.c:186-244); a prime power has `_ratpoints_sieve_init_power` (init.c:355-393) and a product `_ratpoints_sieve_init_product` (init.c:395-445). A builder publishes its row in the entry, where it stays for the run.

Definition 7.4 (The tables of rows)**.** A *reference* to an entry, for a pointer to it: a prime by itself, a prime power by its prime and its exponent, a product by the position of its modulus among the moduli. The *store* maps each reference to its table of rows, m rows for the modulus m, each present or absent; *publishing* a row puts it into the table of a reference and leaves a store without that table as it is. The kinds of builder, for the pointer `init` (rp-private.h:497-498): the function of a prime below 64 from `CODE_INIT_SIEVE1` and that of a prime above 64 from `CODE_INIT_SIEVE2`, which the primes of the table have (gen_init_sieve_h.c:48-62), and the builders of a prime power and of a product. A sieve entry holds the reference to its table of rows and the kind of its builder: for a prime or a prime power those of its one factor, for a product its own (`Ratpoints.Sieve.SieveEntry.refInit`).

Lemma 7.5 (The rows of a prime)**.** *The row of* 0 *of a prime* p*: the* p *words of the forbidden divisor* p*, whose word* r *has bit* t *clear exactly when* $p \mid 64r + t$ *(`Ratpoints.Sieve.forbiddenRow`), written* rbaPack *times and followed by the copies (gen_find_points_h.c:122-134). The other rows by one function for the primes below* 64 *and one for those above, from the table of the prime, the reference, the residue* b *and the store: bit* $a < p$ *of the pattern the entry of the table at* $ad \bmod p$ *for the inverse* d *of* b*, by a loop that steps four residues by* $4d \bmod p$ *(init.c:72-83), rotated into the first* p *words, the copies (init.c:99-111), and the row published in the table of the reference (init.c:178-183, init.c:239-244). Then:*

(1) *for* $p > 0$ *the row of* 0 *is the closed form of the row of* 0 *of every table of* p*, and its first* p *words are those that the forbidden divisors read (Chapter 8);*

(2) *for the table of a prime* $p < 64$ *and* $0 < b < p$ *whose inverse in the table is below* p*, the function from* `CODE_INIT_SIEVE1` *returns the closed form of the row of* b*, and the store with it published in the table of the reference; the same holds for* $p \ge 64$ *with* $p \bmod 64 \ne 0$ *and the function from* `CODE_INIT_SIEVE2`*.*

The builders need no primality and nothing of the table of squares.

Proof. The loop keeps four lanes at the residues $s + (a + k)\,d \bmod p$, $k < 4$, and its word holds the bits below a of the pattern. `CODE_INIT_SIEVE1` rotates the words of the row out of two words by the shifts $\delta = (\lfloor 64/p \rfloor + 1)\,p \bmod 64$ and $64 - \delta$, `CODE_INIT_SIEVE2` out of a ring of $\lfloor p/64 \rfloor + 1$ words by $p \bmod 64$ and $64 - (p \bmod 64)$; every shift executed lies in $[0, 63]$, where the shifts of the C code are defined, since $\delta \in [1, 63]$ for $p < 64$ and $p \bmod 64 \neq 0$ otherwise. With the copies, word j holds the 64 bits from $64j$ on of the pattern read at $Nd \bmod p$, the admission test at $b \neq 0$, and Lemma 7.3(3) concludes. At $b = 0$ every lane would stay at 0, which is why the row of 0 comes from `sieves0`, whose words are right since the multiples pi, $i < 64$, are the multiples of p below $64p$. □

Lemma 7.6 (The rows of a prime power)**.** *The layout of a pattern of period m: word $x <$ m* rbaPack *of the row the 64 bits from the offset $64x \bmod m$ on of the pattern repeated through $w = \lfloor (m + 127)/64 \rfloor$ words, then the copies (`init.c:259-267`, `init.c:280-297`). The row of the residue b of a prime power m, from its tables, the reference and the store: for a unit b, bit x of the pattern set when bit $xb^{-1} \bmod m$ of* `fsq` *is (`init.c:369-375`), and for $p \mid b$, bit x of a unit x set when bit $bx^{-1} \bmod m$ of* `gsq` *is (`init.c:378-385`); the pattern laid out, and the row published in the table of the reference (`init.c:391-393`). Then:*

(1) *for $0 < m \leq 64$* modWords *and a pattern whose bits x with $m \leq x < 64\lceil m/64 \rceil$ are clear, word j of the row holds the 64 bits of the pattern, read with period m, from $64j$ on;*

(2) *for $b < m \leq 64$* modWords *whose entry in the table of inverses is below m, the builder returns the closed form of the row of b, and the store with it published in the table of the reference.*

Proof. The repeated pattern has at every bit $N < 64w$ the bit $N \bmod m$ of the pattern, and $64w \geq m + 64$, so the window at $64x \bmod m$ is the window at $64x$ of the periodic pattern; the C code's guard against a shift by 64 keeps every shift in $[1, 63]$. The patterns set the bits that the admission test of the prime power reads: `fsq` at xb^{-1} for a unit b, and for $p \mid b$, `gsq` at bx^{-1} for a unit x and nothing for $p \mid x$. □

Lemma 7.7 (The rows of a product)**.** *The row of the residue b of a product: the rows of its factors at b modulo their moduli, read from their tables or built into them through the pointer of the entry of each factor (`init.c:407-414`); the first laid down in blocks of its length (`init.c:419-423`), the others and-ed in (`init.c:424-429`), the copies (`init.c:430-430`), and the row published (`init.c:443-445`). A factor is* buildable *when its builder has what it needs: a prime $p > 0$ with its inverses below p and $p \bmod 64 \neq 0$ for $p \geq 64$, or a prime power $0 < m \leq 64$* modWords *with its inverses below m; it may be called below its modulus, for a prime not at 0. Then:*

(1) *a buildable factor called where it may be called returns the closed form of its row and publishes it in the table given;*

(2) *if the product has a positive modulus m and a factor, the moduli of its factors divide m, its factors are buildable, the rows present in their tables are closed forms, a row absent below the modulus of a factor is one where its builder may be called, and two factors with one reference have the same closed forms, then the builder returns the closed form of the row of b, and the store has* grown *by rows of the factors (the tables keep their sizes, no row is replaced, every new row is the closed form of a factor whose table it is in), with the row of the product published.*

Proof. (1) is Lemma 7.5 or 7.6, by the kind. For (2), the invariant of the loop over the factors: their rows before the position of the loop are closed forms, and the store has grown by rows of

the factors; a row read is a closed form also when a factor before it with the same reference built it. Each modulus q of a factor divides m, so word y of the row is the and of the words $y \bmod q$ rbaPack of the rows of the factors, as the admission test of the product is the and of theirs (`Ratpoints.Sieve.toArray_eq_closedRow_of_productWord`). □

7.4 The rows of a run

The sieve sets up a denominator by `set_rows` (`find_points.c:3149-3170`), the row of the residue of each modulus of the first two stages from its entry, perhaps absent. It builds each absent row by the function of the entry, under the lock of the tables when threads share them and after reading the pointer again, since another thread may have built it in the meantime (`find_points.c:3522-3554`); the model, which has no threads, reads it again as well. Within a run the moduli of the first two stages are pairwise coprime (Chapter 6), so that each table of rows is reached through one modulus; the contracts also admit entries that share a table. The run hands the store on from one denominator and one block to the next.

Definition 7.8 (The tables of rows of a run)**.** The table of rows of a prime p: its row of 0 that of Lemma 7.5, the other $p-1$ rows absent (`find_points.c:1820-1821`). The tables of rows of a sieve entry, each made only when its reference is absent: that of each factor, and for a product its own (`find_points.c:2427-2427`, `find_points.c:2471-2471`). The row of a residue b of a sieve entry *read or built*: the row in its table when it is there, and otherwise the row that its builder builds into it. The sieve specs of a denominator, as `set_rows` makes them (`find_points.c:3149-3170`): for each entry of the first two stages its modulus, the offset of the row of the class and its row at its residue of b, absent when it is not built, with the end p. The missing rows of a denominator: each sieve spec of the first two stages without a row gets the row read or built of its entry at its residue (`find_points.c:3522-3554`).

Lemma 7.9 (The rows of a run are closed forms)**.** *A store* holds *a function R of the references and the residues for a list of entries when every row present in the table of a reference ρ at r is $R(\rho, r)$ and the table of every prime factor of the entries has its row of* 0*. An entry at the position n* fits *R when it has the reference and the kind of builder of that position (and, with one factor, the modulus of that factor), a positive modulus, a factor, moduli of the factors dividing its modulus and buildable factors, and R at its reference and at those of its factors is the closed form of their rows; a store* fits *a list of entries when it holds some R for them that every entry fits. The* rows of the references *of a run are the closed forms of the rows of the table of a prime, of the tables of a prime power and of the entry of a product at its position; a store is* made for *a list of entries when it holds the rows of the references of every run whose entries begin with the list, and every table of rows of a prime in it, or of a prime factor of the entries, has its row of* 0*. Then:*

(1) *if the row present in the table of an entry at b is the closed form, and the builder has what it needs when the row is absent, the row read or built is the closed form of the row of b, and the store is the same or has grown by rows of the factors, with the row published;*

(2) *the store of the loop of the tables, which holds the rows of* 0 *of the primes and no others, is made for no entries; adding the tables of rows of further entries, each with the reference and the kind of builder of its position and with positive primes, gives a store made for the longer list; a store made for the entries of a run holds the rows of their references for them; and an entry of a modulus as the choice makes it (Chapter 6) fits the rows of the references of the run at its position;*

(3) *for a store that holds R for a list L, entries of L that fit R at their positions and residues below their moduli, the sieve specs that* `set_rows` *makes and the missing rows complete have the closed forms of the rows of their entries at their residues, the offsets of the class, the start* 0 *and the end* p*, and the store after them holds R for L; so a store that fits the entries of a run fits them after each denominator.*

Proof. (1) by the cases of the definition and Lemma 7.7. (2) The rows of 0 are `sieves0`, the closed form for every table of the prime, and a table made only when absent holds no other row; two entries that reach one table have one closed form, since R depends on the reference only and the rows of a prime power do not depend on the switch of its density (`Ratpoints.Sieve.admits_run_examinePower`). (3) The invariant of the loop over the specs: the store holds R, and the specs before the position of the loop have their closed forms; a row present is R at its reference, a row built is a closed form by (1), and a builder of a prime is never called at 0, whose row is present from the start. □

The choice of the moduli applies item (2) to its entries (`Ratpoints.IsChoiceEntries.fits_foldl_addTables`), which are sound (Lemma 6.4), and item (3) hides the store from the sieve of one denominator (Chapter 9).

Lemma 7.10 (Every row lets every point through)**.** *The row of b of a sound entry admits a for every triple (a, y, b) with $\gcd(a, b) = 1$ and $y^2 = F(a, b)$, and the numerator of every point with the denominator b that the brute-force reference lists (`Ratpoints.BruteForce.pointsOfDen`) survives every list of sound entries for b.*

Proof. A point of the reference has $\gcd(a, b) = 1$ and $y^2 = F(a, b)$, and a numerator survives a list of entries when every entry admits it. □

Every row that the run reads is the closed form of the admission test of its entry (Lemma 7.9), whose bit at a number is the test at that number (Lemma 7.3); so the sieve never clears the bit of the numerator of a point, which the membership theorem (Theorem 3.3) needs, through the contract of the sieve of one denominator (Definition 9.11) and the block lemma (Theorem 3.13).

7.5 Where the model differs from the C code

Chapter 11 summarizes the general differences (the generated tables as functions of the prime, two builder functions for the primes, the store for the arrays reached through pointers, no threads). The `long`s are natural numbers or integers, with the range obligations as lemmas: the accumulator of Horner's rule below 2^{64}, the cofactors of the extended Euclidean algorithm in $[-m, m]$, and the inverses of a prime power below 2^{16}, the range of their `unsigned short` (`Ratpoints.Sieve.getElem!_inv_run_examinePower_lt_two_pow_16`). Barrett's reduction has the form with the 128-bit product, which the default build compiles, and the branch of the schedule of Horner's rule is chosen for each residue, where `examine_prime` chooses it once. The probes `RP_INIT_NOACC` and `RP_INIT_NOREP` (`init.c:88-93`, `init.c:97-98`), whose tables are wrong by design, and the check `RP_VERIFY_MODULI` (`init.c:299-353`) are excluded.

Chapter 8

The denominators

Before it sieves the numerators of a denominator b (Chapter 9), `ratpoints` decides which denominators it sieves at all, in which order, and which numerators modulo 64 each class of the denominators modulo 64 admits. This chapter describes these choices and proves that none of them loses a point. They are made for the polynomial searched, of degree d with the coefficients $c_0, \ldots, c_d$, that the set-up hands on (Chapter 4), with its binary form $F(x,z) = \sum_k c_k x^k z^{D-k}$ of the even degree $D = 2\lceil d/2 \rceil$ (Definition 2.1). The lemmas speak of triples (a, y, b) of integers a, y and a natural number b with $\gcd(a,b) = 1$ and $y^2 = F(a,b)$, as every point (Definition 2.4) is one, and the denominators range from b_{low} to b_{high}, which are 1 and the height bound H unless the options narrow the range (Definition 2.6).

The 2-adic information says which numerators modulo 64 each class of $b \bmod 64$ admits (Section 8.1). For odd d the denominator of a point is a squarefree divisor of c_d times a square (Section 8.2). For even d `ratpoints` looks for primes that divide no denominator of a point, or only to certain powers (Section 8.3), and tests the denominators by a Jacobi symbol (Section 8.4). The loop over the denominators takes one of four shapes that combine these tests (Section 8.5). Each test removes only candidates that carry no point; what the tests change is the order of the points, and Section 8.6 proves that the model visits the denominators in the order of the specification (Definition 2.9). The forbidden divisors of the primes examined for the sieve come out of the loop of the tables of the choice of the moduli, which also cuts the loop over the denominators into blocks (Chapter 6); the class of b gives the sieve its first pattern and the packing of its numerators (Chapter 9).

In the C code, `find_points_work_1` sets the flags of the tests (`find_points.c:4607-4607`, `find_points.c:4905-4943`): for even d the checked loop and, when c_d is not a square and the caller does not forbid it (the option `-j` of the program), the Jacobi test; for odd d the squares when $|c_d| = 1$ and the divisors of `setup_us1` otherwise. The model keeps the order of the C code (`Ratpoints.searchPoints`, `Ratpoints.sievingInfo`, `Ratpoints.Sieve.sieveRun`).

8.1 The 2-adic information

Write $f(x) = F(x,1)$ and $\tilde{f}(t) = F(1,t)$, the reversed polynomial. In a pair (a,b) with $\gcd(a,b) = 1$, a or b is odd. For odd b, $F(a,b) \equiv b^D f(ab^{-1}) \bmod 64$, where b^D is the square of a unit since D is even; for even b, a is odd and $F(a,b) \equiv a^D \tilde{f}(ba^{-1}) \bmod 64$. So two words of 64 bits decide which pairs of residues modulo 64 can carry a point: the residues k at which $f(k)$ is a square modulo 64, and the even residues t at which $\tilde{f}(t)$ is one.

Definition 8.1 (The 2-adic information)**.** The words fsq and gsq, by Horner's rule on 64-bit words (`find_points.c:1006-1014`, `find_points.c:1015-1021`): bit k of fsq is set exactly when $k < 64$ and $\sum_{i \le d} c_i k^i$ is a square in $\mathbb{Z}/64$, bit t of gsq exactly when $t < 64$ is even and $\sum_{i \le d} c_i t^{D-i}$ is a square in $\mathbb{Z}/64$. The words nb[b] for $b < 64$, scattered from these (`find_points.c:983-994`, `find_points.c:1023-1050`): bit a of nb[b] is set exactly when b is odd and $a = kb \bmod 64$ for a set bit k of fsq, or $a < 64$ is odd and $b = ta \bmod 64$ for a set bit t of gsq. And den_bits, with bit b set exactly when nb[b] $\neq 0$. These descriptions are the contracts.

By the congruences above, bit a of nb[b] is set exactly when a or b is odd and $F(a, b)$ is a square modulo 64.

Lemma 8.2 (The class of a point)**.** *Let a and y be integers and b a natural number with* $\gcd(a, b) = 1$ *and* $y^2 = F(a, b)$*. Then bit $a \bmod 64$ of* nb[$b \bmod 64$] *is set. So the word of the class of a denominator over which the brute-force reference finds a point is not* 0*; and if* den_bits $= 0$*, the curve has no point of any height, the points at infinity included.*

Proof. For odd b, $f(k)$ is a square modulo 64 at $k = ab^{-1} \bmod 64$, and the scatter sets bit $kb \equiv a$ of nb[$b \bmod 64$]; for even b, $\tilde{f}(t)$ is a square at the even $t = ba^{-1} \bmod 64$, and the scatter sets bit $a \bmod 64$ in the word of $ta \equiv b$ (`Ratpoints.isSquare_sum_mul_pow_sub_of_sq_eq_binaryForm` and its companion, in any commutative ring); Horner's rule on 64-bit words is exact modulo 64. A point at infinity is a triple $(1, y, 0)$ and also sets a bit of den_bits. □

So the 2-adic exit, which ends the search when den_bits $= 0$ and there is no point at infinity, loses nothing, and its condition on the points at infinity always holds. Since bit 0 of fsq says that c_0 is a square modulo 64 and bit 0 of gsq that c_D is one, the exit is taken only for even d with c_0 and c_d non-squares modulo 64.

The classes of the numerators (`Ratpoints.Sieve.numClasses`, Definition 9.5) have the pattern 0 exactly when their word is 0. So the class test of the loop over the denominators, that the pattern of the class of b mod 64 is not 0, removes no denominator of a point, and the pattern, with which the sieve of b starts, removes no numerator of a point.

8.2 Odd degree: squares and divisors

For odd d, $F(a, b) = b\,G(a, b)$ with $G(a, b) = c_d a^d + b \sum_{k<d} c_k a^k b^{d-1-k}$. Let (a, y, b) be a triple as above and p a prime with $p \nmid c_d$ and $v_p(b) = m \ge 1$. Then $p \nmid a$, the term $c_d a^d b$ of $F(a, b)$ has the valuation m at p, and every other term has at least $2m$. When in a sum $y^2 = \sum_i f_i$ of integers one term $p^v u$ with $p \nmid u$ has a lower valuation than all others, v is even and u is a square modulo p (`Int.even_and_exists_isSquare_add_of_sq_eq_sum`); so m is even. Hence $b = sk^2$ with s squarefree and $s \mid c_d$, and b is a square when $|c_d| = 1$. For a prime $p \mid c_d$ there is a second test: when $v_p(b)$ is large enough, the term $c_d a^d$ alone has the least valuation in $G(a, b)$, so that $v_p(b) + v_p(c_d) = v_p(y^2)$ is even.

Definition 8.3 (The valuation)**.** The valuation v of an integer $n \neq 0$ at p with the unit part $(|n|/p^v) \bmod p$, by repeated division with remainder, and (veryBig, 0) for $n = 0$, with veryBig $=$ 1000 (`find_points.c:422-422`): one function for `valuation` on GMP integers and `valuation1` on `long`s (`find_points.c:424-454`). The contract: for $p > 1$ the loop ends, and for $n \neq 0$, v is the p-adic valuation of $|n|$ and the unit part is not 0.

Definition 8.4 (The divisors of the leading coefficient)**.** For odd d and $|c_d| \neq 1$, as `setup_us1` computes them: the factorization of $|c_d|$ by trial division over the primes of the table, a cofactor

below the square of the next prime taken as a prime (find_points.c:777-860); the squarefree divisors at most b_{high} by a doubling loop, at most maxDivisors = 512 of them (find_points.c:864-880); and for each prime p of c_d a record of p, $v_p(c_d)$ and the slope, the least $n \geq 1$ with $(d-k)\,n > v_p(c_d) - v_p(c_k)$ for all $k < d$ (find_points.c:894-912). The records and the divisors are returned exactly when the factorization ends with the cofactor 1 and there are fewer than maxDivisors divisors (find_points.c:914-935). The contracts say so, and: for $c_d \neq 0$ the factorization ends with the cofactor 1 exactly when `ratpoints` factors $|c_d|$ completely in the sense of the specification (FactorsByTrialDivision, Definition 2.9), and then lists the primes of $|c_d|$ in increasing order with their valuations; the slope satisfies $n \geq 1$ and $1 + v_p(c_d) - v_p(c_k) \leq (d-k)\,n$ for all $k < d$, where $v_p(0) =$ veryBig; for $c_d \neq 0$ and $b_{\text{high}} \geq 1$ the set-up returns divisors exactly when $|c_d|$ is factored completely and has fewer than maxDivisors squarefree divisors at most b_{high}, and these are then binaryDivisors$(|c_d|, b_{\text{high}})$, in its order.

The contract of the factorization rests on the invariant `Ratpoints.Setup.FactorLcfInvariant` of its loop. By `Ratpoints.Setup.foldl_extendDivisors_eq_filter`, the doubling loop lists the products at most b_{high} of the sublists of the primes, in the order of binary counting, the first prime the lowest bit, or a prefix of them.

Lemma 8.5 (The denominators of odd degree)**.** *Let d be odd, a and y integers and b a natural number with $\gcd(a,b) = 1$ and $y^2 = F(a,b)$.*

(1) *If $|c_d| = 1$, then b is a square.*

(2) *$b = sk^2$ for a squarefree s that divides c_d.*

(3) *If p is a prime with $v_p(b) \geq 1$ and $v_p(c_d) + 1 \leq v_p(c_k) + (d-k)\,v_p(b)$ for every $k < d$ with $c_k \neq 0$, then $v_p(c_d) + v_p(b)$ is even. So for $b \geq 1$ the record of any prime p with its slope does not exclude b, where a record excludes b when $v_p(b)$ is at least the slope and $v_p(b) + v_p(c_d)$ is odd.*

When the set-up returns divisors, for $c_d \neq 0$ and $b_{\text{high}} \geq 1$, they are the squarefree divisors of c_d at most b_{high}, each once.

Proof. Parts (1) and (2) are the argument at the beginning of this section. For (3), $p \nmid a$, and in $G(a,b)$ the term $c_d a^d$ has the valuation $v_p(c_d)$, every other nonzero term a larger one, so that $v_p(b) + v_p(c_d) = v_p(y^2)$ is even; when $v_p(b)$ is at least the slope, the contract of the slope gives the hypothesis. The divisors are binaryDivisors$(|c_d|, b_{\text{high}})$, each once (`Ratpoints.mem_binaryDivisors_iff`, `Ratpoints.nodup_binaryDivisors`). □

8.3 Even degree: the forbidden divisors

For even d, a prime p and a triple (a,y,b) as above with $v_p(b) = m \geq 1$, $p \nmid a$, and the term j of $F(a,b) = \sum_{j \leq d} c_{d-j}\, a^{d-j} b^j$ has the valuation $w_j + jm$, where $w_j = v_p(c_{d-j})$. If one term j_0 alone has the least valuation, that valuation is even, and for even j_0 the unit part r_{j_0} of c_{d-j_0} is a square modulo p: the argument of the Newton polygon. A valuation m at which this fails is one that no denominator of a point has at p. For $p \nmid c_d$ and $p \mid b$, further, $y^2 \equiv c_d a^d \bmod p$ with a^d the square of a unit; so when c_d is not a square modulo p, p divides no denominator of a point. Such primes are the forbidden divisors, of two kinds: primes tested on words of 64 denominators at once (`Ratpoints.Sieve.ForbiddenEntry`), and primes with a mask of excluded valuations (`Ratpoints.Sieve.ForbiddenVal`).

Definition 8.6 (The forbidden divisors)**.** Let d be even. The test of a valuation m at a prime p is true when, for the valuations w_j and the unit parts r_j modulo p, signs included, of the nonzero c_{d-j}, the least $w_j + jm$ is taken at one j_0 only and is odd, or j_0 is even and r_{j_0} is not a square modulo p (`find_points.c:2218-2228`); the mask of the forbidden valuations at p, a natural number, has bit m set for the $m \geq 1$ with $p^m \leq b_{\text{high}}$ that the test excludes (`find_points.c:2229-2234`). The forbidden divisor of an examined prime p (`find_points.c:2642-2676`): for $p \mid c_d$ and a nonzero mask, p with its words when the mask holds every $m \geq 1$ with $p^m \leq b_{\text{high}}$, and p with its mask otherwise; for $p \nmid c_d$, p with its words when c_d is not a square modulo p, as the entry at infinity of its table says. Then, unless c_d is a square, the further primes p of the table, from the first one not examined, with $p^2 \leq b_{\text{high}}$ and c_d not a square modulo p, with their words, while fewer than maxForbidden forbidden divisors are found (`find_points.c:2709-2772`). The contracts: a true test means that one term of least valuation excludes m, and the mask is as described. A forbidden divisor is *admissible* when it is the forbidden divisor of a prime of the table, made from the residues of the coefficients and the entry at infinity of its table, or, with words, a prime of the table with its words and its reciprocal at which c_d is not a square; from admissible entries the further primes add only admissible ones, whatever the cap.

The words of a prime p are those of `Ratpoints.Sieve.forbiddenRow`, p words with bit j of word r clear exactly when $p \mid 64r + j$, read from the generated pattern `sieves0` for the first numPrimes primes of the table and made as `sieving_info` makes them for the others (`Ratpoints.Sieve.extract_sieves0`). For even d, and while fewer than maxForbidden forbidden divisors have been found, the loop of the tables records the forbidden divisor of each examined prime (`find_points.c:2624-2626`); these are admissible (`Ratpoints.Sieve.isForbiddenEntry_of_run_examinePrime?_eq_some` and its companion), so that the choice of the moduli hands on admissible forbidden divisors only (`Ratpoints.IsSievingInfo`, Chapter 6).

Lemma 8.7 (The forbidden divisors keep every point)**.** *Let d be even, a and y integers and b a natural number with $\gcd(a, b) = 1$ and $y^2 = F(a, b)$. For every prime p and every b_{high}, bit $v_p(b)$ of the mask of the forbidden valuations at p is clear. An admissible forbidden divisor with words has the words of `Ratpoints.Sieve.forbiddenRow`, and for $1 \leq b \leq b_{\text{high}}$ its prime does not divide b; and for $b \geq 1$ no admissible forbidden divisor with a mask excludes b.*

Proof. The first statement is the Newton polygon at p (`Ratpoints.even_and_isSquare_of_sq_eq_binaryForm_of_even`), from the statement on a sum with one term of least valuation; bit 0 is never set. A prime with words from a mask has every valuation up to b_{high} excluded; one with $p \nmid c_d$ divides no denominator of a point by the congruence above, the entry at infinity of its table saying, for even d, whether c_d is a square modulo p. □

8.4 The Jacobi test

For even d, an odd prime $q \mid b$ with $q \nmid c_d$ makes c_d a square modulo q, by the congruence of Section 8.3. So for the largest divisor b^* of b prime to $2c_d$, the Jacobi symbol $J(c_d \mid b^*)$ is 1: the test rejects every denominator with an odd number of prime factors, counted with multiplicity, at which c_d is not a square. The C code computes the symbol by a binary algorithm or, when the leading coefficient allows it, by tables of the non-squares modulo its odd primes and quadratic reciprocity.

Definition 8.8 (The Jacobi tests)**.** The binary algorithm (`find_points.c:498-566`, `find_points.c:568-641`) for $b \geq 1$ and an integer ℓ: subtraction and halving on ℓ mod b and b, with the signs of $J(2 \mid m)$ and of the reciprocity, the gcd divided out of b when it is not 1. Its contract: the

result is $J(\ell \mid b^*)$, where b^* is the largest divisor of b prime to 2ℓ. The tables (find_points.c:683-708), for $\ell = \pm 2^v \prod_i q_i^{e_i}$ and $b_{\text{high}} \leq \text{mulModLimit} = 2^{32} - 1$: the odd primes q_i, at most jacobiPrimes = 16, each with its reciprocal and, for odd e_i, the table of the non-squares modulo q_i, at most jacobiTable = 8192 entries in all; and the sign for each b^* mod 8, the product of $J(-1 \mid b^*)$ for negative ℓ, $J(2 \mid b^*)^v$ and the signs of the reciprocity of the $q_i \equiv 3 \bmod 4$ with odd e_i; none when a prime of ℓ lies beyond the table or a bound is passed. The test of b (find_points.c:734-759) divides 2 and the q_i out of b with the reciprocals and passes when the sign of the result times its entries modulo the q_i of odd exponent is 1. The contracts: tables exist only for $b_{\text{high}} \leq \text{mulModLimit}$ and $\ell \neq 0$, and then describe ℓ as above, with primes of the table and their reciprocals; for such tables and $1 \leq b < 2^{64}$ with $bq < 2^{64}$ for every prime q of the tables, the test is the negated exclusive or of the sign and the entries at the part of b free of 2 and of these primes.

The contract of the binary algorithm rests on invariants of its loops; b^* is `Nat.coprimePart`, and the symbol is Mathlib's `jacobiSym`.

Lemma 8.9 (The Jacobi tests keep every point)**.** *When the tables of ℓ for b_{high} exist, the test by tables passes $1 \leq b \leq b_{\text{high}}$ exactly when $J(\ell \mid b^*) = 1$. Let d be even, a and y integers and b a natural number with $\gcd(a, b) = 1$ and $y^2 = F(a, b)$. Then $J(c_d \mid b^*) = 1$ for the largest divisor b^* of b prime to $2c_d$; so for $b \geq 1$ the binary algorithm gives 1 on b and c_d, and the test by tables of c_d for $b_{\text{high}} \geq b$, when they exist, passes b.*

Proof. The denominators are below 2^{32} and the primes of the table below 2^{10}, so the quotients by the reciprocals are exact (`Ratpoints.Sieve.mulDiv_magic`) and leave b^*; by reciprocity each $J(q_i \mid b^*)$ of odd exponent is the Legendre symbol of b^* mod q_i, which the table gives, times a sign by b^* mod 4. For a triple of even degree, c_d is a square modulo every prime of b prime to it (`Ratpoints.Sieve.isSquare_getD_degree_of_sq_eq_binaryForm`). □

8.5 The shapes of the loop

Definition 8.10 (The shapes of the loop over the denominators)**.** The four shapes with their data (find_points.c:3725-3728, find_points.c:3736-3749), which `run_setup` chooses from the flags and returns with the bounds of the search intervals (find_points.c:3778-3806), and their denominators from b_{low} to b_{high} in the order of the loop, each with its test of a single denominator:

(1) under the flag of the squares, the squares k^2 for $\lceil \sqrt{b_{\text{low}}} \rceil \leq k \leq \lfloor \sqrt{b_{\text{high}}} \rfloor$, with the class test;

(2) otherwise, when the set-up has returned the records and the divisors of Definition 8.4, the divisors times squares: for each divisor s in turn, the sk^2 of the range in increasing order of k, with the class test and the valuation test, which b fails when the record of a prime excludes it (find_points.c:3944-3953);

(3) otherwise, under the flag of the checked loop, the words, with the forbidden divisors, the switch of the Jacobi test, its tables when the test is on and they exist, and c_d: word by word, the $64w + j$ of the range for the set bits j, lowest first, of den_bits and-ed with word w mod p of each forbidden divisor p with words, tested by the tables, then by the forbidden divisors with masks, and, with the Jacobi test on and no tables, by the binary algorithm (find_points.c:4013-4041);

(4) otherwise every denominator, $b_{\text{low}}, \ldots, b_{\text{high}}$, with the class test.

The class test is that the pattern of the class of b mod 64 is not 0. The lists of the denominators of the shapes (`Ratpoints.Sieve.Shape.dens`) and the choice of the test of a single denominator (`Ratpoints.Sieve.Shape.keep`) are definitions of the proofs, the reference of the loops over the blocks, which test and sift each denominator as they reach it, as `run_block` does; the tests themselves model the C code. The denominators of the search (`Ratpoints.searchDens`) are those of the divisors times squares when d is odd, $|c_d| \neq 1$ and the set-up for b_{high} returns divisors, and $b_{\text{low}}, \dots, b_{\text{high}}$ otherwise.

The loops over the blocks (`Ratpoints.Sieve.Shape.blockLoop`, Chapter 6) visit the denominators of the list of the shape in its order (`Ratpoints.Sieve.Shape.flatMap_blockDens`). In the shape of the words they reduce the number of the first word of a block modulo each forbidden divisor, with its reciprocal while that number is at most mulModLimit and by division otherwise, and step on from there (`find_points.c:3981-4006`).

Lemma 8.11 (The shapes keep every point)**.** *Let $1 \leq b_{\text{low}}$. The denominators of every shape lie in the range: the squares are its squares in increasing order; for positive divisors, the divisors times squares are its sk^2 with s a divisor, without duplicates for distinct squarefree divisors; for forbidden divisors with positive primes and the words of `Ratpoints.Sieve.forbiddenRow`, the words give, in increasing order, the denominators of the range with their bit in* den_bits *set that none of these primes divides. Let a and y be integers and $1 \leq b \leq H$ with $\gcd(a, b) = 1$ and $y^2 = F(a, b)$. For odd d, b passes the valuation test with the records of Definition 8.4; for even d and admissible forbidden divisors for $b_{\text{high}} = H$, the bit of b* mod 64 *in* den_bits *is set, no forbidden divisor with words divides b, and b passes the tests of the words, with the tables for H when the Jacobi test is on. When bit j of* den_bits *says whether the pattern of class j is not* 0*, a denominator of a shape that passes its test has a nonzero pattern.*

Proof. The squares come from the integer square roots, $\lceil \sqrt{n} \rceil \leq k$ exactly when $n \leq k^2$; sk^2 determines the squarefree s and k (`Nat.eq_of_mul_sq_eq_mul_sq`); and bit t of word w mod p of a forbidden divisor is clear exactly when $p \mid 64w + t$ (`Ratpoints.Sieve.getLsbD_getElem!_forbiddenRow`). The tests pass b by Lemmas 8.2, 8.5, 8.7 and 8.9. □

8.6 The order of the denominators

The shapes visit their denominators in increasing order, except the divisors times squares, which go divisor by divisor. The denominators that the other shapes and the tests leave out carry no point (Lemma 8.11), so among those that carry points only the divisor shape changes the order. For $y^2 = 1 + 2x$ and $H = 20$, with the divisors 1 and 2, the loop visits $1, 4, 9, 16$ and then $2, 8, 18$ (4 and 16 fail the valuation test of the prime 2), so that `ratpoints` prints $(-4 : 3 : 9)$ before $(-1 : 0 : 2)$.

Lemma 8.12 (The denominators of the search)**.** *For $b_{\text{high}} \geq 1$ the denominators of the search of c of its degree from 1 to b_{high} are* searchDenominators(maxDivisors, c, b_{high}) *(Definition 2.9), and those from $b_{\text{low}} \geq 1$ are those from 1 that are at least b_{low}. For $1 \leq b_{\text{low}}$ and $1 \leq b_{\text{high}} \leq H$, the points that the brute-force reference finds over $(1, 0)$, the points at infinity, followed by those over the denominators of the search with the numerators $-H, \dots, H$, are a permutation of the points of* bruteForce(c, H) *with $z = 0$ or $b_{\text{low}} \leq z \leq b_{\text{high}}$.*

Proof. The specification takes the divisors times squares under the same condition, with the divisors binaryDivisors (Definition 8.4); from 1 the divisor shape lists the $sk^2 \leq b_{\text{high}}$ divisor by divisor (`Ratpoints.Sieve.squares1Dens_one_eq_filter`). The divisor shape holds every denominator of a point in the range once (Lemmas 8.5 and 8.11), so its points are a permutation of those of the whole range (`Ratpoints.perm_flatMap_squares1Dens`). □

With the block lemma (Theorem 3.13), these lemmas give the list that a search delivers (Chapter 3), on which the order theorem (Theorem 3.5) and the membership theorem (Theorem 3.3) rest.

8.7 Where the model differs from the C code

One function serves for `valuation` and `valuation1`, one for `jacobi` and `jacobi1`; the square shapes take exact integer square roots, which agree with the C code's corrected square roots in `double` on the arguments it passes; and the symbol $J(c_d \mid p) = -1$ of the further primes is read off the table of the squares modulo p. The masks of the forbidden valuations have no bound $m < 63$, which never binds for a `long` b_{high} (`Ratpoints.Sieve.lt_forty_of_pow_le`), so that the model is sound for every height. The other deviations, natural numbers for the C integer types and no failure for lack of memory, are those of the whole model (Chapter 11).

Chapter 9

The sieve of one denominator

For each denominator b that the loop over the denominators visits and that passes the tests of Chapter 8, the search looks for the points $(a : y : b)$ with $\gcd(a, b) = 1$, $|a| \leq H$ and a/b in a search interval. `ratpoints` does this with a sieve. The numerators become the bits of bit arrays of 64-bit words, packed by the stride of the 2-adic class of b. The rows of the moduli for b, which say for which numerators the form can be a square modulo the modulus (Chapter 7), are and-ed into them, in a first stage over every bit array and a second stage over the bit arrays that the first one leaves nonzero. Each numerator whose bit survives is tested for common factors with b, then against the primes of a third stage, and then checked exactly; its points go to the callback (Chapter 10).

This chapter describes the model of this sieve, `Ratpoints.Sieve.sift` and its parts, and its contract (Definition 9.11): from a delivery whose flag of the stop is clear, the sieve of b delivers, in increasing order, what the exact check finds for the numerators of the search intervals that pass the 2-adic word of the class of b and survive the rows of all the moduli. The block lemma (Theorem 3.13), and through it the main theorems, rests on this contract: as every row and the 2-adic word let the numerator of every point through (Chapters 7 and 8), the sieve delivers the points of the brute-force reference over b with their numerators in the ranges. Theorem 9.13 restates the contract, with the exact check on, as the check of a plain sieve; no main theorem needs it. The moduli, the cut points of the stages and the tables of rows come from Chapter 6, the search intervals and the ranges of their numerators from Chapter 5.

9.1 The sieve in the C code and in the model

In the C code the sieve of a denominator is `sift` (`find_points.c:3442-3617`), which cuts the bits of each search interval into pieces for `_ratpoints_sift0` (`sift.c:498-1305`), and each numerator that survives goes through `accepted` (`sift.c:438-446`) to the exact check (`find_points.c:5250-5338`). The model has a component for each step, in the sections below, with the data and the loops of the default build.

A C `long` is an integer or a natural number, with a lemma that it stays in range, except in the test for common factors and the bits of an interval, which compute on 64-bit words. The contracts use Lean core and `Std` only; that of a component which finds points says that, from a delivery whose flag of the stop is clear, it returns the reference of the delivery (Definition 2.13) of the list of the points it finds.

Definition 9.1 (The counters of the sieve)**.** The six counters that the sieve keeps for the correction of the moduli (Chapter 6), 64-bit words that wrap (rp-private.h:579-581): the words that the first stage sweeps, the bit arrays that survive it, the numerators extracted, those prime to the denominator, those that pass the third stage, and the calls of the sieve.

9.2 Bit arrays of words

A bit array of the C code is a vector of words, with macros for its operations in each build (rp-private.h:113-300). In the model it is rbaPack consecutive `UInt64` words of an array, one definition of each operation serving every build; the proofs read the bits through `BitVec.getLsbD`, whose lemmas Lean core has, and need no `bv_decide`.

Definition 9.2 (Bit arrays of words)**.** A bit array has rbaPack $= 4$ words, rbaLength $= 256$ bits, as in the default build (rp-private.h:172-172, rp-private.h:174-175); bit t of bit array j is bit $t \bmod 64$ of word rbaPack$\cdot j+\lfloor t/64 \rfloor$. The operations: bit array j of an array of words and-ed with bit array x of a row, word by word (`AND`, rp-private.h:176-176); the lowest s bits of bit array 0, or the highest s bits of bit array j, cleared for $s <$ rbaLength (`MASKL`, `MASKU`, rp-private.h:206-213, rp-private.h:214-221), so that the low mask keeps bit t of word y exactly when $s \le 64y+t$ and the high mask keeps it in a word $y <$ rbaPack $(j+1)$ exactly when $64y+t+s <$ rbaLength $(j+1)$; and whether bit array j is zero (`TEST`, `TESTZ`, rp-private.h:179-202). The lowest set bit of a word $x \neq 0$ comes from the loop of `RP_CTZL` for a compiler without the builtin, which shifts x right while its lowest bit is clear (rp-private.h:429-444); its contract: the count is the number of trailing zeros of x.

The proofs use of the width only rbaPack > 0 and, for the packing, rbaLength $= 2^{\text{rbaShift}}$ with rbaShift < 32, so they hold for every build (Chapter 11).

9.3 Reductions by a reciprocal

`ratpoints` reduces modulo p by a multiplication with the reciprocal of p wherever that is exact: the row indices of the first two stages, the residues of a denominator for its stride and for the third stage, the offsets of the rows, the third stage, the first word of the forbidden divisors of a block and the Jacobi test (Chapter 8). The reciprocal is a field of the sieve entry (Chapter 6).

Definition 9.3 (The reductions by a reciprocal)**.** The reciprocal of p is magic$(p) = \lfloor (2^{64}-1)/p \rfloor + 1$. The high half mulHi$(a,b)$ of the 128-bit product of two words, which the C code takes from gcc's `__uint128_t` (rp-private.h:404-408, rp-private.h:409-410), comes in the model from the four products of their 32-bit halves, no partial sum leaving a word. The remainder by the reciprocal m, `RP_MULMOD`, is mulHi$(mu \bmod 2^{64}, p)$, and the quotient, `RP_MULDIV`, is mulHi(u,m), the arguments cast to words as the macros cast them; the builds that compile no 128-bit product (no such type, or `RP_MULMOD_DIVIDE` or `RP_NO_INT128` set) divide instead (rp-private.h:413-414, rp-private.h:415-416), with the same results wherever the multiplication is exact. The remainder of an integer a, `mod_mul`, is the remainder r of $|a|$, and $p-r$ for negative a with $r \neq 0$ (sift.c:243-247).

Lemma 9.4 (The reductions are exact)**.** *For all words,* mulHi$(a,b) = \lfloor ab/2^{64} \rfloor$. *For* $0 < p$ *and* $up < 2^{64}$ *the remainder with the reciprocal of* p *is* $u \bmod p$*; for* $1 < p$ *and* $up < 2^{64}$ *the quotient is* $\lfloor u/p \rfloor$*; for* $0 < p$ *and* $|a|\,p < 2^{64}$*,* `mod_mul` *gives the Euclidean remainder* $a \bmod p$.

Proof. With $a = 2^{32}a_1 + a_0$ and $b = 2^{32}b_1 + b_0$, the middle sum of the halves is below $3 \cdot 2^{32}$, and its high half is the carry. With magic$(p)\,p = 2^{64} + \delta$, $0 \le \delta < p$, and $u = kp + s$, $s < p$, we have magic$(p)\,u = k\,2^{64} + (s\,2^{64} + u\delta)/p$, the second part an integer below 2^{64} as $u\delta < 2^{64}$: the high half of the product is k, and its low half times p has the high half s. The reciprocal 2^{64} of 1 reads as 0 on a word, as in the C code, which spoils the quotient but not the remainder. □

The C code reduces so only values below 2^{32} modulo moduli below 2^{10}, which the contracts of the callers prove; it divides larger values with its helper `mod` (`Ratpoints.Sieve.modLong`, the Euclidean remainder) or, in the third stage and its set-up, with the operator `%`.

9.4 The packing of the numerators

The 2-adic information gives for each class of the denominators modulo 64 the word w of the numerators modulo 64 that a point can have (Chapter 8). In most classes the set bits of w lie in one residue class $a_0 \bmod 2^k$, and `ratpoints` sieves only that class: bit n of the sieve of b stands for the numerator $a_0 + 2^k n$, and bit t of the bit array with the *word number* i is the bit $n = \mathrm{rbaLength}\,i + t$. Each modulus p then reads the row of $b\,2^{-k} \bmod p$, at the bit array $(i + \mathrm{offset}) \bmod p$, with an offset computed once per run for each packing and each modulus.

Definition 9.5 (The packing)**.** A class (`rp-private.h:494-495`) has a pattern, a stride exponent k, a residue a_0 and the position of its row of offsets. The class of a word w of the 2-adic information (`find_points.c:1064-1081`): for $w = 0$ the stride 1 and the pattern 0; otherwise, for the lowest set bit a of w, the largest $k \le 6$ below numStrides such that every set bit of w is $\equiv a \bmod 2^k$, $a_0 = a \bmod 2^k$, and the pattern whose bit t is bit $(a_0 + 2^k t) \bmod 64$ of w (`find_points.c:1082-1094`); with the classes the word `den_bits` of the classes whose word is not 0. The offsets (`find_points.c:1142-1158`): for each entry of an odd modulus p the inverses $\mathrm{dinv}[k] = 2^{-k} \bmod p$ and $(2^k\,\mathrm{rbaLength})^{-1} \bmod p$, by halvings, and for each packing (k, a_0) of a class with a nonzero pattern a row with the offset $a_0\,(2^k\,\mathrm{rbaLength})^{-1} \bmod p$ plus the bias of p, the least multiple of p from $\mathrm{rowBias} = 2^{31}$ on (`find_points.c:1164-1182`). The residues $b\,\mathrm{dinv}[k] \bmod p$ of a denominator for the moduli of the first two stages, in three branches by the size of b (`find_points.c:3639-3679`). The bits $\lceil(\mathrm{low} - a_0)/2^k\rceil$ to $\lfloor(\mathrm{high} - a_0)/2^k\rfloor$ of an interval of numerators, by arithmetic shifts of signed 64-bit words, in a plain form and in a form that splits each end into its multiple of 2^k and its rest, which cannot overflow (`find_points.c:3355-3370`).

Lemma 9.6 (What the packing computes)**.**

(1) *The class of* w packs w*: it is* well formed *(*$k \le 6$*,* $k <$ numStrides*,* $a_0 < 2^k$*), every set bit of* w *is* $\equiv a_0 \bmod 2^k$*, and bit* t *of its pattern is bit* $(a_0 + 2^k t) \bmod 64$ *of* w*; bit* j *of* `den_bits` *is set exactly when* $j < 64$ *and the word of* j *is not* 0*.*

(2) *For entries of odd moduli* $1 < p \le$ maxPrimeEven *with their reciprocals, the inverse of* rbaLength *and a bias in* $p\mathbb{Z} \cap [\mathrm{rowBias}, \mathrm{rowBias} + p)$*, well-formed classes and an array large enough, every class with a nonzero pattern gets a row whose offset* o *for each entry is* packed for *it:* $\mathrm{rowBias} \le o < \mathrm{rowBias} + 2p$*,* $o\,2^k\,\mathrm{rbaLength} \equiv a_0 \bmod p$ *and* $\mathrm{dinv}[k]\,2^k \equiv 1 \bmod p$ *with* $\mathrm{dinv}[k] < p$*; nothing else changes.*

(3) *For entries with* $0 < p \le$ maxPrimeEven*, their reciprocals and* $\mathrm{dinv}[k] < p$*, the residues of a denominator are* $b\,\mathrm{dinv}[k] \bmod p$*.*

(4) *For a well-formed class the split form gives the bits* $-\lfloor(a_0 - \mathrm{low})/2^k\rfloor$ *and* $\lfloor(\mathrm{high} - a_0)/2^k\rfloor$*, and so does the plain form when* $a_0 - \mathrm{low} < 2^{63}$ *and* $\mathrm{high} - a_0 \ge -2^{63}$*.*

(5) *If the tables of inverses of the factors of an entry invert their units and* $vu \equiv 1 \bmod p$*, the row of* ub *admits* ua *exactly when the row of* b *admits* a*; so the closed form of the row of* b' *with* $2^k b' \equiv b$ *(Chapter 7), read at an offset* o *with* $o\,2^k \,\mathrm{rbaLength} \equiv a_0$*, has at bit* j *the admission of* $a_0 + 2^k j$ *by the row of* b*.*

Proof. (1) The mask of the multiples of 2^k is empty beyond $k = 6$, which ends the loop over the strides; the scatter puts bit a at $(a - a_0)/2^k$, and the doubling repeats the pattern. (2) A halving multiplies by 2^{-1} modulo p; the rows occupy disjoint ranges of the array, one for each key $2^k + a_0$; the values reduced are below $64\,p$, so the reductions are exact (Lemma 9.4), and so in (3). (4) The arithmetic shift by k is the floor of the quotient by 2^k, the and with $2^k - 1$ the Euclidean remainder. (5) The table of a prime is read at $a\,b^{-1}$, that of a prime power at $a\,b^{-1}$ or $b\,a^{-1}$, which u does not change. Read at the offset o, bit j is the admission of $o\,\mathrm{rbaLength} + j$ by the row of b', and multiplied by the unit 2^k this index becomes $a_0 + 2^k j$ modulo p. □

So the packed sieve keeps exactly the numerators of the class a_0 mod 2^k that a sieve without the packing keeps, in the same order, and the pattern lets the bit n through exactly when w lets $a_0 + 2^k n$ through (`Ratpoints.Sieve.NumClass.Packs.classBit_bits`); a numerator outside the class is one that w rejects. The third stage is not packed.

9.5 The first stage

The first stage and-s the rows of the first s_1 moduli into every bit array of a piece, the bit arrays with the word numbers $w_0, \ldots, w_0 + n - 1$, from the pattern of the class. The C code has two arms of it, which the preprocessor chooses by the number `RATPOINTS_CHUNK` of registers: for a chunk from 2 to 16 the chunked arm, which and-s the rows of all the moduli into a chunk of bit arrays in registers; otherwise the arm that and-s one row at a time into the whole piece. The model has both and chooses between them by the same test.

Definition 9.7 (The first stage)**.** The chunk is 16, as in the default build (`rp-private.h:222-224`). A sieve spec (`rp-private.h:378-379`) holds a modulus p, its row for the denominator at hand ($p + \mathrm{chunk} - 1$ bit arrays), the offset of the row, and a start and an end for the chunked arm. It *fits* when $0 < p \le \mathrm{maxPrimeEven}$, $\mathrm{rowBias} \le \mathrm{offset} < \mathrm{rowBias} + 2p$, its end is p, and bit array $x < p + \mathrm{chunk} - 1$ of its row is bit array x mod p. The test `small` says whether the word numbers of the piece lie between $-\mathrm{rowBias}$ and $\mathrm{rowBias} - 2\,\mathrm{maxPrimeEven}$; then the sums of a word number and an offset are reduced with the reciprocal, and otherwise by `mod`. The arm one row at a time (`sift.c:943-1022`) sets every bit array to the class pattern and then, for each modulus, and-s bit array j with bit array $(w_0 + \mathrm{offset} + j) \bmod p$ of the row. The chunked arm sets the start of each row to $(w_0 + \mathrm{offset}) \bmod p$ (`sift.c:596-606`); with no modulus in the stage ($s_1 = 0$) it writes the class pattern into every bit array, and otherwise it finds where the whole chunks end (`sift.c:608-620`). A whole chunk is loaded into registers as the class pattern and-ed with the first row from its start on, and-ed with the other rows and stored, each start moving on by the chunk modulo p; the $t <$ chunk bit arrays after the whole chunks go in legs of 8, 4, 2 and 1 registers, one for each set bit of t (`sift.c:915-940`). The contract of each arm (of the chunked one under chunk ≤ 16): for specs that fit, the reciprocals of their moduli and the bounds of `small`, word $y < n\,\mathrm{rbaPack}$ of the piece is the class pattern and-ed, for each spec, with word y mod rbaPack of bit array $(w_0 + \mathrm{offset} + \lfloor y/\mathrm{rbaPack} \rfloor) \bmod p$ of its row.

The chunked arm reads what the other arm reads: from the start $s = (w_0 + \mathrm{offset} + c) \bmod p$ of the chunk at the bit array c, bit array $s + j < p + \mathrm{chunk} - 1$ of the row is bit array $(w_0 + \mathrm{offset} + c + j) \bmod p$. That the legs store each bit array of a tail once is decided for each $t < 16$;

the test that chooses the arm gives chunk ≤ 16. The rows that the run reads fit (Chapter 7). The masks of the ends of a piece come after the first stage, as in the C code.

9.6 The second stage and the extraction

Definition 9.8 (The second stage and the extraction)**.** The second stage on a copy of the bit array with the word number i: for each of its moduli in turn, while the copy is nonzero, the copy is and-ed with bit array $(i + \text{offset}) \bmod p$ of the row, the index reduced as in the first stage (`sift.c:1175-1202`). The walk through a word x whose bit t stands for $a_0 + st$: while $x \neq 0$, the numerator of the lowest set bit goes to the test of a candidate and, if it passes, to the exact check (Definition 9.9), the walk returning if the callback has stopped the search; then x becomes $x \,\&\, (x-1)$ (`sift.c:262-264`, `sift.c:1227-1255`). A bit array (`sift.c:1213-1226`, `sift.c:1257-1264`): the second stage on the copy and, if the copy is still nonzero, the walk through each of its words, bit t of word κ standing for $a_0 + 2^k(\text{rbaLength}\, i + 64\kappa + t)$. The scan of a piece (`sift.c:1061-1082`): a sentinel of ones just past the piece; the zero bit arrays are stepped over, the scan stops at the sentinel, and each other bit array is sifted as above. The contracts, for specs that fit, the reciprocals of their moduli and the bounds of `small`, from a delivery whose flag is clear: the copy after the second stage is the copy and-ed with the rows of all the moduli of the stage; the walk, and a bit array in either arm, deliver in increasing order the points of the numerators of the set bits (of x, and of the words and-ed with the rows of the second stage) that pass the test of a candidate; and if, for a test φ of the bits, bit t of word y of the piece and-ed with the rows of the second stage is $t < 64 \wedge \varphi(\text{rbaLength}\, w_0 + 64y + t)$, the scan delivers in the same way the points of the numerators $a_0 + 2^k j$ of the bits j of the piece that pass φ and the test of a candidate.

Stopping the second stage early loses nothing, since a zero bit array stays zero. A build with `USE_LONG_IN_PHASE_2` and-s one word at a time (`Ratpoints.Sieve.secondStageWord`); the model has both arms, chosen by the constant `Ratpoints.useLongInPhase2`, false as in the default build, and proves that they deliver the same points (`Ratpoints.Sieve.siftBitArray_spec`). The walk ends since clearing the lowest set bit makes the word smaller, the scan since the sentinel is not zero.

9.7 The test of a candidate and the exact check

Definition 9.9 (The test of a candidate and the exact check)**.** The test for common factors on signed 64-bit words (`sift.c:318-327`): for $m = 0$, whether $n = 1$; otherwise m replaced by $-m$ when it is negative, false when m and n are both even, and else the binary algorithm on their odd parts: while $n \neq m$, with $d = m - n$ and the mask $d \gg 63$, n becomes $n + (d \,\&\, \text{mask})$ and m becomes $(d \oplus \text{mask}) - \text{mask}$ shifted right by its trailing zeros; the result is whether m ends as 1 (`sift.c:328-342`). What the third stage needs of the entry of a prime p for b (`sift.c:414-428`): the entry of its table of inverses at $b \bmod p$, or 0 when $p \mid b$; the bias pH when $2pH < 2^{32}$, else 0; the reciprocal and the table of p; for the entries of the stage, a thunk made once per denominator (`sift.c:405-413`). The third stage for a: each prime whose inverse is not 0 rejects a when its table says that f is not a square modulo p at the residue of $a\, b^{-1}$, reduced with the reciprocal after adding the bias if there is one, and by division otherwise (`sift.c:291-301`). The test of a candidate: a is prime to b and passes the third stage (`sift.c:438-446`). The exact check (`find_points.c:5250-5338`): unless noCheck, for the value v of Horner's rule at a on the products $c_k b^{d-k}$ (a thunk made once per denominator), times b for odd d, so that $v = F_c(a, b)$

(Definition 2.1), and $y = \lfloor\sqrt{v}\rfloor$: when $v = y^2$, the call with $(a : y : b)$ and then, unless that call stopped the search, $y = 0$ or noY, the call with $(a : -y : b)$; under noCheck one call with $(a : 0 : b)$; the points mapped back to the caller's curve after a reversal.

Lemma 9.10 (The test of a candidate)**.** *For words $m > -2^{63}$ and $n > 0$ the test for common factors gives whether* $\gcd(m, n) = 1$*. A* prime entry *is a sieve entry whose single factor is a table of the kind of a prime, of a modulus $1 < p \leq 2^{32}$ (p need not be prime), whose table of inverses holds a value in $[1, p-1]$ at each nonzero residue below p (that these are inverses is not needed), and whose modulus and reciprocal are those of p. For prime entries,* $\gcd(a, b) = 1$ *and $|a| \leq H$, the third stage lets a through exactly when the rows of b in all the entries admit a. For $0 < b < 2^{63}$ and $|a| < 2^{63}$ the test of a candidate is* $\gcd(a, b) = 1$ *followed by the third stage. So for prime entries in the third stage with their reciprocals, $0 < b < 2^{63}$ and $|a| \leq H < 2^{63}$, the points that the extraction delivers for a are those of the check with the flags (`Ratpoints.Sieve.checkNumWith`, the points of the exact check if* $\gcd(a, b) = 1$*) when the rows of the third stage admit a, and none otherwise.*

Proof. For positive words, $d = m - n$ is a word and the step gives $\min(m, n)$ and the odd part of $|m - n|$; the invariant keeps both values odd and positive, so that no step leaves the range of a `long` and the arithmetic of the 64-bit words is that of the C code; their gcd stays the same and their sum decreases, so that the loop ends at $m = n =$ gcd. For $m \neq 0$, $-m$ is a word, and removing the powers of 2 changes whether the gcd is 1 only when both are even. In the third stage an entry whose modulus p does not divide b is not skipped, since its table of inverses is not 0 there; the biased value lies in $[H, 2pH - H]$, both reductions are exact (Lemma 9.4), and the residue read is where the row of b reads the table (Chapter 6). An entry with $p \mid b$ is skipped, and its row admits a, since $p \nmid a$ by the gcd. □

The check with the flags is the check of the reference (`Ratpoints.Sieve.checkNum`, what the brute-force reference finds for one numerator, mapped back after a reversal), without the points with $y < 0$ under noY, and the one survivor $(a, 0, b)$ under noCheck (Chapter 10). The third stage of the run holds only primes of the table, also after a correction of the moduli (Chapter 6), and these are prime entries.

9.8 One denominator

Definition 9.11 (The sieve of one denominator)**.** One piece (`sift.c:498-531`): the test `small`, a fresh array with room for the sentinel, the first stage with the first s_1 moduli in the arm that $1 <$ chunk ≤ 16 chooses, the masks of the ends (`sift.c:1030-1041`) and the scan with the moduli of the second stage. The bits from low to high of an interval (`find_points.c:3563-3612`): the word numbers from ⌊low/rbaLength⌋ to ⌊high/rbaLength⌋ in pieces of at most A bit arrays, the first and the last masked. One denominator (`find_points.c:3442-3488`): for each search interval the bits of its numerators within the height bound (Chapter 5); at the first interval that the height bound does not rule out (also one without numerators) the sieve specs (Definition 7.8), the modulus, the offset of the row of the class and the row of the residue of b for the stride of each entry of the first two stages, the missing rows built into their tables (Chapter 7); then the bits of the interval in pieces. Each returns when the callback stops the search. The contract of one denominator: for $A \geq 1$, $0 < b < 2^{63}$, $H < 2^{63}$, prime entries in the third stage, the reciprocals of all the entries, the residues $b\,\mathrm{dinv}[k] \bmod p$ of the entries of the first two stages, whose tables of inverses invert, tables of rows that fit them, and a delivery whose flag is clear: for every word w that the class packs, a row of offsets packed for the class, and the ranges of the numerators that the bounds of the search intervals give at b, the delivery is that of the points of

the check with the flags of the numerators $-H, \ldots, H$ that lie in a range, pass w and survive the entries of all three stages, in increasing order; when the callback does not stop the search, the counters of the sieve grow by a closed form that depends neither on the callback and the flags nor on the tables of rows; and the tables of rows afterwards fit the entries again.

The contracts compose from the bottom: the walk gives that of a bit array, the bit arrays that of the scan, and the scan with either arm of the first stage and the masks that of a piece; the pieces of an interval lie end to end. Lemma 9.6 turns the bits into numerators and the rows for the stride into the rows of b, and Lemma 9.10 the test of a candidate into the rows of the third stage. The counters (Definition 9.1), which feed the correction of the moduli (Chapter 6), are counted before the exact check, as in the C code.

9.9 The stages compute the plain sieve

The plain sieve, a sieve in its simplest form, models no C code.

Definition 9.12 (The plain sieve)**.** For a denominator b, the height bound H and a list of sieve entries (Chapter 6): an array of $2H + 1$ Booleans, entry i for the numerator $a = i - H$, all true at the start, each and-ed in turn with the row of b in every entry at a. Its contract: entry i is the survivor test `Ratpoints.Sieve.survives` at $i - H$, the conjunction of the rows. The check of such an array takes the numerators whose entry is true, in increasing order, and tests and checks each as the brute-force reference does. Its contract: if entry i is $\varphi(i - H)$ for a test φ of the numerators, the result is what the brute-force reference (Definition 2.14) finds over b and the numerators $-H, \ldots, H$ that pass φ.

Theorem 9.13 (The stages compute the plain sieve)**.** *Let $A \geq 1$, $0 < b < 2^{63}$ and $H < 2^{63}$; let the class pack a word w and have a row of offsets packed for it, the entries of the first two stages have tables of inverses that invert and tables of rows that fit them, the entries of the third stage be prime entries, and the bounds of the search intervals give the ranges of the numerators at b. Then, with the exact check on, from a delivery whose flag is clear,* `sift` *on b, with the reciprocals of the entries and the residues of b for the stride, delivers the points that the check of the plain sieve with the entries of all three stages finds, restricted to the numerators that lie in a range and pass w, mapped back after a reversal, and without those with $y < 0$ under* noY*.*

Proof. The check of the plain sieve gives what the brute-force reference finds for the numerators that survive all the entries (Definition 9.12), and the sieve delivers the points of the check with the flags of the numerators in the ranges that pass w and survive all the entries (Definition 9.11). Without noCheck this check is the check of the reference, mapped back and filtered by noY, and filtering the numerators filters the points of the reference by their numerators. A numerator survives a concatenation of lists of entries exactly when it survives each list (`Ratpoints.Sieve.survives_append`), so that the entries of the three stages together give the survivor test of all of them. □

9.10 Where the model differs from the C code

The `Deviation:` paragraphs of the components record each difference, and Chapter 11 sorts them by kind. The high half of a 128-bit product comes from the 32-bit halves, and the lowest set bit from the loop for a compiler without the builtin. The test for common factors and the bits of an interval compute on 64-bit words in two's complement with the arithmetic shift, which the C standard leaves to the implementation and gcc defines so; the words wrap where a `long` would

overflow only in the plain form of the bits, for an end within a_0 of -2^{63}; there both the C code and the model use the split form. The comparison $2pH < 2^{32}$ is made on natural numbers, for one of `double`s, with the same outcome for every $H < 2^{63}$. The flag of the stop is tested after each walk, bit array, piece and interval, where the C code returns from within the walk.

Chapter 10

Delivery and the program

The library function `find_points` of `ratpoints` returns no list of points: it hands each point that it finds to a function of its caller, the callback, which may count it, print it or stop the search. This chapter describes how the model does the same, with the callback, its weights and its stop, the options of the output `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y`, what a search writes back and the report of the library under `RATPOINTS_VERBOSE`; and the program `ratpoints-model`, which reads the command line of `ratpoints` and prints what `ratpoints` prints, but for its banner. The reference of the delivery, with which the statements compare a search, belongs to the specification (Definition 2.13); the theorems on what a search delivers and what the program prints are in Chapter 3; the components of the sieve that pass the delivery on are those of Chapter 9.

10.1 The callback and the delivery

The header of the library declares the callback of `find_points` and `find_points_work` (`ratpoints.h:436-437`, `ratpoints.h:441-442`) as `int proc(long, long, const mpz_t, void*, int*)`; the manual names its parameters `x`, `z`, `y`, `info` and `quit`. The library adds the value that each call returns to a count `total`, which it returns, or one of the negative codes of an error (`ratpoints.h:427-430`); a call stops the search by setting `*quit`, which the library clears before the search and tests after the steps that may have made calls (Section 10.2), so that it is clear at every call. Every refusal of the input comes before the first call; only `RATPOINTS_NO_MEMORY` can come after calls, when an allocation fails, and the model has no allocation that can fail. The callback of the model and the delivery that a search threads through its components are those of Definition 2.13; the delivery stands for the pointer `info`, for the counts `total` of the functions of the search as one running count, and for the flags of the stop of the points at infinity and of the sieve (`find_points.c:4570-4571`, `find_points.c:3129-3129`).

Definition 10.1 (The flags of the check and the calls of a search)**.** The flags of the check, which decide what the exact check and the points at infinity deliver: the bits `RATPOINTS_REVERSED` (`ratpoints.h:414-414`), `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the word of flags, as three Booleans, clear by default. One call (Definition 2.13) with a point $(a : y : b)$ of the polynomial searched, mapped back to the caller's curve when the flag of the reversal is set: with $(b : y : a)$ for $a \geq 0$ and $(-b : y : -a)$ for $a < 0$, and with $(a : y : b)$ otherwise, the three branches that the exact check of the C code writes at each of its calls. The callback of the model that appends each point (x, y, z) to an array, with the weight 1, and never stops the search; with it the data of a search is the list of the points that it delivers, in their order. And a text printed through

the print channel of a search: the data as the channel leaves it, the count and the flag as they are (Section 10.5).

Apart from the flags of the check, which stand for bits of `args->flags`, these definitions model no C code of their own; the appending callback is that of findPoints, of which the statements speak (Definition 3.1).

Lemma 10.2 (The delivery of a list)**.** *The delivery of $L_1 +\!\!+ L_2$ is the delivery of L_2 after that of L_1. A delivery whose flag is set stays as it is, whatever the list; so if the delivery of a prefix M of L sets the flag, the delivery of L is that of M, and a test of the flag after the delivery of L_1, followed by the delivery of L_2 when the flag is clear, is the delivery of $L_1 +\!\!+ L_2$. With the appending callback, from a delivery whose flag is clear, the array grows by L and the count by the length of L. For every list L there is a prefix M of L whose delivery is that of L, such that the flag is clear before each point of M, so that each of them gets its call, and the delivery of M sets the flag unless $M = L$.*

Proof. By induction on the list, from the fold, in Lean core only, so that the contracts of the sieve, which are proved without Mathlib, can state what they deliver. □

10.2 The stops

In the C code on one thread only a call of the callback sets the flag (with threads a worker also sets it when the search has been stopped, `find_points.c:3610-3610`), and each function tests it after each step that may have made calls, returning its count at once when it is set: `_ratpoints_sift0` after the check of a numerator (`sift.c:1250-1251`, `sift.c:1287-1288`, and `sift.c:1168-1169` in the arm of `USE_LONG_IN_PHASE_2`); `sift` after each call of `_ratpoints_sift0` (`find_points.c:3611-3611`); `run_block` after each denominator, in each shape of the loop (`find_points.c:3928-3928`, `find_points.c:3957-3957`, `find_points.c:4044-4044`, `find_points.c:4068-4068`); the loop over the blocks after each block (`find_points.c:5158-5158`); and `find_points_work_1` after the points at infinity, before it sieves (`find_points.c:5100-5103`). So the calls made are those of the search that no call stops, up to and including the first call that sets the flag, and the count is the sum of their weights. With threads the calling thread of the C code delivers the points of the blocks in their order; that blocks sifted apart and delivered in this way make the calls of the run on one thread is the thread lemma (Theorem 3.15), and that the pool of `ratpoints` meets its hypotheses is read off the C code, not proved.

The model tests the flag at the same places, with one difference of form: where the C code returns from `_ratpoints_sift0` from within the walk through the set bits of a word, the walk of the model returns, and its callers test the flag after each walk, and the scan of a piece after each bit array, and return in turn; this makes no call that the C code does not make and skips none. Every component of the sieve from the walk up to the run over the blocks takes the callback, the flags and a delivery and hands them on (Chapter 9); its contract has the precondition that the flag is clear and concludes that the delivery after it is the delivery (Definition 2.13) of a list, the points of the component in their order. Each loop that may return early keeps an invariant with a case for the return at a stop, closed by Lemma 10.2 (the invariants `Ratpoints.Sieve.WalkDeliveryInvariant` of the walk and `Ratpoints.Sieve.SieveRunLoopInvariant` of the run over the blocks). So the search with any callback delivers the list of the search with the appending callback, up to and including the first call that stops it: the protocol of the delivery (Theorem 3.7).

10.3 The options of the output

Two bits of the flags of the caller change which points the library delivers (`ratpoints.h:403-404`): under `RATPOINTS_NO_Y` only the point with $y \geq 0$ of a pair $(x : \pm y : z)$ goes to the callback, and under `RATPOINTS_NO_CHECK` every numerator that passes the sieve goes to it with $y = 0$, without the exact check, whether or not it gives a point. The C code reads them only in the exact check (`find_points.c:5250-5338`), at the points at infinity (`find_points.c:5068-5099`), and `RATPOINTS_NO_CHECK` also in a line of the report; the reversal, the Sturm step, the exits, the tests on the denominators and the sieve run as without them. Conversely, what `RATPOINTS_NO_CHECK` delivers depends on everything that decides which numerators pass the sieve: the moduli, the Jacobi test, the reversal and the region of the Sturm step.

In the model the two options are the fields noCheck and noY of the options of a search, false by default (Definition 2.11), which the search copies, with the flag of the reversal, into the flags of the check (Definition 10.1). What the points at infinity deliver with these flags is the list of Definition 4.9; what the exact check (Chapter 9) delivers for one numerator to a callback that does not stop the search is the following list, a definition of the proofs that holds no C code.

Definition 10.3 (The points of a numerator with the flags)**.** The points that the exact check of the numerator a over the denominator b delivers, for the products w of the coefficients with the powers of b, the factor e (b for odd degree, 1 for even degree) and the flags: under noCheck the survivor $(a, 0, b)$; otherwise, when the value of Horner's rule in a on w, times e, is y^2 with $y \geq 0$, the point (a, y, b) and, for $y \neq 0$, the point $(a, -y, b)$, which is left out under noY, and none when that value is no square; each mapped to the caller's curve when the flag of the reversal is set, (a, y, b) to (b, y, a) for $a \geq 0$ and to $(-b, y, -a)$ for $a < 0$. The points of a numerator with the flags: these when $\gcd(a, b) = 1$, and none otherwise.

Lemma 10.4 (The check and the points at infinity with the flags)**.** *On a delivery whose flag is clear, the exact check of a numerator delivers its list of Definition 10.3, as the points at infinity deliver theirs (Definition 4.9): the second call of a pair, left out after a first call that stops the search, is the part of the delivery that a set flag passes over. Without* noCheck*, the points of a numerator with the flags are those of the check of the reference (`Ratpoints.Sieve.checkNum`), which tests* $\gcd(a, b) = 1$ *and the square as the brute-force reference does, mapped after a reversal, without those with* $y < 0$ *under* noY*; under* noCheck *they are the survivor* $(a, 0, b)$*, mapped after a reversal, when* $\gcd(a, b) = 1$*, and nothing otherwise. With the exact check on and* noY *set, the points of a numerator with* y *replaced by* 0 *form a sublist of its points under* noCheck*, with the same flag of the reversal and either setting of* noY*, and the points at infinity with* y *replaced by* 0 *are those under* noCheck*.*

Proof. Each call is the delivery of a list of one point, and the test of the flag between the two calls of a pair is the delivery of their concatenation (Lemma 10.2). The rest by the cases of the flags: over one numerator the reference finds (a, y, b) and $(a, -y, b)$, or $(a, 0, b)$, or nothing, so that the point with $y \geq 0$, with y replaced by 0, is the survivor, and the map of the reversal keeps y. □

The contracts of the sieve carry these lists upwards: the search delivers the points at infinity with the flags followed by the points with the flags of the numerators that pass the test of the sieve, a test that depends neither on the callback nor on the flags. So the output under `RATPOINTS_NO_CHECK` holds, as a sublist and so in the same order, the survivors $(x, 0, z)$ of the points (x, y, z) that the same search delivers with the exact check on and `RATPOINTS_NO_Y` set, and each of its elements has $y = 0$ and a normalized pair (x, z) of height at most H in the search region (Theorem 3.8).

10.4 What a search gives back

Besides the count, `find_points` writes back what the program reads after the search: three bits of `args->flags` (`ratpoints.h:414-417`), `RATPOINTS_REVERSED`, set by the reversal check, and `RATPOINTS_USE_SQUARES` and `RATPOINTS_USE_SQUARES1`, set for odd degree when the leading coefficient is ± 1 and, otherwise, when the set-up of its divisors succeeds (`find_points.c:4910-4943`), for message 7; the search intervals as the Sturm step leaves them, for message 8; and the numbers of the moduli used in the stages (`ratpoints.h:378-378`), for message 4. The outcome of a search in the model holds exactly these, with the data of the callback and the count (Definition 2.11). The search sets the bits before its exits and before the points at infinity, and it returns after the Sturm step only for even degree, where the two bits of the squares are clear; so the bits and the intervals are written back on every path that does not refuse the input, whatever the callback does. The numbers of the moduli are zeroed on entry and set after the sieve from the moduli of the last block sifted, so that they depend on where the callback stops the search (Chapter 3). Nothing else is written back: the reversed coefficients and the private fields that the choice of the moduli writes are not, and the bits of the input come back as the caller set them (`find_points.c:4606-4606`).

10.5 The report of the library

Under the bit `RATPOINTS_VERBOSE` of the flags (`ratpoints.h:407-407`), which the option `-v` of the program sets, the library prints a report of the search on the standard output, between the points that the callback of the program prints there. The model has no standard output: the search takes a *print channel*, a function that prints a text into the data of the callback, which holds the standard output of the caller (Definition 10.1). The search with a print channel is findPointsVerbose, and findPointsWith takes by default the channel that prints nothing (Definition 3.1).

Definition 10.5 (The report of the library)**.** The texts of the report, each as `printf` prints it from the data that the C code prints, with `%ld`, `%3ld`, and `%.pf` for the doubles and the values of `Float` as glibc prints them, the exact value rounded to p places with ties to even: the arguments of the search as it normalizes them (`find_points.c:4691-4725`); the outcome of the reversal check (`find_points.c:4740-4743`, `find_points.c:4757-4758`, `find_points.c:4771-4774`, `find_points.c:4791-4801`, `find_points.c:4804-4807`); the Sturm step (`find_points.c:4864-4878`); the points at infinity and the tests on the denominators (`find_points.c:4897-4898`, `find_points.c:4915-4918`, `find_points.c:4921-4926`, `find_points.c:4928-4941`, `find_points.c:4946-4947`); the packing of the classes of the numerators (`find_points.c:4969-4985`); the choice of the moduli (`find_points.c:3089-3095`), the moduli of the three stages with the cost of an exact check (`find_points.c:5027-5042`) and the forbidden divisors (`find_points.c:5043-5048`); and the lines before the sieve (`find_points.c:5111-5113`).

Four texts complete the report: the two exits on the way (`Ratpoints.Report.noClasses`, `Ratpoints.Report.noPoints`), the line before the points at infinity (`Ratpoints.Report.atInfinity`) and the forbidden valuations (`Ratpoints.Report.excludedValuations`). Each text is printed where the C code prints it, or once from the result of a check where the C code prints in its course. The report is compared byte for byte with that of the C program (Chapter 11) and not proved: with the print channel that prints nothing every text vanishes (`Ratpoints.findPointsWith_eq`), and the theorems speak about that channel.

10.6 The program

The executable `ratpoints-model` has the command line of `ratpoints`: the coefficients $c_0, \dots, c_n$ of $y^2 = c_n x^n + \cdots + c_0$ as one argument, the height bound and the options. It reads them as `main.c` does, with its checks, messages and exit statuses in its order, runs the search with the callback of `main.c`, and prints what `ratpoints` prints, but for the banner and the usage, which name `ratpoints-lean` (the earlier name of the program), so that the test scripts of the C repository can run it in place of ratpoints. Its root `ModelMain.lean` imports only the model, so that the program depends neither on Mathlib nor on the proofs. It acts on every option of `ratpoints` but `-t`, the number of threads, which it checks and ignores: the model sifts the blocks of the run in turn, and the thread lemma (Theorem 3.15) shows that blocks sifted apart under hypotheses that the pool of `ratpoints` meets by its code, and delivered in their order, give the same output.

Definition 10.6 (The command line)**.** The coefficient list, the height bound, the options of the sieve, the options of the search and the settings of the output that a command line gives, or its first failure, in the order of `main.c`. Fewer than two arguments fail with error 2 (`main.c:210-214`). The first argument gives the coefficients, constant term first, integers separated by white space: a word that is not an integer fails with error 11 wherever it stands, then more than maxDegree $+1$ integers fail with error 3 and fewer than two with error 5. The second gives the height bound, a C `long` of at least 1, or error 4. The options are read from left to right, the last setting of each kind winning. An unknown option, an argument among the options that does not begin with `-`, a missing or malformed argument of an option (but for `-l` and `-u`) and `-f` or `-d` followed by another character fail with error 6 (`main.c:393-400`, `main.c:437-532`, `main.c:533-577`, `main.c:638-649`, `main.c:650-668`); `-l` and `-u` read the ends of the search intervals (Definition 10.7), with error 7 for ends that are missing, no numbers, not alternating, too many or not ordered. Then a pending interval is completed, the format is `(%x : %z)\n` under `-x` or `-y` and `(%x : %y : %z)\n` otherwise, unless `-f` gave one, and `-q` cancels `-v` (`main.c:688-688`). A failure prints the message of the function `error` of `main.c`, for errors 2 and 6 with the banner and the usage (`main.c:1036-1048`).

Definition 10.7 (The numbers and the search region of the command line)**.** The readers of the numbers, on the words of an argument, its maximal runs of characters other than C white space: an integer of the coefficient list as `scan_mpz` reads it, with an optional sign and of any size; a C `long` as `scan_long` reads it, from -2^{63} to $2^{63} - 1$, with only white space around it (`main.c:733-740`); and a real number as `scan_double` reads it, the double that `strtod` of glibc reads in the C locale, none for a NaN (`main.c:746-752`): $\pm\infty$ for an infinity, and otherwise the double nearest to the exact value $\pm m\,10^{e-k}$ of the word, or $\pm m\,2^{e-4k}$ for a hexadecimal word, with ties to even. An end of a search interval as `scan_end` reads it: the number that the argument denotes, or an infinity, also for a number beyond the range of the doubles, and $\pm 2^{-1075}$ for a nonzero number that `strtod` reads as 0, which lies between 0 and every other x of the search as the number does; with the double that the library is given for it, the double of `strtod` moved outward to the largest double at most a lower end, or the smallest double at least an upper end (`main.c:798-806`). The reading of the intervals of `-l` and `-u` into the doubles and the exact ends of at most maxDegree intervals (`main.c:66-66`, `main.c:73-73`), a missing first lower end or last upper end standing for $\mp\infty$ with the double $\mp h$ of the height bound, or $\mp\infty$ when that double would turn the interval round. The test whether a/b lies in one of these intervals, by the binary search of `in_region` over their exact ends (`main.c:126-137`). And the intervals that the library is given, those of the command line joined where their doubles meet or overlap (`main.c:891-905`).

Lemma 10.8 (The numbers of the command line)**.** `strtod` *never gives a NaN, and when it gives a finite double other than zero for a word that is not an infinity, the exact value of the word lies below the next double above it: half of a faithful rounding, the half that the lower ends*

need. `outward` *moves a double to the side of the end that its direction asks, and onto the largest double at most a lower end when it starts from a double whose next double lies above the end; so for an argument of fewer than* 2^{56} *characters the double that* `scan_end` *gives for a lower end is the largest double at most the end (for an upper end the proofs show only that it is at least the end). The binary search of* `in_region` *decides whether* a/b *lies in increasing and disjoint exact intervals. For intervals without a NaN,* `join_intervals` *gives no interval exactly when it is given none, and intervals without a NaN that hold every number that the given ones hold; when moreover each lower end is at most its upper end and the lower ends increase from the second on, its intervals are increasing and strictly apart, search intervals in the sense of the specification (Definition 2.6).*

Proof. The parts of a word give its exact value, and the exponent that `scan_end` accumulates, which stops at `LONG_MAX / 20`, is the exact one for arguments of fewer than 2^{56} characters whose double is finite and not zero; the rounding of `strtod` is that of the doubles (Chapter 5). The loops of `outward`, `in_region` and `join_intervals` are proved by named invariants with the program logic of Lean core. □

Definition 10.9 (The callback of the program)**.** The data that `main.c` passes to its callback `process` (`main.c:110-113`): whether a point was printed, the settings of the output that `process` reads, the search intervals of the command line, before they are joined, with their exact ends, and the text printed so far. The callback on (x, z, y) (`main.c:139-142`, `main.c:150-156`) passes over, with the weight 0 and the flag clear, a point at infinity under `-i` and an affine point whose x/z lies in none of the search intervals of the command line, when it gives some (the library searches them with the doubles outward of the exact ends); every other point is one that it keeps. It gives such a point the weight 1, stops the search at it under `-1`, and prints it in the format unless `-z`, after the string before the points if it is the first point printed and after the string between them otherwise. The format prints `%x`, `%y` and `%z` as the coordinates in decimal (`%y` as itself under `-x` or `-y`), `\t`, `\n`, `\\` and `\%` as a tab, a line feed, a backslash and a percent sign, and every other character as itself (`main.c:157-177`, `main.c:178-182`, `main.c:183-186`); a string prints in the same way, its percent signs not special (`main.c:952-960`).

The delivery of a list to this callback prints, unless `-z`, the points of the list that it keeps, under `-1` only the first of them, and its count is their number (`Ratpoints.Cli.deliver_process`); the theorem on what the program prints (Theorem 3.9) rests on this and on the protocol of the delivery.

Definition 10.10 (The program and its report)**.** The messages of the report of `main.c` (`main.c:979-1033`), among them the search region of the command line (message 3), the numbers of the moduli used (message 4), what the search did, from the bits written back (message 7), the search intervals as the search leaves them (message 8), and the count of the points, or of the point pairs under `-x` or `-y` (message 2). The program on a command line (`main.c:195-198`): on a failure of the command line, the message and the status of the failure. Otherwise, unless `-q`, the head of the report: the banner, the curve, the height bound unless `-v` (`main.c:215-220`), and message 3 when the command line gives intervals (`main.c:224-224`); then the search with a print channel (Definition 3.1), with the options of the sieve of the command line and the array size that `main.c` sets (`main.c:221-222`), the options of the search on the joined intervals (`main.c:248-248`), the callback of the program on its data and, under `-v`, the print channel that appends to the text of the data (`main.c:251-252`). On a refusal it prints what the library printed and the message of the error that `main.c` gives for it (`Ratpoints.Cli.Failure.ofRefusal`: error 8 for a binary form that is not squarefree), and exits with that status. Otherwise it prints what the callback and the library printed, the string before the points if no point was printed (`main.c:265-265`), the string

after them (main.c:267-267) and, unless -q, the end of the report: two line feeds and messages 4, 7 and 8 unless -v, then message 2 (main.c:268-276); its status is 0 (main.c:284-285).

The exit statuses are thus those of main.c: 0 after a search, 2 to 7 and 11 for the command line, 8 for a binary form that is not squarefree, and 9 for search intervals that the library refuses (main.c gives 9 for every code of the library but those of errors 8 and 10); by Theorem 3.9 no command line whose arguments are shorter than 2^{56} characters leads to error 9. Error 10, for lack of memory, has no counterpart. When the reader of its output is gone, SIGPIPE ends ratpoints and the shell reports the status $141 = 128 + 13$; the Lean runtime ignores that signal, and ratpoints-model exits quietly with the status 141 when its write fails.

Remark 10.11 (Deviations)**.** Besides the deviations stated above, the program keeps the data of main.c as a value with the text printed so far, for the pointer info and the standard output, and writes its output once, at the end, where main.c prints and flushes after each point. It reads the formats and the strings as Lean strings, decoded from the command line as UTF-8, where main.c copies the bytes; since every escape of main.c is ASCII and no byte of a multibyte character of UTF-8 is, both print the same bytes for every format and string in UTF-8. The lines of main.c without a counterpart, such as the includes and the initialization of the GMP integers, are excluded in the record (Chapter 11).

Chapter 11

The link to the C code

Lean has no semantics of C, and no theorem of this blueprint speaks about the C code of `ratpoints`: the theorems speak about the model, which mirrors the C code component by component, with the same loops, tables, bit arrays of 64-bit words and index arithmetic. This chapter describes what ties the model to the C code of release 3.1.1: the record of which declaration models which lines of C, and its checks; the lines that no declaration models; the departures of the model from the C code; the differential tests of the program `ratpoints-model` against the C program; the builds with other values of the compile-time constants; and CI. The nodes of the chapters before this one cite the main ranges of their definitions; Appendix A, generated from the record, lists every range with its owners and their deviations, or its exclusion.

The link is not a proof. That a definition does what its lines of C do is checked by reading the two side by side; the record makes sure that every line of C is accounted for, that each claim is made for a definite text on both sides, and that a change on either side is noticed; the differential tests check the correspondence on behaviour, on finitely many inputs. Comparator certifies the statements about the model and says nothing about the C code (Section 3.8).

11.1 The record

Every line number refers to one commit of `ratpoints`, the pin: release 3.1.1, tag `v3.1.1`, commit `1a99853`, named in `correspondence/ratpoints.toml` and, for the links of the blueprint, in the macro `\ratpointscommit`. The record covers `find_points.c`, `sift.c`, `init.c`, `sturm.c`, the headers `rp-private.h`, `ratpoints.h` and `primes.h`, the generators of tables `gen_find_points_h.c` and `gen_init_sieve_h.c`, and the program `main.c`. Every line of code of these files, that is, every line that holds something besides white space and comments (a line of the preprocessor counts), lies in exactly one recorded range, and no two ranges overlap.

A range is modelled when the docstring of a public declaration of the model names it, in a paragraph that begins with `C:`; an optional paragraph that begins with `Deviation:` says where the declaration departs from the C code. The docstring of `Ratpoints.Sieve.squaresMod`, for instance, ends with

```
C: `find_points.c:37`, `gen_find_points_h.c:69-88`

Deviation: a function of `p`, computed when it is called, for the row of `p` in the table
`squares` that the build generates and that `find_points.c` includes. -/
```

A range may have several owners (line 37 of `find_points.c` includes three generated tables and is a range of the three functions that compute them), and a declaration that calls another,

which models a part of its C code, cites the lines around that part. A private declaration carries no record, since no node can name it. A range is excluded when a table `[[excluded]]` of the record gives it, with a kind and the reason (Section 11.2).

The script `scripts/correspondence.py` checks the record. It reads the docstrings of the model with the help of `scripts/Correspondence.lean`, and the C files at the pin from a clone of `ratpoints`, and it checks:

(1) that the pin is a commit of the clone, to which its tag points where the clone has the tag, that every range lies within the lines of a file of the record, that no line lies in two ranges, and that every exclusion has a known kind and a reason;

(2) that every range and every C link of the blueprint is locked: for every range, the lock holds the SHA-256 hash of its text at the pin and its owners (the declarations that model it, or its exclusion), and for every link the hash of the text that it shows; a range whose text or owners differ from those of the lock, and a new link, are reported until they have been compared with their counterparts by hand and locked again by `scripts/correspondence.py lock`, which writes the lock into `correspondence/lock.json`;

(3) that the appendix is the one that the record generates, and that every link inside a node of the dependency graph that names declarations with `\lean` is one of the ranges of the model declarations that the node names, exactly as recorded (a node need not cite all of them; prose and remarks may link to any range);

(4) with `--complete`, which CI passes, that every line of code lies in a range.

After a move of the pin, a range counts as locked when some locked range has its text and its owners, and a link when some locked link shows its text, wherever their lines are; so a move of the pin to another release of `ratpoints` makes the check report exactly the ranges whose text or owners changed and the links whose text changed (`correspondence/README.md` describes the procedure).

11.2 The exclusions

An excluded range has one of the kinds `threads`, `memory`, `instrumentation`, `interface` and `other`.

`threads`: the pool of threads, with which `ratpoints` sieves the blocks of its loop over the denominators apart (`find_points.c:4129-4519`), its set-up in the search, the atomic loads and stores of the fields that the threads share (`rp-private.h:76-84`), the fields of the arguments that hold the number of threads and the pool, and the default of one thread in `main.c`. The model sieves the blocks in turn. The thread lemma (Theorem 3.15) shows that blocks sifted apart and delivered in their order give the same; that the pool of the C code meets its hypotheses is read off the C code (Chapter 3).

`memory`: the buffers of the sieve entries, the rows, the tables, the words of the forbidden divisors and the prime powers, with their alignment and enlargement (`find_points.c:181-270`) and their release (`find_points.c:377-416`), the arrays of the run and of a worker, the copy of the caller's intervals in the Sturm step, the GMP variables that `main.c` initializes and releases, and the failures for lack of memory, error 10 of the program among them. The model allocates nothing itself and cannot fail for lack of memory; such failures inside modelled ranges, such as the drop of the forbidden divisors beyond the table whose words find no memory, are deviations of their owners.

`instrumentation`: the output under `DEBUG`, the timers and counters of `RP_PHASE_TIMING` and `RP_PHASE_COUNTS` (`sift.c:55-176`), the report of `RP_PRIME_STATS`, `RP_STOP_AFTER`, `RP_MOD_CHOICE`,

RP_MOD_COUNTS, the check RP_VERIFY_MODULI of the rows (init.c:299-353) and two probes for a benchmark, all compiled only on request or empty in the default build. Output under DEBUG inside a modelled block goes with the block, whose deviation says that the model prints nothing there; the report under RATPOINTS_VERBOSE (-v) is modelled (Chapter 10).

interface: what only another caller of the library meets: find_points_init (find_points.c:272-375) and its prototype, the input fields of the arguments that find_points_work saves and puts back, the checks of arrays that only such a caller can fail (find_points.c:4618-4623), the declaration of the arguments in main.c and the initialization of the fields that the model passes as values, and the prototype of the Sturm step.

other: lines without meaning for the model: the includes, those of the vector intrinsics among them, and the prototypes; the inlining attributes and the pragma that unrolls the loops of tail_leg (sift.c:466-472); FLOOR, RATPOINTS_DEFAULT_SP1 and RATPOINTS_DEFAULT_SP2, which the C code reads nowhere, and a count stored that nothing reads; the text of message 1, which main never prints (main.c:977-978); the frames of the main functions of the generators of tables; and the arm of exact_scale for a build without a 128-bit integer type (find_points.c:3239-3256), since the default build compiles the arm with unsigned __int128.

11.3 The deviations

The Deviation: paragraphs record, declaration by declaration, where the model departs from its lines of C. Most of the departures are of the following kinds. None corrects the C code: the model is that of release 3.1.1 as it is.

Integer types. The model computes with natural numbers and integers where the C code has longs, ints, unsigned longs and GMP integers, and with rational numbers for GMP rationals; where a value of the C code could leave its type, the range obligation is a lemma that the deviation names. Lean's 64-bit words stand where the C code depends on the words themselves: in the bit arrays, the reductions by a reciprocal and the counters of the run, which wrap as unsigned longs wrap; in the test for common factors (Ratpoints.Sieve.relprime), whose step without branches acts on the two's complement of a negative long and shifts it arithmetically, as gcc defines these operations, which the C standard leaves to the implementation; and in the bits of the numerators of an interval, the forms of the ends of the intervals and the numerators of the Sturm step, whose arithmetic wraps where that of the longs would overflow, on no path that the search takes. The 128-bit products are taken on two words, the high word from four products of 32-bit halves (Ratpoints.Sieve.mulHi). The height bound is a natural number, refused from 2^{63} on with error 4, as the program refuses such a height when it reads it.

Pointers and buffers. Arrays and indices stand for pointers into arrays, an option for a pointer that may be null, and values returned, often structures, for what the C code writes through pointers or into buffers of its caller; state that the C code reaches through pointers, such as the delivery of the points, the tables of rows (Ratpoints.Sieve.RowStore) and the counters, is passed and returned. A search gives back what the program reads (Definition 2.11), not the other fields that the C code leaves in the arguments. The end of a row that is not built yet is p (Ratpoints.Sieve.setRows), where set_rows adds p to the null pointer (find_points.c:3149-3170), which the C standard leaves undefined; this happens in ordinary runs, whenever set_rows reports a row as missing, and the end is set again when the row is built, before it is read.

Generated tables. The squares, the inverses and the rows of 0 of the primes, which the build generates into find_points.h, are functions of the prime, computed when they are called: Ratpoints.Sieve.squaresMod, Ratpoints.Sieve.inversesMod and Ratpoints.Sieve.sieves0. The

functions that the build generates into `init_sieve.h`, one for each prime, which build its rows, are two functions, `Ratpoints.Sieve.sieveInit1` and `Ratpoints.Sieve.sieveInit2`, with the prime an argument; and a pointer to such a function is the kind of the builder, with a match on it (Chapter 7).

Vector code. A bit array of the default build is a vector of 256 bits; in the model it is rbaPack = 4 consecutive words of an array of `UInt64`, and the `&` of two bit arrays the `&` of their words (`Ratpoints.Sieve.andBitArray`). The registers of the chunked first stage are a vector of chunk = 16 bit arrays, the code that the build unrolls under `#if (RATPOINTS_CHUNK >= k)` is a loop over them (`Ratpoints.Sieve.chunkSieve`), and the conditional compilation that chooses an arm is a test on the constant (Chapter 9). The other builds differ in these constants (Section 11.5).

Doubles. The doubles of the search region are their IEEE 754 bits (`Ratpoints.Double`), with a function, which owns no range of C, for each operation that `ratpoints` applies to them, as gcc and glibc compute it with rounding to nearest; `strtod` and `printf` under `%f` are modelled by their values, not by glibc's algorithms (Chapter 5). The doubles of the cost model are Lean's `Float`, with the platform's arithmetic, `log` and `pow`, and no proof reads their values; the sorts by `qsort` are a stable merge sort (`Ratpoints.sortFrom`), which gives the order of glibc's `qsort`, a stable merge sort whenever it can allocate its buffer, since no key is a NaN (Chapter 6). Where the C code computes in `double` what the model computes exactly (the comparison of $2pH$ with the limit of the third stage, the integer square roots of the shapes), the deviation shows that the outcome is the same.

Structure and the program. The model has one function, `Ratpoints.findPointsVerbose`, for `find_points`, `find_points_work` and the start of `find_points_work_1`, whose rest is `Ratpoints.findPointsWork`; parts of long functions and bodies of loops are functions of their own; arrays that the C code fills once per denominator under a flag are thunks; and the report under `RATPOINTS_VERBOSE` goes through a print channel of the search. The program writes its output once, at the end, where `main.c` prints after every point; it differs from `ratpoints` on a closed pipe and on format words that are not UTF-8, where the runtime of Lean acts otherwise than that of C (Chapter 10).

Inputs that `ratpoints` never meets. Where the C code would overflow or divide by zero on an input that it is never given, the model takes a defined value, and the contract excludes the input or the proofs show that it does not occur: the negation of -2^{63} in the test for common factors wraps, and the integer square root rounded up gives 1 at 0. The loop that counts the trailing zeros does not end on 0, as in the C code (`Ratpoints.Sieve.ctz`), and the masks of the forbidden valuations drop the bound $m < 63$ of the C code, which never binds for a height that is a `long` (Chapter 8).

11.4 The differential tests

The script `scripts/difftest.py` runs `ratpoints-model`, the model compiled by Lean (Chapter 10), on the test suites of `ratpoints` and on further command lines, and compares it with the outputs that the suites record and with the C program at the pin on the same command line; the C program must be the default build, without the machine-dependent tuning that `make tune` writes. Every list of points is compared in its order. Under `-q` the points are compared with a recorded output and byte for byte with the output of the C program, and so are the outputs with the options and formats of the output and the reports, with messages 3, 4 and 8 on the search region, the moduli used and the search intervals, the banner exchanged for that of the C program, also under `-v`, where the report of the library comes between the points. The outputs that `test4.sh` filters are compared after the filter and as a whole; those that `test3.sh` filters, as

a whole, before the filter. Under `-x`, where the program prints the pairs that survive the sieve, the pairs are compared byte for byte with the C program, since the model chooses and corrects its moduli as `ratpoints` does, and, without a format and without `-v`, they also pass a check that does not depend on the moduli: normalized coprime pairs in the range of the height and in the search region, none repeated, and the points among them the same, in the same order, in both outputs. Exit statuses and messages are compared exactly, but for the banner and the usage; `ratpoints-model` must write nothing to its standard error (a panic of the Lean runtime ends it); and failed writes are compared too.

The default suites search up to the height 30000, and above it in narrow search regions. Since the choice of the moduli computes with the `log` and `pow` of the platform's C library, which glibc does not round correctly and chooses by the processor, the comparisons that depend on it (the pairs under `-x`, message 4 and the report under `-v`) are made with both programs on one machine; CI runs both on its runner. On the state of the project that this blueprint describes, the suites make these comparisons, none of which fails:

suite	what it runs	comparisons
`test4`	the runs of `test4.sh`, against `testbase4` and the C program, but for 13 above the height 30000	590
`test3`	the runs of `test3.sh`	21
`rptest`	the curves of `testdata.h`, `testdata-many.h` and `testdata-degrees.h` at the height 100, against their recorded points	1206
`random`	random polynomials of degree 1 to 8 at heights up to 30	300
`outputs`	random options of the output, formats and strings	300
`errors`	hand-written command lines for every check of `main.c` and every option, the refusals, the report under `-v`, searches stopped at once at the height 10^{12}, failed writes	384
`fuzz`	random command lines	800
`params`	the options that set the moduli and the cut points	272
`high`	point-rich curves up to the height 30000, and curves on which the C program corrects its moduli	19
`regions`	random search regions and ranges of the denominators	400
`depths`	random depths of the Sturm step	300
`tuning`	random values of the tuning options	300
`threads`	the C program with `-t 4` and `-t 7`	205
`verbose`	random searches under `-v`	300
	the default suites	5397
`test4high`	the runs of `test4.sh` at the heights 40000 to 100000	13
`adapt`	searches at the heights 16383 to 50000 on which the C program corrects its moduli, under `-x` with and without `-A`	92

The last two suites are not run by default, nor the comparison of a build of the C program with a fixed length of the blocks (`-DRP_BLOCK_LEN=3`) with `ratpoints-model` built with the same length (`Ratpoints.blockLen`), which tests the boundaries of the blocks. The `#guard` examples of the model, which every build runs, compare components with values taken from the C program, gcc and glibc, none of which depends on the `log` or `pow` of the platform.

11.5 Other values of the constants

The file `Ratpoints/Model/Constants.lean` holds the constants that the C code fixes when it is compiled, each a definition with the value of the default build and a docstring that names its macro: among them the bound on the degree (`Ratpoints.maxDegree`), the number of bits of the moduli,

the width of a bit array (`Ratpoints.rbaPack`), the chunk of the first stage (`Ratpoints.chunk`), the default depth of the Sturm step, the limits of the reductions by a reciprocal, the doubles that tune the cost model and the constants of the blocks (`Ratpoints.adaptLag`, `Ratpoints.blockLen`). No proof depends on their values beyond a few facts that the file decides: that the width, the chunk and the array sizes are positive, that the limits of the reductions are at most 2^{32}, that a bit array has 2^{rbaShift} bits with rbaShift < 32, that maxBitsInPrime ≤ 10, that the bound on the degree is positive and below 2^{57}, that a fixed length of the blocks is positive, and a few more of that kind.

The script `scripts/constants_check.py` checks this by building copies of the committed project, with all its proofs and without the `#guard` examples of the model, which test the default values, for eleven other sets of values, each of which builds: two that change every constant, one to small values (among them bit arrays of 64 bits, the chunk 1, the lag 0 and blocks of length 1) and one to large values (bit arrays of 512 bits, the chunk 17, which compiles the other arm of the first stage, the lag 1000); four that change the chunk alone, to 2, 3, 5 and 12; three that pair the chunked first stage with the widths of the other builds, 512, 128 and 64 bits; one with the lag 1 and blocks of length 3; and one with one block wanted. So the proofs hold for the models of the configurations of the C code that differ in these constants, the widths of the bit arrays of all its builds among them. The script builds the copies one after the other, with the built packages of the repository, and runs on demand, outside CI; the sets of values are a table in it, and `--variant` gives others.

11.6 Continuous integration

The workflow `.github/workflows/build-project.yml` runs on every push to the main branch and on every pull request. It checks the layout (`scripts/check_layout.py`: the imports of each part; no `partial`, `unsafe`, `implemented_by`, `extern` or experimental option in the model, which would let the executable run code that the proofs do not see; and nowhere a tactic that adds an axiom, such as `native_decide`), the style of the Lean code and the length of its lines, and the listing `Ratpoints.lean` of the modules; builds, with every warning a failure, the model, the specification and the proofs, `ratpoints-model` and the solution of comparator, and then the challenge, whose `sorry`s warn by design; runs the tests of the scripts; checks the record with `--complete` against `ratpoints` at the pin; and builds the C program at the pin and runs the default suites of the differential tests against it. The build fails if a main theorem depends on an axiom beyond `propext`, `Classical.choice` and `Quot.sound`, by the guards `#print axioms` under `#guard_msgs`. Comparator (Section 3.8), the suites `test4high` and `adapt`, the pair with a fixed length of the blocks and the builds with other values of the constants (`scripts/constants_check.py`) run outside CI.

Appendix A

The correspondence with the C code

The model is checked against `ratpoints` v3.1.1 (commit `1a998538dfa8`). The C files below are those of the library and `main.c`, whose parts that the command-line program does not mirror are excluded. Every line of code of them lies in exactly one recorded range: a range is either modelled by the Lean declarations named, or excluded from the model with a reason. Each recorded range is checked against the text it had when it was reviewed.

file	lines of code	modelled	excluded	not recorded
`find_points.c`	3244	2589	655	0
`sift.c`	820	660	160	0
`init.c`	302	238	64	0
`sturm.c`	215	202	13	0
`rp-private.h`	282	246	36	0
`ratpoints.h`	128	114	14	0
`primes.h`	11	11	0	0
`gen_find_points_h.c`	76	60	16	0
`gen_init_sieve_h.c`	26	13	13	0
`main.c`	779	727	52	0
total	5883	4860	1023	0

`find_points.c`

- `find_points.c:30-30`: excluded (other): the include of the private header; the model has no headers
- `find_points.c:32-32`: modelled by `Ratpoints.primeTable`
- `find_points.c:37-37`: modelled by `Ratpoints.Sieve.inversesMod`, `Ratpoints.Sieve.sieves0`, `Ratpoints.Sieve.squaresMod` (deviation of `Ratpoints.Sieve.inversesMod`: a function of `p`, computed when it is called, for the row of `p` in the table `inverses` that the build generates and that `find_points.c` includes; the trial stops before `p` and gives `0` when it finds nothing, where `inv_mod_p` would not stop, which never happens for a prime `p` and a residue other than `0`.) (deviation of `Ratpoints.Sieve.sieves0`: a function of `p`, computed when it is called, for the row of `p` in the table `sieves0` that the build generates and that `find_points.c` includes;

its first `(p + chunk - 1) rbaPack` words, for the row of `RBA_PACK (RATPOINTS_MAX_PRIME_EVEN + RATPOINTS_CHUNK - 1)` words whose other words are `0` and never read.) (deviation of `Ratpoints.Sieve.squaresMod`: a function of `p`, computed when it is called, for the row of `p` in the table `squares` that the build generates and that `find_points.c` includes.)

- `find_points.c:41-43`: excluded (threads): the declarations of the lock of the tables of rows and of the destruction of the pool of threads; the model sieves the blocks in turn on one thread
- `find_points.c:69-118`: excluded (instrumentation): RP_PHASE_TIMING and RP_PHASE_COUNTS, development instrumentation: the counters of sift.c declared extern and the timers RP_BC_TIC/TOC, RP_BP_TIC/TOC, RP_SIFT_TIC/TOC, RP_INIT_TIC/TOC and RP_SETUP_TIC/TOC, which time with rdtsc the part of the exact check that is done once per denominator, the residues of a denominator modulo the moduli of the first two stages, the whole of sift, the building of a row and the set-up of a denominator, with the arm of the production build, where they are empty
- `find_points.c:121-121`: modelled by `Ratpoints.maxDivisors`
- `find_points.c:124-124`: modelled by `Ratpoints.maxArraySize`
- `find_points.c:133-133`: modelled by `Ratpoints.Sieve.sieveInit` (deviation of `Ratpoints.Sieve.sieveInit`: a function of `p`, for the entry of `p` in the table of the pointers to the functions that the build generates, one for each prime, and that `init.c` includes; the kind of the function, for the pointer to it.)
- `find_points.c:147-149`: modelled by `Ratpoints.ModulusCandidate` (deviation of `Ratpoints.ModulusCandidate`: a natural number for the `long` modulus, a `UInt64` for the `unsigned long` mask, an integer for the `short` count, the codes a list of natural numbers, for the array of `RP_MAX_FACTORS` `shorts`, and the sieve entry an option, `none` for the null pointer.)
- `find_points.c:153-153`: modelled by `Ratpoints.maxModuli`
- `find_points.c:155-155`: modelled by `Ratpoints.Setup.DenInfo` (deviation of `Ratpoints.Setup.DenInfo`: natural numbers for the `int` fields, which the primes fit, a cofactor below the square of a prime of the table among them (`Ratpoints.Setup.mul_self_lt_of_mem_primeTable`), and the valuations and the slopes for coefficients below `2 ^ (2 ^ 31 - 1)`.)
- `find_points.c:157-161`: modelled by `Ratpoints.Sieve.ForbiddenEntry` (deviation of `Ratpoints.Sieve.ForbiddenEntry`: the words as an array, for the pointers `start` and `end` into the pattern `sieves0` of the build or into the buffer of the arguments; a natural number for the `long p` and for the `unsigned long` reciprocal `magic`.)
- `find_points.c:166-166`: modelled by `Ratpoints.Sieve.ForbiddenVal` (deviation of `Ratpoints.Sieve.ForbiddenVal` a natural number for the `long p`, and one for the `unsigned long` mask, so that it holds every valuation of a denominator, also beyond the 64 bits of a word (`Ratpoints.Sieve.forbiddenValuations`).)
- `find_points.c:181-270`: excluded (memory): pointer_align, which aligns the buffers and the survivors array; ba_buffer_need, alloc_ba_buffer and ensure_ba_buffer, which size, allocate and enlarge the buffer from which the builders take the rows of the tables; num_powers, the number of prime powers below RATPOINTS_MAX_PRIME_EVEN, which sizes the buffers of the sieve entries and of the prime powers (and gives add_moduli its bound on the prime powers it keeps, line 2351, which never binds); and hold_nothing, which clears

the pointers to the buffers: the model allocates nothing itself and keeps the rows in tables of its own

- find_points.c:272-375: excluded (interface): find_points_init, which a caller of the library calls before find_points_work: it answers a degree outside 0, ..., RATPOINTS_MAX_DEGREE with the work length -1, but find_points_work refuses such a degree itself (lines 4557-4558, which the model has), so the work length -1, refused at lines 4621-4622, matters only to a caller who changes the degree between the two calls; it allocates the scratch integers work and the buffers and arrays of the arguments, giving all of them back when one cannot be had, clears the pointer to the pool of threads, and prints messages under DEBUG; the model allocates nothing

- find_points.c:377-416: excluded (memory): find_points_clear, which destroys the pool of threads, clears the scratch integers, frees the buffers and arrays of the arguments and clears the pointers to them, with messages under DEBUG; the model allocates nothing

- find_points.c:422-422: modelled by `Ratpoints.veryBig`

- find_points.c:424-454: modelled by `Ratpoints.Setup.valuation` (deviation of `Ratpoints.Setup.valuation`: one function for `valuation` on GMP integers, with its scratch integer `vvv`, and `valuation1` on `long`s, which gives `v` alone and is the first component; natural numbers for `|n|`, the quotients and the remainders; the pair returned, for the remainder that `valuation` stores through its pointer `r`.)

- find_points.c:464-471: modelled by `Ratpoints.Sieve.ceilSqrt` (deviation of `Ratpoints.Sieve.ceilSqrt`: the exact square root of the natural numbers (`Nat.sqrt`), for the square root in `double` corrected by a step either way; `1` for `n = 0`, where the C code divides by `0`; natural numbers for the `long`s, so that a negative argument, whose square root in `double` is not a number, cannot occur.)

- find_points.c:477-492: modelled by `Ratpoints.Sieve.modLong` (deviation of `Ratpoints.Sieve.modLong`: one definition for the two copies of the function, in `find_points.c` and in `sift.c`, which are the same but for the hooks of the counts of the reductions in `sift.c` (`RP_MOD_TICK`, `RP_MOD_DIV`), empty unless `RP_MOD_COUNTS` is set; integers for the `long`s, with the obligations that the values it computes stay in the range of a `long` stated as lemmas: for `0 ≤ b < 2 ^ 59`, `16 b` (`Ratpoints.fitsLong_sixteen_mul`), its negation (`Ratpoints.fitsLong_neg`), and `a + 16 b` for `-16 b ≤ a < 0` (`Ratpoints.fitsLong_add_sixteen_mul`), its other values lying between `-b` and `16 b`; `Int.tmod` for C's division of `long`s, which truncates; the multiples `16 b`, `8 b`, `4 b` and `2 b` as products, for the shifts of `b1`.)

- find_points.c:498-566: modelled by `Ratpoints.Sieve.jacobi` (deviation of `Ratpoints.Sieve.jacobi`: one function for `jacobi`, on the GMP integer `lcf`, and `jacobi1`, on a `long`, which the run calls when all coefficients are `long`s (`use_c_long`), with the steps of `jacobi`: `jacobi1` takes the sign of `lcf` out as `(-1 / b)^neg` and its value `LONG_MIN` as `LONG_MIN / 4`, so that the negation of a negative `lcf` is a `long` (`Ratpoints.fitsLong_neg_ite_div_four`), reduces `|lcf|` modulo `b` only for `b < 2^59`, and does not reduce it after dividing `b` by the gcd, which changes the steps but not the symbol; the round of the algorithm and its two subtraction loops, which the C code writes out twice each, as helpers (`Ratpoints.Sieve.jacobiRound`, `Ratpoints.Sieve.jacobiReduce`, `Ratpoints.Sieve.twoSign`); natural numbers for the `long`s `b`, `f`, `n` and `m`, and integers for the `long` sign `s` and the `long` `lcf` of `jacobi1`; `f` the Euclidean `lcf % b`, for `mpz_fdiv_r_ui` with its scratch integer `tmp`; the tests `f = 0` of the C code, at

the start and after each division of `b`, as the condition of the outer loop; the masks of the signs by `&&&` on natural numbers; the symbol as an integer, for the `int`.)

- find_points.c:568-641: modelled by `Ratpoints.Sieve.jacobi` (deviation of `Ratpoints.Sieve.jacobi`: one function for `jacobi`, on the GMP integer `lcf`, and `jacobi1`, on a `long`, which the run calls when all coefficients are `long`s (`use_c_long`), with the steps of `jacobi`: `jacobi1` takes the sign of `lcf` out as `(-1 / b)^neg` and its value `LONG_MIN` as `LONG_MIN / 4`, so that the negation of a negative `lcf` is a `long` (`Ratpoints.fitsLong_neg_ite_div_four`), reduces `|lcf|` modulo `b` only for `b < 2^59`, and does not reduce it after dividing `b` by the gcd, which changes the steps but not the symbol; the round of the algorithm and its two subtraction loops, which the C code writes out twice each, as helpers (`Ratpoints.Sieve.jacobiRound`, `Ratpoints.Sieve.jacobiReduce`, `Ratpoints.Sieve.twoSign`); natural numbers for the `long`s `b`, `f`, `n` and `m`, and integers for the `long` sign `s` and the `long` `lcf` of `jacobi1`; `f` the Euclidean `lcf % b`, for `mpz_fdiv_r_ui` with its scratch integer `tmp`; the tests `f = 0` of the C code, at the start and after each division of `b`, as the condition of the outer loop; the masks of the signs by `&&&` on natural numbers; the symbol as an integer, for the `int`.)
- find_points.c:669-669: modelled by `Ratpoints.jacobiPrimes`
- find_points.c:670-670: modelled by `Ratpoints.jacobiTable`
- find_points.c:672-681: modelled by `Ratpoints.Sieve.JacobiInfo` (deviation of `Ratpoints.Sieve.JacobiInfo`: a list of pairs for the arrays `q` and `nonsq` and their count `nq`, and an array of its own for the array `magic`; the tables as arrays of their own, for the space that they take from the table `jtab` of the run; natural numbers for the `long`s `q` and the `unsigned long`s `magic`, and Booleans for the `unsigned char`s of the tables and of the signs.)
- find_points.c:683-708: modelled by `Ratpoints.Sieve.jacobiSetup` (deviation of `Ratpoints.Sieve.jacobiSetup`: the tables as fresh arrays, for the table `jtab` of the run that `tab` and `tab_len` give, whose bound `Ratpoints.jacobiTable` on their total size is kept; the valuations by `Ratpoints.Setup.valuation`, for `mpz_scan1` at `2` and the loop of exact divisions at the odd primes, the power then divided off at once; natural numbers for the GMP integer `tmp`, the `long`s and the `unsigned long`s of the reciprocals, which are `ULONG_MAX / q + 1` for the odd primes `q`; `none` and the tables returned, for the return values `0` and `1` and the structure that `ji` points to.)
- find_points.c:709-713: modelled by `Ratpoints.Sieve.nonSquares` (deviation of `Ratpoints.Sieve.nonSquares`: a fresh array, for the space in the table `jtab` of the run; Booleans for the `unsigned char`s, natural numbers for the `long`s `x` and `s`.)
- find_points.c:714-720: modelled by `Ratpoints.Sieve.jacobiSetup` (deviation of `Ratpoints.Sieve.jacobiSetup`: the tables as fresh arrays, for the table `jtab` of the run that `tab` and `tab_len` give, whose bound `Ratpoints.jacobiTable` on their total size is kept; the valuations by `Ratpoints.Setup.valuation`, for `mpz_scan1` at `2` and the loop of exact divisions at the odd primes, the power then divided off at once; natural numbers for the GMP integer `tmp`, the `long`s and the `unsigned long`s of the reciprocals, which are `ULONG_MAX / q + 1` for the odd primes `q`; `none` and the tables returned, for the return values `0` and `1` and the structure that `ji` points to.)
- find_points.c:721-729: modelled by `Ratpoints.Sieve.jacobiSigns` (deviation of `Ratpoints.Sieve.jacobiSigns`: Booleans for the bits.)
- find_points.c:730-731: modelled by `Ratpoints.Sieve.jacobiSetup` (deviation of `Ratpoints.Sieve.jacobiSetup`: the tables as fresh arrays, for the table `jtab` of the run that `tab` and `tab_len` give, whose

bound Ratpoints.jacobiTable on their total size is kept; the valuations by Ratpoints.Setup.valuation, for mpz_scan1 at 2 and the loop of exact divisions at the odd primes, the power then divided off at once; natural numbers for the GMP integer tmp, the longs and the unsigned longs of the reciprocals, which are ULONG_MAX / q + 1 for the odd primes q; none and the tables returned, for the return values 0 and 1 and the structure that ji points to.)

- find_points.c:734-759: modelled by Ratpoints.Sieve.jacobiTest (deviation of Ratpoints.Sieve.jacobiTest: the list of the primes with their tables walked with the positions (List.zipIdx), at which the reciprocals are read, for the index i of the arrays; the odd part as the shift by the lowest set bit of the word b (Ratpoints.Sieve.ctz, the loop of RP_CTZL); b mod 8 for b & 7; natural numbers for the longs, the difference b - k q truncated at 0, which is the difference of the C code, since k q ≤ b where the multiplication gives the quotient; a Boolean for the unsigned char s, with exclusive or for ^= and negation for the test s == 0, and the test returned as a Boolean, for the int.)

- find_points.c:766-776: modelled by Ratpoints.Setup.setupUs1 (deviation of Ratpoints.Setup.setupUs1: the leading coefficient read from the list, for the integer work[0] that the caller sets to |c_d| (lines 4911-4912); lists, for the arrays den_info and divisors of the arguments (ratpoints.h:390) with their terminating zeros; none for a factorization that is not complete or too many divisors, where setup_us1 leaves the flag RATPOINTS_USE_SQUARES1 clear, and the lists returned, where it sets the flag; no output under DEBUG.)

- find_points.c:777-860: modelled by Ratpoints.Setup.factorLcf (deviation of Ratpoints.Setup.factorLcf: the valuations by Ratpoints.Setup.valuation, for mpz_scan1 at 2 and for the loop of divisions with remainder at the odd primes, the power then divided off at once; natural numbers for the GMP integers work[0] and work[1], the longs, and the int p with its square p * p (line 800), which is below 2 ^ 31 (Ratpoints.Setup.mul_self_lt_of_mem_primeTable); a list of pairs of natural numbers and the cofactor returned, for the int fields p and val of the entries of den_info (they hold the primes, the cofactor below p ^ 2 and, for coefficients below 2 ^ (2 ^ 31 - 1), the valuations: Ratpoints.Setup.DenInfo), for their count, the terminating zero and the integer work[0]; no output under DEBUG.)

- find_points.c:863-863: modelled by Ratpoints.Setup.setupUs1 (deviation of Ratpoints.Setup.setupUs1: the leading coefficient read from the list, for the integer work[0] that the caller sets to |c_d| (lines 4911-4912); lists, for the arrays den_info and divisors of the arguments (ratpoints.h:390) with their terminating zeros; none for a factorization that is not complete or too many divisors, where setup_us1 leaves the flag RATPOINTS_USE_SQUARES1 clear, and the lists returned, where it sets the flag; no output under DEBUG.)

- find_points.c:864-880: modelled by Ratpoints.Setup.squarefreeDivisors (deviation of Ratpoints.Setup.squarefreeDivisors: a list, for the array divisors and its pointers div, div0 and div1; natural numbers for the longs, the products x p being formed only for x ≤ ⌊bHigh / p⌋, so that they are at most bHigh and longs for a long bHigh (Ratpoints.Setup.fitsLong_mul_of_le_div)

- find_points.c:881-893: modelled by Ratpoints.Setup.setupUs1 (deviation of Ratpoints.Setup.setupUs1: the leading coefficient read from the list, for the integer work[0] that the caller sets to |c_d| (lines 4911-4912); lists, for the arrays den_info and divisors of the arguments (ratpoints.h:390) with their terminating zeros; none for a factorization that is not complete or too many divisors, where setup_us1 leaves the flag RATPOINTS_USE_SQUARES1 clear, and the lists returned, where it sets the flag; no output under DEBUG.)

- find_points.c:894-912: modelled by Ratpoints.Setup.slope (deviation of Ratpoints.Setup.slope: integers for the ints, and the slope as a natural number; no output under DEBUG.)
- find_points.c:914-935: modelled by Ratpoints.Setup.setupUs1 (deviation of Ratpoints.Setup.setupUs1: the leading coefficient read from the list, for the integer work[0] that the caller sets to |c_d| (lines 4911-4912); lists, for the arrays den_info and divisors of the arguments (ratpoints.h:390) with their terminating zeros; none for a factorization that is not complete or too many divisors, where setup_us1 leaves the flag RATPOINTS_USE_SQUARES1 clear, and the lists returned, where it sets the flag; no output under DEBUG.)
- find_points.c:981-981: modelled by Ratpoints.Sieve.squares64
- find_points.c:983-994: modelled by Ratpoints.Sieve.twoAdic (deviation of Ratpoints.Sieve.twoAdic: the words returned, for the classes cls and the word den_bits that get_2adic_info fills from them in the same function, which Ratpoints.Sieve.numClasses makes from the words; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); remainders for the masks by 0x3f; natural numbers for the longs k, b, t and a; no output under DEBUG.)
- find_points.c:995-995: modelled by Ratpoints.Sieve.numClasses (deviation of Ratpoints.Sieve.numClasses: the classes and den_bits returned, for the array cls and the word den_bits that get_2adic_info fills through its pointer arguments, from the words that Ratpoints.Sieve.twoAdic returns; the packed pattern of a class with k ≥ 1 by a function of its own (Ratpoints.Sieve.packPattern); the pattern of a class as one word, for the vector RBA(packed) of line 1098; natural numbers for the longs k, a0 and a, and the remainder by 2 ^ k for the mask (1L << k) - 1; the mask of a stride read from the array of the masks with the default 0 beyond its seven entries, as the C code reads the zeros that it puts beyond them; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); no output under DEBUG.)
- find_points.c:996-1000: modelled by Ratpoints.Sieve.twoAdic (deviation of Ratpoints.Sieve.twoAdic: the words returned, for the classes cls and the word den_bits that get_2adic_info fills from them in the same function, which Ratpoints.Sieve.numClasses makes from the words; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); remainders for the masks by 0x3f; natural numbers for the longs k, b, t and a; no output under DEBUG.)
- find_points.c:1002-1004: modelled by Ratpoints.Sieve.coeffsMod64 (deviation of Ratpoints.Sieve.coeffsMod64: an array of D + 1 words, for the array cm on the stack; the Euclidean % of the integers, for mpz_fdiv_ui.)
- find_points.c:1006-1014: modelled by Ratpoints.Sieve.fsq (deviation of Ratpoints.Sieve.fsq: Horner's rule from 0, where the C code starts from cm[D], with the same value.)
- find_points.c:1015-1021: modelled by Ratpoints.Sieve.gsq (deviation of Ratpoints.Sieve.gsq: Horner's rule from 0, where the C code starts from cm[0], with the same value.)
- find_points.c:1023-1050: modelled by Ratpoints.Sieve.twoAdic (deviation of Ratpoints.Sieve.twoAdic: the words returned, for the classes cls and the word den_bits that get_2adic_info fills from them in the same function, which Ratpoints.Sieve.numClasses makes from the words; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); remainders for the masks by 0x3f; natural numbers for the longs k, b, t and a; no output under DEBUG.)
- find_points.c:1058-1062: modelled by Ratpoints.Sieve.classMasks (deviation of Ratpoints.Sieve.classMasks: an array of UInt64, for the array of RP_NUM_STRIDES unsigned longs, which the C code initializes with these seven words and zeros beyond them, and an array read with the default 0 beyond its seven entries.)

- find_points.c:1064-1081: modelled by Ratpoints.Sieve.numClasses (deviation of Ratpoints.Sieve.numClasses: the classes and den_bits returned, for the array cls and the word den_bits that get_2adic_info fills through its pointer arguments, from the words that Ratpoints.Sieve.twoAdic returns; the packed pattern of a class with k ≥ 1 by a function of its own (Ratpoints.Sieve.packPattern); the pattern of a class as one word, for the vector RBA(packed) of line 1098; natural numbers for the longs k, a0 and a, and the remainder by 2 ^ k for the mask (1L << k) - 1; the mask of a stride read from the array of the masks with the default 0 beyond its seven entries, as the C code reads the zeros that it puts beyond them; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); no output under DEBUG.)
- find_points.c:1082-1094: modelled by Ratpoints.Sieve.packPattern (deviation of Ratpoints.Sieve.packPattern: a function of its own for the two loops of the packed pattern, which the loop over the classes of get_2adic_info calls for k ≥ 1 (Ratpoints.Sieve.numClasses), as the loop of the binary algorithm of relprime is one (Ratpoints.Sieve.relprimeOdd); natural numbers for the longs a, a0, t and the shift a of the doubling, s here: the difference a - a_0 of natural numbers, which a ≥ a_0 makes the difference of the C code, the remainder by 64 for the mask by 0x3f, the same on a non-negative value, and the division by 2 ^ k for the shift by k; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz).)
- find_points.c:1095-1097: modelled by Ratpoints.Sieve.numClasses (deviation of Ratpoints.Sieve.numClasses: the classes and den_bits returned, for the array cls and the word den_bits that get_2adic_info fills through its pointer arguments, from the words that Ratpoints.Sieve.twoAdic returns; the packed pattern of a class with k ≥ 1 by a function of its own (Ratpoints.Sieve.packPattern); the pattern of a class as one word, for the vector RBA(packed) of line 1098; natural numbers for the longs k, a0 and a, and the remainder by 2 ^ k for the mask (1L << k) - 1; the mask of a stride read from the array of the masks with the default 0 beyond its seven entries, as the C code reads the zeros that it puts beyond them; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); no output under DEBUG.)
- find_points.c:1098-1098: modelled by Ratpoints.Sieve.andRows, Ratpoints.Sieve.andRowsChunked (deviation of Ratpoints.Sieve.andRows: the bit arrays set to the class word by a loop that counts up, bit array n - 1 - i at the step i, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector RBA(packed) that get_2adic_info makes once per class (find_points.c:1098) and _ratpoints_sift0 copies into each bit array (bits64, sift.c:519, sift.c:954); the moduli of the stage with their rows and their reciprocals as two lists, for the first sp1 sieve specs and the array magics from position 0 on; the value -w_low - offset written out in both reductions, for the long a, which stays a long where the helper mod reduces it outside small (Ratpoints.Sieve.fitsLong_neg_sub); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the longs; a Boolean for the int small; no output under DEBUG.) (deviation of Ratpoints.Sieve.andRowsChunked: no moduli tested by the size of the local array of the sieve specs, for sp1 == 0; the bit arrays set to the class word by a loop that counts up, bit array n - 1 - i at the step i, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector RBA(packed) that get_2adic_info makes once per class (find_points.c:1098) and _ratpoints_sift0 copies into each bit array (bits64, sift.c:519); the end of the whole chunks the index full, for the word number w_high_full, which is also the index of the first bit array of the tail, for the pointer surv after the chunks; natural numbers and integers for the longs.)
- find_points.c:1099-1104: modelled by Ratpoints.Sieve.numClasses (deviation of Ratpoints.Sieve.numClasses:

the classes and `den_bits` returned, for the array `cls` and the word `den_bits` that `get_2adic_info` fills through its pointer arguments, from the words that `Ratpoints.Sieve.twoAdic` returns; the packed pattern of a class with `k ≥ 1` by a function of its own (`Ratpoints.Sieve.packPattern`); the pattern of a class as one word, for the vector `RBA(packed)` of line 1098; natural numbers for the `long`s `k`, `a0` and `a`, and the remainder by `2 ^ k` for the mask `(1L << k) - 1`; the mask of a stride read from the array of the masks with the default `0` beyond its seven entries, as the C code reads the zeros that it puts beyond them; the lowest set bit by the loop of `RP_CTZL` (`Ratpoints.Sieve.ctz`); no output under `DEBUG`.)

- `find_points.c:1106-1114`: modelled by `Ratpoints.Sieve.numClasses` (deviation of `Ratpoints.Sieve.numClasses`: the classes and `den_bits` returned, for the array `cls` and the word `den_bits` that `get_2adic_info` fills through its pointer arguments, from the words that `Ratpoints.Sieve.twoAdic` returns; the packed pattern of a class with `k ≥ 1` by a function of its own (`Ratpoints.Sieve.packPattern`); the pattern of a class as one word, for the vector `RBA(packed)` of line 1098; natural numbers for the `long`s `k`, `a0` and `a`, and the remainder by `2 ^ k` for the mask `(1L << k) - 1`; the mask of a stride read from the array of the masks with the default `0` beyond its seven entries, as the C code reads the zeros that it puts beyond them; the lowest set bit by the loop of `RP_CTZL` (`Ratpoints.Sieve.ctz`); no output under `DEBUG`.)
- `find_points.c:1115-1115`: modelled by `Ratpoints.Sieve.twoAdic` (deviation of `Ratpoints.Sieve.twoAdic`: the words returned, for the classes `cls` and the word `den_bits` that `get_2adic_info` fills from them in the same function, which `Ratpoints.Sieve.numClasses` makes from the words; the lowest set bit by the loop of `RP_CTZL` (`Ratpoints.Sieve.ctz`); remainders for the masks by `0x3f`; natural numbers for the `long`s `k`, `b`, `t` and `a`; no output under `DEBUG`.)
- `find_points.c:1119-1130`: modelled by `Ratpoints.Sieve.numPackings` (deviation of `Ratpoints.Sieve.numPackings`: an array of two words, for the array `seen` of two `unsigned long`s; natural numbers for the `long`s `key` and `n`, the division by 64 and the remainder by 64 for the shift by 6 and the mask by 63 of the key; a natural number for the `long` result.)
- `find_points.c:1142-1158`: modelled by `Ratpoints.Sieve.classOffsets` (deviation of `Ratpoints.Sieve.classOffsets`: the classes, the entries and the array of the offsets returned, for the arrays that `class_offsets` writes through its pointer arguments; the number `np` of the moduli the number of the sieve entries `es`, for the run all `sp3_max` of them (`Ratpoints.sieving`), with the reserve of further primes that the correction of the moduli may take into the second stage during the run (`Ratpoints.Sieve.adaptPrimes`); the inverses of `2 ^ k` an array of `kmax + 1` entries, for the array `dinv` of `RP_NUM_STRIDES` `long`s of the sieve entry, whose entries beyond `kmax` stay unset and unread; the inverses `rinv` an array of arrays, for the array on the stack; the loop over the classes by a function of its own (`Ratpoints.Sieve.classRows`); natural numbers for the `long`s.)
- `find_points.c:1159-1160`: modelled by `Ratpoints.Sieve.halve` (deviation of `Ratpoints.Sieve.halve`: a function of its own for the expression that the C code writes out for `d` and for `e`; the remainder by 2 for the mask `& 1` and the division by 2 for the shift `>> 1`; natural numbers for the `long`s.)
- `find_points.c:1161-1162`: modelled by `Ratpoints.Sieve.classOffsets` (deviation of `Ratpoints.Sieve.classOffsets`: the classes, the entries and the array of the offsets returned, for the arrays that `class_offsets` writes through its pointer arguments; the number `np` of the moduli the number of the sieve entries `es`, for the run all `sp3_max` of them (`Ratpoints.sieving`), with the reserve of further primes that the correction of the moduli may take into the second stage during the run (`Ratpoints.Sieve.adaptPrimes`); the inverses of `2 ^ k` an array of

kmax + 1 entries, for the array dinv of RP_NUM_STRIDES longs of the sieve entry, whose entries beyond kmax stay unset and unread; the inverses rinv an array of arrays, for the array on the stack; the loop over the classes by a function of its own (Ratpoints.Sieve.classRows); natural numbers for the longs.)

- find_points.c:1164-1182: modelled by Ratpoints.Sieve.classRows (deviation of Ratpoints.Sieve.classRows: a function of its own for the loop over the classes of class_offsets (Ratpoints.Sieve.classOffsets); the classes and the array of the offsets returned, for the array of the classes and the array of the offsets that class_offsets writes through its pointer arguments; the table of the rows by their keys an array of options of positions, for the array row_of of 2 ^ RP_NUM_STRIDES pointers into the offsets, none for NULL, and the row of a class its position in the array of the offsets, for the pointer row; the inverses rinv an array of arrays, for the array of RP_NUM_STRIDES longs for each modulus on the stack; natural numbers for the longs, the product a_0 rinv[n][k] a natural number for the long that RP_MULMOD converts to an unsigned long, below 2 ^ 63 (Ratpoints.mul_lt_two_pow_63); the reciprocal and the bias read from the sieve entry, as the C code reads them.)
- find_points.c:1183-1183: modelled by Ratpoints.Sieve.classOffsets (deviation of Ratpoints.Sieve.classOffsets: the classes, the entries and the array of the offsets returned, for the arrays that class_offsets writes through its pointer arguments; the number np of the moduli the number of the sieve entries es, for the run all sp3_max of them (Ratpoints.sieving), with the reserve of further primes that the correction of the moduli may take into the second stage during the run (Ratpoints.Sieve.adaptPrimes); the inverses of 2 ^ k an array of kmax + 1 entries, for the array dinv of RP_NUM_STRIDES longs of the sieve entry, whose entries beyond kmax stay unset and unread; the inverses rinv an array of arrays, for the array on the stack; the loop over the classes by a function of its own (Ratpoints.Sieve.classRows); natural numbers for the longs.)
- find_points.c:1194-1198: modelled by Ratpoints.ModulusCandidate.keyLe (deviation of Ratpoints.ModulusCandidate.keyLe: the test of the merge sort, whether the comparison is at most 0, for the comparison of the C code, which returns 1, -1 or 0.)
- find_points.c:1202-1206: modelled by Ratpoints.ModulusCandidate.rLe (deviation of Ratpoints.ModulusCandidate.rLe: the test of the merge sort, whether the comparison is at most 0, for the comparison of the C code, which returns 1, -1 or 0.)
- find_points.c:1220-1235: modelled by Ratpoints.primeCost
- find_points.c:1241-1247: modelled by Ratpoints.primeKey (deviation of Ratpoints.primeKey: log the function of the C library of the platform that the Lean runtime binds (Float.log), on glibc an older symbol version of the routine that ratpoints binds, which gives the same values on every argument that occurs here, and which glibc does not round correctly and chooses by the processor, so that the value is that of the C program on the same machine.)
- find_points.c:1256-1270: modelled by Ratpoints.phase1Key (deviation of Ratpoints.phase1Key: log the function of the C library of the platform that the Lean runtime binds (Float.log), on glibc an older symbol version of the routine that ratpoints binds, which gives the same values on every argument that occurs here, and which glibc does not round correctly and chooses by the processor, so that the value is that of the C program on the same machine; the candidate returned with its key and cost, for the fields of the record that phase_1_key sets; Ratpoints.rbaPack for RBA_PACK and 8 Ratpoints.rbaPack for the size of a bit array in bytes.)

- find_points.c:1285-1318: modelled by Ratpoints.checkCost (deviation of Ratpoints.checkCost: the coefficients, the degree and the height bound as arguments, for the fields of the arguments that check_cost reads; log the function of the C library of the platform that the Lean runtime binds (Float.log), on glibc an older symbol version of the routine that ratpoints binds, which gives the same values on every argument that occurs here, and which glibc does not round correctly and chooses by the processor, so that the value is that of the C program on the same machine; log 2 computed at run time, where gcc folds log(2.0) into the same double; ceil the function of the C library of the platform (Float.ceil), which is exact, where gcc compiles an instruction; the bit length of |c_k| (Nat.log2 + 1) for mpz_sizeinbase(c_k, 2), which it is for c_k ≠ 0; 64 for LONG_LENGTH.)
- find_points.c:1354-1354: modelled by Ratpoints.adaptWords
- find_points.c:1355-1355: modelled by Ratpoints.adaptArrays
- find_points.c:1356-1356: modelled by Ratpoints.adaptBits
- find_points.c:1363-1367: modelled by Ratpoints.Sieve.RunCounts (deviation of Ratpoints.Sieve.RunCounts: the six counters of the sieving thread extended (Ratpoints.Sieve.Counters), for the six fields; a UInt64 for the unsigned long n_words_2 and a natural number for the long sp2.)
- find_points.c:1379-1381: modelled by Ratpoints.adaptLag
- find_points.c:1383-1388: modelled by Ratpoints.Sieve.adaptPrimes (deviation of Ratpoints.Sieve.adaptPrimes: the fields of the arguments that adapt_primes reads passed as the options, the result of the choice and the entries, and those that it writes, sp2, sp3 and adapt_at, returned; the doubles of the options by their bits; Floats for the doubles, in the order of the C code; integers for the longs adapt and sp3_extra and natural numbers for the others; no output under RP_PRIME_STATS.)
- find_points.c:1389-1389: modelled by Ratpoints.Sieve.adaptRates (deviation of Ratpoints.Sieve.adaptRates: a function of its own for this part of adapt_primes, its returns the result none; Floats for the doubles, in the order of the C code, the conversions of the UInt64 counters to Float as the C code converts the unsigned longs to doubles, and the words since the restart the difference of the UInt64s, wrapping, converted, as the C code computes it; the densities read from the entries, for sieve_list[n]->r; natural numbers for the longs, so that sp1 ≤ 0 is sp1 = 0.)
- find_points.c:1390-1398: modelled by Ratpoints.Sieve.adaptPrimes (deviation of Ratpoints.Sieve.adaptPrimes: the fields of the arguments that adapt_primes reads passed as the options, the result of the choice and the entries, and those that it writes, sp2, sp3 and adapt_at, returned; the doubles of the options by their bits; Floats for the doubles, in the order of the C code; integers for the longs adapt and sp3_extra and natural numbers for the others; no output under RP_PRIME_STATS.)
- find_points.c:1400-1420: modelled by Ratpoints.Sieve.adaptRates (deviation of Ratpoints.Sieve.adaptRates: a function of its own for this part of adapt_primes, its returns the result none; Floats for the doubles, in the order of the C code, the conversions of the UInt64 counters to Float as the C code converts the unsigned longs to doubles, and the words since the restart the difference of the UInt64s, wrapping, converted, as the C code computes it; the densities read from the entries, for sieve_list[n]->r; natural numbers for the longs, so that sp1 ≤ 0 is sp1 = 0.)

- find_points.c:1421-1438: modelled by Ratpoints.Sieve.adaptPrimes (deviation of Ratpoints.Sieve.adaptPrimes: the fields of the arguments that adapt_primes reads passed as the options, the result of the choice and the entries, and those that it writes, sp2, sp3 and adapt_at, returned; the doubles of the options by their bits; Floats for the doubles, in the order of the C code; integers for the longs adapt and sp3_extra and natural numbers for the others; no output under RP_PRIME_STATS.)
- find_points.c:1439-1448: modelled by Ratpoints.Sieve.adaptSecond (deviation of Ratpoints.Sieve.adaptSecond: a function of its own for this part of adapt_primes; Floats for the doubles, in the order of the C code; natural numbers for the longs.)
- find_points.c:1449-1457: modelled by Ratpoints.Sieve.adaptUp (deviation of Ratpoints.Sieve.adaptUp: a function of its own for this loop of adapt_primes; the entries an array, for sieve_list up to sp3_max, its size; Floats for the doubles, in the order of the C code; natural numbers for the longs.)
- find_points.c:1458-1459: modelled by Ratpoints.Sieve.adaptSecond (deviation of Ratpoints.Sieve.adaptSecond: a function of its own for this part of adapt_primes; Floats for the doubles, in the order of the C code; natural numbers for the longs.)
- find_points.c:1460-1472: modelled by Ratpoints.Sieve.adaptDown (deviation of Ratpoints.Sieve.adaptDown: a function of its own for this loop of adapt_primes; the entries an array, for sieve_list; Floats for the doubles, in the order of the C code; natural numbers for the longs, n stepping down from sp2 - 1 while it is above sp1.)
- find_points.c:1473-1476: modelled by Ratpoints.Sieve.adaptSecond (deviation of Ratpoints.Sieve.adaptSecond: a function of its own for this part of adapt_primes; Floats for the doubles, in the order of the C code; natural numbers for the longs.)
- find_points.c:1477-1486: modelled by Ratpoints.Sieve.adaptPrimes (deviation of Ratpoints.Sieve.adaptPrimes: the fields of the arguments that adapt_primes reads passed as the options, the result of the choice and the entries, and those that it writes, sp2, sp3 and adapt_at, returned; the doubles of the options by their bits; Floats for the doubles, in the order of the C code; integers for the longs adapt and sp3_extra and natural numbers for the others; no output under RP_PRIME_STATS.)
- find_points.c:1487-1505: modelled by Ratpoints.Sieve.adaptThird (deviation of Ratpoints.Sieve.adaptThird: a function of its own for this part of adapt_primes, the fields sp3_per_denom (its double read by its bits by the caller) and check_rel of the arguments that it reads and the local variables level, chance, r2 and s of adapt_primes passed as arguments; the end of the third stage returned, for the field sp3 of the arguments that it writes; Floats for the doubles, in the order of the C code; natural numbers for the longs.)
- find_points.c:1506-1509: modelled by Ratpoints.Sieve.adaptPrimes (deviation of Ratpoints.Sieve.adaptPrimes: the fields of the arguments that adapt_primes reads passed as the options, the result of the choice and the entries, and those that it writes, sp2, sp3 and adapt_at, returned; the doubles of the options by their bits; Floats for the doubles, in the order of the C code; integers for the longs adapt and sp3_extra and natural numbers for the others; no output under RP_PRIME_STATS.)
- find_points.c:1510-1514: excluded (instrumentation): RP_PRIME_STATS in adapt_primes, a development probe that prints each correction of the moduli on stderr

- find_points.c:1515-1515: modelled by Ratpoints.Sieve.adaptPrimes (deviation of Ratpoints.Sieve.adaptPrimes: the fields of the arguments that adapt_primes reads passed as the options, the result of the choice and the entries, and those that it writes, sp2, sp3 and adapt_at, returned; the doubles of the options by their bits; Floats for the doubles, in the order of the C code; integers for the longs adapt and sp3_extra and natural numbers for the others; no output under RP_PRIME_STATS.)
- find_points.c:1539-1561: modelled by Ratpoints.downstreamFactor (deviation of Ratpoints.downstreamFactor: pow the function of the C library of the platform that the Lean runtime binds (Float.pow), on glibc an older symbol version of the routine that ratpoints binds, which gives the same values on every argument that occurs here, and which glibc does not round correctly and chooses by the processor, so that the value is that of the C program on the same machine; natural numbers for the long positions, the comparisons with the candidates left after n made on integers.)
- find_points.c:1583-1589: modelled by Ratpoints.phase1Wants
- find_points.c:1598-1609: modelled by Ratpoints.primesForPhase1
- find_points.c:1619-1626: modelled by Ratpoints.phase2Offset (deviation of Ratpoints.phase2Offset: natural numbers for the longs, which are not negative here; the conversion of e + 0.5 to a long, which truncates, by Float.toUInt64, which truncates e + 0.5 < most as well.)
- find_points.c:1639-1650: modelled by Ratpoints.Sieve.barrett (deviation of Ratpoints.Sieve.barrett: natural numbers for the unsigned longs, the product u m exact as the 128-bit product of the C code, and the difference exact, which does not wrap since the quotient times p is at most u for m ≤ (2 ^ 64 - 1) / p; the obligation that u is below 2 ^ 64, which the C type gives, a hypothesis of the lemma that the result is u % p (Ratpoints.Sieve.barrett_eq); the form with the 128-bit product, which the modelled builds compile (the compiler defines __SIZEOF_INT128__, and RP_NO_INT128 is not set), for the division u % p of the other builds, which gives the same remainder.)
- find_points.c:1652-1655: modelled by Ratpoints.hornerSteps (deviation of Ratpoints.hornerSteps: computed from Ratpoints.maxBitsInPrime as the macro computes it from RATPOINTS_MAX_BITS_IN_PRIME, with 64 for LONG_LENGTH.)
- find_points.c:1683-1697: modelled by Ratpoints.Sieve.examinePrime? (deviation of Ratpoints.Sieve.examinePrime?: the prime p itself, for its index pn in the table prime; the table a fresh array of p + 1 entries, for the integers that examine_prime takes from the buffer int_next and hands back through is_f_square_p, and the coefficients a fresh array returned with the entry, for the array coeffs_mod_p of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which Ratpoints.Sieve.hornerBarrett chooses for each residue; the exit -1 as none, and the codes 0, 1 and 2 returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code 0 too, where examine_prime makes none, and the density 0.0 for that code, which examine_prime does not compute; the fields p, mask and nf of the record prec_entry of the ranking, which examine_prime sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (Ratpoints.ModulusCandidate.ofPrime); the entry is a fresh value, for the space that examine_prime takes from the buffer of the sieve entries (se_next), and returned, where examine_prime stores it in the field ssp of prec_entry; it keeps only p, the table and the inverses (Ratpoints.Sieve.Entry), and no rows: the table of rows of the prime, with the row

of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`), of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1698-1701`: excluded (instrumentation): the message under DEBUG at the start of examine_prime

- `find_points.c:1703-1723`: modelled by `Ratpoints.Sieve.examinePrime?` (deviation of `Ratpoints.Sieve.examinePrime?`: the prime `p` itself, for its index `pn` in the table `prime`; the table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask` and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`), of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1724-1724`: modelled by `Ratpoints.Sieve.hornerBarrett` (deviation of `Ratpoints.Sieve.hornerBarrett`: the choice between the two branches made for each residue, where `examine_prime` makes it once, around its loop over the residues; natural numbers for the `unsigned long` accumulator, with the obligation that it stays below `2 ^ 64` stated as a lemma (`Ratpoints.Sieve.HornerInvariant.lt_two_pow_64`); the steps counted up from `0` to `d - 1`, for the index `n` that counts down.)

- `find_points.c:1725-1725`: modelled by `Ratpoints.Sieve.examinePrime?` (deviation of `Ratpoints.Sieve.examinePrime?`: the prime `p` itself, for its index `pn` in the table `prime`; the table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the

buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask` and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`), of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1726-1731`: modelled by `Ratpoints.Sieve.hornerBarrett` (deviation of `Ratpoints.Sieve.hornerBarrett`: the choice between the two branches made for each residue, where `examine_prime` makes it once, around its loop over the residues; natural numbers for the `unsigned long` accumulator, with the obligation that it stays below `2 ^ 64` stated as a lemma (`Ratpoints.Sieve.HornerInvariant.lt_two_pow_64`); the steps counted up from `0` to `d - 1`, for the index `n` that counts down.)

- `find_points.c:1732-1734`: modelled by `Ratpoints.Sieve.examinePrime?` (deviation of `Ratpoints.Sieve.examinePrime?`: the prime `p` itself, for its index `pn` in the table `prime`; the table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask` and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by

the end of the choice (Ratpoints.choiceEnd, with Ratpoints.Sieve.SieveEntry.addTables); the fields init, nf and pw that examine_prime sets (lines 1800 and 1817) and its reciprocal magic (line 1808) are those of the sieve entry of the prime (Ratpoints.Sieve.SieveEntry), of which the prime is the single factor, made by the entries that the choice of the moduli makes (Ratpoints.primeEntry), which set the reciprocal there, and so are the bias bias, the inverse rbainv of Ratpoints.rbaLength, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- find_points.c:1735-1735: modelled by Ratpoints.Sieve.hornerBarrett (deviation of Ratpoints.Sieve.hornerBarrett: the choice between the two branches made for each residue, where examine_prime makes it once, around its loop over the residues; natural numbers for the unsigned long accumulator, with the obligation that it stays below 2 ^ 64 stated as a lemma (Ratpoints.Sieve.HornerInvariant.lt_two_pow_64); the steps counted up from 0 to d - 1, for the index n that counts down.)

- find_points.c:1736-1736: modelled by Ratpoints.Sieve.examinePrime? (deviation of Ratpoints.Sieve.examinePrime?: the prime p itself, for its index pn in the table prime; the table a fresh array of p + 1 entries, for the integers that examine_prime takes from the buffer int_next and hands back through is_f_square_p, and the coefficients a fresh array returned with the entry, for the array coeffs_mod_p of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which Ratpoints.Sieve.hornerBarrett chooses for each residue; the exit -1 as none, and the codes 0, 1 and 2 returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code 0 too, where examine_prime makes none, and the density 0.0 for that code, which examine_prime does not compute; the fields p, mask and nf of the record prec_entry of the ranking, which examine_prime sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (Ratpoints.ModulusCandidate.ofPrime); the entry is a fresh value, for the space that examine_prime takes from the buffer of the sieve entries (se_next), and returned, where examine_prime stores it in the field ssp of prec_entry; it keeps only p, the table and the inverses (Ratpoints.Sieve.Entry), and no rows: the table of rows of the prime, with the row of 0 from the build and the others absent (lines 1820-1821, Ratpoints.Sieve.primeRows), is made in the store of the run by the loop of the tables when the prime says something (Ratpoints.examineInLoop), and for a prime that the third stage takes into the pool by the end of the choice (Ratpoints.choiceEnd, with Ratpoints.Sieve.SieveEntry.addTables); the fields init, nf and pw that examine_prime sets (lines 1800 and 1817) and its reciprocal magic (line 1808) are those of the sieve entry of the prime (Ratpoints.Sieve.SieveEntry), of which the prime is the single factor, made by the entries that the choice of the moduli makes (Ratpoints.primeEntry), which set the reciprocal there, and so are the bias bias, the inverse rbainv of Ratpoints.rbaLength, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- find_points.c:1737-1744: modelled by Ratpoints.Sieve.hornerBarrett (deviation of Ratpoints.Sieve.hornerBarrett: the choice between the two branches made for each residue, where examine_prime makes it once, around its loop over the residues; natural numbers for the unsigned long accumulator, with the obligation that it stays below 2 ^ 64 stated as a lemma (Ratpoints.Sieve.HornerInvariant.lt_two_pow_64); the steps counted up from 0 to d - 1, for the index n that counts down.)

- find_points.c:1745-1749: modelled by Ratpoints.Sieve.examinePrime? (deviation of Ratpoints.Sieve.examinePrime?: the prime p itself, for its index pn in the table prime; the

table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask` and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`), of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1751-1759`: excluded (instrumentation): the table of examine_prime with its last entry, printed under DEBUG

- `find_points.c:1761-1781`: modelled by `Ratpoints.Sieve.examinePrime?` (deviation of `Ratpoints.Sieve.examinePrime?`: the prime `p` itself, for its index `pn` in the table `prime`; the table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask` and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`),

of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1782-1785`: modelled by `Ratpoints.ModulusCandidate.ofPrime` (deviation of `Ratpoints.ModulusCandidate.ofPrime`: the record returned, for the record `prec_entry` that `examine_prime` fills, its key and cost `0.0` until the caller sets them, where the C code leaves them as they are; the count `-1` read from the code, which is `2` exactly when `np ≥ p`.)

- `find_points.c:1795-1819`: modelled by `Ratpoints.Sieve.examinePrime?` (deviation of `Ratpoints.Sieve.examinePrime?`: the prime `p` itself, for its index `pn` in the table `prime`; the table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask` and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`), of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1820-1821`: modelled by `Ratpoints.Sieve.primeRows` (deviation of `Ratpoints.Sieve.primeRows`: the table returned, for the array `sieve` of the entry, which `examine_prime` fills; the row of `0` the words of the pattern, for the pointer to them.)

- `find_points.c:1822-1826`: modelled by `Ratpoints.Sieve.examinePrime?` (deviation of `Ratpoints.Sieve.examinePrime?`: the prime `p` itself, for its index `pn` in the table `prime`; the table a fresh array of `p + 1` entries, for the integers that `examine_prime` takes from the buffer `int_next` and hands back through `is_f_square_p`, and the coefficients a fresh array returned with the entry, for the array `coeffs_mod_p` of its caller, which it fills; one loop over the residues, for the two loops of the two branches of the schedule of Horner's rule, between which `Ratpoints.Sieve.hornerBarrett` chooses for each residue; the exit `-1` as `none`, and the codes `0`, `1` and `2` returned with the entry, the coefficients and the density, as natural numbers, the entry made and returned for the code `0` too, where `examine_prime` makes none, and the density `0.0` for that code, which `examine_prime` does not compute; the fields `p`, `mask`

and `nf` of the record `prec_entry` of the ranking, which `examine_prime` sets with the density (lines 1782-1785), made by the caller from the table, the position of the prime and the code (`Ratpoints.ModulusCandidate.ofPrime`); the entry is a fresh value, for the space that `examine_prime` takes from the buffer of the sieve entries (`se_next`), and returned, where `examine_prime` stores it in the field `ssp` of `prec_entry`; it keeps only `p`, the table and the inverses (`Ratpoints.Sieve.Entry`), and no rows: the table of rows of the prime, with the row of `0` from the build and the others absent (lines 1820-1821, `Ratpoints.Sieve.primeRows`), is made in the store of the run by the loop of the tables when the prime says something (`Ratpoints.examineInLoop`), and for a prime that the third stage takes into the pool by the end of the choice (`Ratpoints.choiceEnd`, with `Ratpoints.Sieve.SieveEntry.addTables`); the fields `init`, `nf` and `pw` that `examine_prime` sets (lines 1800 and 1817) and its reciprocal `magic` (line 1808) are those of the sieve entry of the prime (`Ratpoints.Sieve.SieveEntry`), of which the prime is the single factor, made by the entries that the choice of the moduli makes (`Ratpoints.primeEntry`), which set the reciprocal there, and so are the bias `bias`, the inverse `rbainv` of `Ratpoints.rbaLength`, the density and the count of factors (lines 1811 and 1815-1817), which they set there with the formulas of these lines.)

- `find_points.c:1836-1854`: modelled by `Ratpoints.numeratorsFor` (deviation of `Ratpoints.numeratorsFor`: the search intervals as an argument, for the fields `domain` and `num_inter` of the arguments, their ends doubles by their bits (`Ratpoints.Double.toFloat`); the sum and the three counts returned, the counts as doubles, as the C code hands them back through its pointers.)

- `find_points.c:1863-1876`: modelled by `Ratpoints.clippedPadding` (deviation of `Ratpoints.clippedPadding`: the two sums passed and returned, for the pointers; integers for the `long`s, the arithmetic shifts as divisions that round down and the masks by `RBA_LENGTH - 1` as Euclidean remainders.)

- `find_points.c:1917-1917`: modelled by `Ratpoints.runShapeSamples`

- `find_points.c:1921-1922`: modelled by `Ratpoints.invStride` (deviation of `Ratpoints.invStride`: an array of the seven doubles of the table, which `rp_inv_stride` sizes by `RP_NUM_STRIDES`, read by `getD` with `0.0`, the value of the entries beyond the seven initializers when `RP_NUM_STRIDES` is larger; the stride of a class is at most `2 ^ 6`.)

- `find_points.c:1926-1937`: modelled by `Ratpoints.forbiddenFraction` (deviation of `Ratpoints.forbiddenFraction`: natural numbers for the `long` and the `unsigned long` mask; `64` for `LONG_LENGTH`.)

- `find_points.c:1939-1944`: modelled by `Ratpoints.runShape` (deviation of `Ratpoints.runShape`: the switch `checkDenom` for the bit `RATPOINTS_CHECK_DENOM` of the flags, and the forbidden divisors as lists, for the arrays with their counts `fba` and `fdc`.)

- `find_points.c:1945-1957`: modelled by `Ratpoints.ShapeAcc` (deviation of `Ratpoints.ShapeAcc`: a structure for the local variables, with their initial values.)

- `find_points.c:1959-1959`: modelled by `Ratpoints.runShape` (deviation of `Ratpoints.runShape`: the switch `checkDenom` for the bit `RATPOINTS_CHECK_DENOM` of the flags, and the forbidden divisors as lists, for the arrays with their counts `fba` and `fdc`.)

- `find_points.c:1960-1962`: modelled by `Ratpoints.shapeSquares` (deviation of `Ratpoints.shapeSquares`: `ceil` and `floor` the functions of the C library of the platform (`Float.ceil`, `Float.floor`), which are exact, where gcc compiles instructions.)

- find_points.c:1964-1973: modelled by `Ratpoints.sampleSteps` (deviation of `Ratpoints.sampleSteps`: one function for the samples of the squares and of every denominator, which the C code writes out twice.)
- find_points.c:1976-1979: modelled by `Ratpoints.shapeSquares` (deviation of `Ratpoints.shapeSquares`: `ceil` and `floor` the functions of the C library of the platform (`Float.ceil`, `Float.floor`), which are exact, where gcc compiles instructions.)
- find_points.c:1980-1981: modelled by `Ratpoints.addClass` (deviation of `Ratpoints.addClass`: one function for the three places of the C code.)
- find_points.c:1982-1985: modelled by `Ratpoints.shapeSquares` (deviation of `Ratpoints.shapeSquares`: `ceil` and `floor` the functions of the C library of the platform (`Float.ceil`, `Float.floor`), which are exact, where gcc compiles instructions.)
- find_points.c:1986-1986: modelled by `Ratpoints.runShape` (deviation of `Ratpoints.runShape`: the switch `checkDenom` for the bit `RATPOINTS_CHECK_DENOM` of the flags, and the forbidden divisors as lists, for the arrays with their counts `fba` and `fdc`.)
- find_points.c:1987-1992: modelled by `Ratpoints.shapeSquares1` (deviation of `Ratpoints.shapeSquares1`: the divisors as a list, for the zero-terminated array `args->divisors`.)
- find_points.c:1993-2009: modelled by `Ratpoints.sampleDivisor` (deviation of `Ratpoints.sampleDivisor`: `ceil` and `floor` the functions of the C library of the platform (`Float.ceil`, `Float.floor`), which are exact, where gcc compiles instructions.)
- find_points.c:2010-2016: modelled by `Ratpoints.shapeSquares1` (deviation of `Ratpoints.shapeSquares1`: the divisors as a list, for the zero-terminated array `args->divisors`.)
- find_points.c:2017-2018: modelled by `Ratpoints.addClass` (deviation of `Ratpoints.addClass`: one function for the three places of the C code.)
- find_points.c:2019-2023: modelled by `Ratpoints.shapeSquares1` (deviation of `Ratpoints.shapeSquares1`: the divisors as a list, for the zero-terminated array `args->divisors`.)
- find_points.c:2024-2024: modelled by `Ratpoints.runShape` (deviation of `Ratpoints.runShape`: the switch `checkDenom` for the bit `RATPOINTS_CHECK_DENOM` of the flags, and the forbidden divisors as lists, for the arrays with their counts `fba` and `fdc`.)
- find_points.c:2025-2027: modelled by `Ratpoints.shapeAll` (deviation of `Ratpoints.shapeAll`: the set bits of the word visited by the lowest set bit (`Ratpoints.Sieve.ctz`), as the C code visits them with `RP_CTZL`.)
- find_points.c:2028-2037: modelled by `Ratpoints.sampleSteps` (deviation of `Ratpoints.sampleSteps`: one function for the samples of the squares and of every denominator, which the C code writes out twice.)
- find_points.c:2038-2044: modelled by `Ratpoints.shapeAll` (deviation of `Ratpoints.shapeAll`: the set bits of the word visited by the lowest set bit (`Ratpoints.Sieve.ctz`), as the C code visits them with `RP_CTZL`.)
- find_points.c:2045-2046: modelled by `Ratpoints.addClass` (deviation of `Ratpoints.addClass`: one function for the three places of the C code.)

- find_points.c:2047-2050: modelled by Ratpoints.shapeAll (deviation of Ratpoints.shapeAll: the set bits of the word visited by the lowest set bit (Ratpoints.Sieve.ctz), as the C code visits them with RP_CTZL.)
- find_points.c:2051-2065: modelled by Ratpoints.keepChecked (deviation of Ratpoints.keepChecked: the forbidden divisors as lists, for the arrays of the arguments with their counts fba and fdc.)
- find_points.c:2066-2066: modelled by Ratpoints.shapeAll (deviation of Ratpoints.shapeAll: the set bits of the word visited by the lowest set bit (Ratpoints.Sieve.ctz), as the C code visits them with RP_CTZL.)
- find_points.c:2074-2105: modelled by Ratpoints.shapeEstimate (deviation of Ratpoints.shapeEstimate: the estimate returned as a structure, for the four pointers of run_shape; ceil the function of the C library of the platform (Float.ceil), which is exact, where gcc compiles an instruction; 64 for LONG_LENGTH.)
- find_points.c:2106-2106: modelled by Ratpoints.runShape (deviation of Ratpoints.runShape: the switch checkDenom for the bit RATPOINTS_CHECK_DENOM of the flags, and the forbidden divisors as lists, for the arrays with their counts fba and fdc.)
- find_points.c:2128-2132: modelled by Ratpoints.bpwAdd (deviation of Ratpoints.bpwAdd: the two sums passed and returned, for the pointers; the number of the set bits of the word a loop over its bits (Ratpoints.popcount), for gcc's builtin __builtin_popcountl.)
- find_points.c:2134-2142: modelled by Ratpoints.meanBitsPerWord (deviation of Ratpoints.meanBitsPerWord: the set bits of the word visited by the lowest set bit (Ratpoints.Sieve.ctz), as the C code visits them with RP_CTZL.)
- find_points.c:2143-2151: modelled by Ratpoints.bpwSquares1
- find_points.c:2152-2159: modelled by Ratpoints.meanBitsPerWord (deviation of Ratpoints.meanBitsPerWord: the set bits of the word visited by the lowest set bit (Ratpoints.Sieve.ctz), as the C code visits them with RP_CTZL.)
- find_points.c:2192-2197: modelled by Ratpoints.Sieve.forbiddenValuations (deviation of Ratpoints.Sieve.forbiddenValuations: the prime p itself, for its index pn in the table prime, and the squares modulo p by Ratpoints.Sieve.squaresMod, for the row of p in the table squares of the build; an array for the pointer coeffs_mod_p; a natural number for the unsigned long mask, without the cap m < LONG_LENGTH - 1 of the loop, which never binds for a long b_high and an odd prime, the loop ending before m = 40 (Ratpoints.Sieve.lt_forty_of_pow_le), and for p ≤ 1, which the C code never examines, the loop, once entered, does not end; natural numbers for the longs.)
- find_points.c:2198-2200: modelled by Ratpoints.Sieve.unitParts (deviation of Ratpoints.Sieve.unitParts: an array of options of pairs, for the arrays present, w and r on the stack; natural numbers for the longs.)
- find_points.c:2201-2203: modelled by Ratpoints.Sieve.forbiddenValuations (deviation of Ratpoints.Sieve.forbiddenValuations: the prime p itself, for its index pn in the table prime, and the squares modulo p by Ratpoints.Sieve.squaresMod, for the row of p in the table squares of the build; an array for the pointer coeffs_mod_p; a natural number for the unsigned long mask, without the cap m < LONG_LENGTH - 1 of the loop,

which never binds for a `long b_high` and an odd prime, the loop ending before `m = 40` (`Ratpoints.Sieve.lt_forty_of_pow_le`), and for `p ≤ 1`, which the C code never examines, the loop, once entered, does not end; natural numbers for the `long`s.)

- `find_points.c:2204-2215`: modelled by `Ratpoints.Sieve.unitParts` (deviation of `Ratpoints.Sieve.unitParts`: an array of options of pairs, for the arrays `present`, `w` and `r` on the stack; natural numbers for the `long`s.)

- `find_points.c:2216-2217`: modelled by `Ratpoints.Sieve.forbiddenValuations` (deviation of `Ratpoints.Sieve.forbiddenValuations`: the prime `p` itself, for its index `pn` in the table `prime`, and the squares modulo `p` by `Ratpoints.Sieve.squaresMod`, for the row of `p` in the table `squares` of the build; an array for the pointer `coeffs_mod_p`; a natural number for the `unsigned long` mask, without the cap `m < LONG_LENGTH - 1` of the loop, which never binds for a `long b_high` and an odd prime, the loop ending before `m = 40` (`Ratpoints.Sieve.lt_forty_of_pow_le`), and for `p ≤ 1`, which the C code never examines, the loop, once entered, does not end; natural numbers for the `long`s.)

- `find_points.c:2218-2228`: modelled by `Ratpoints.Sieve.forbiddenAt` (deviation of `Ratpoints.Sieve.forbiddenAt`: the valuations and unit parts of the coefficients as the array of options `us` (`Ratpoints.Sieve.unitParts`), for the arrays `present`, `w` and `r`, its size `d + 1` for the bound `degree` of the loop, and the squares modulo `p` as the argument `sq` (`Ratpoints.Sieve.squaresMod`), for the row `squares[pn]` of the table of the build; the test returned, for the bit that the caller sets; remainders by 2 for the masks `& 1`; natural numbers for the `long`s.)

- `find_points.c:2229-2234`: modelled by `Ratpoints.Sieve.forbiddenValuations` (deviation of `Ratpoints.Sieve.forbiddenValuations`: the prime `p` itself, for its index `pn` in the table `prime`, and the squares modulo `p` by `Ratpoints.Sieve.squaresMod`, for the row of `p` in the table `squares` of the build; an array for the pointer `coeffs_mod_p`; a natural number for the `unsigned long` mask, without the cap `m < LONG_LENGTH - 1` of the loop, which never binds for a `long b_high` and an odd prime, the loop ending before `m = 40` (`Ratpoints.Sieve.lt_forty_of_pow_le`), and for `p ≤ 1`, which the C code never examines, the loop, once entered, does not end; natural numbers for the `long`s.)

- `find_points.c:2241-2252`: modelled by `Ratpoints.Sieve.modinv` (deviation of `Ratpoints.Sieve.modinv`: natural numbers for `x`, `m` and the remainders, which are never negative for `x ≥ 0`, so that the correction of a negative `x % m` has no counterpart; integers for the `long` cofactors, which stay between `-m` and `m` (`Ratpoints.Sieve.ModinvInvariant.natAbs_add_mul_le`); the result a natural number.)

- `find_points.c:2267-2282`: modelled by `Ratpoints.Sieve.examinePower` (deviation of `Ratpoints.Sieve.examinePower`: the tables returned as a fresh value, for the entry `pw` of the buffer of the prime powers that `examine_power` fills; `m` as `p ^ e`, for the product of `e` factors `p`; the bit arrays `sq`, `fsq` and `gsq` fresh arrays of `Ratpoints.modWords` words, which the loop at line 2282 clears, for the local array that the function leaves unset and the arrays of the entry `pw`; natural numbers for the `long`s and `unsigned long`s, the variable `t` for the multiples of `p`, where the C code reuses `x`, and the steps `v x + c` of Horner's rule, with `v`, `x` and `c` below `m`, staying below `m ^ 2`, with the obligation stated as a lemma (`Ratpoints.mul_add_lt_two_pow_64_of_lt_maxPrimeEven`); the code that `examine_power` returns read from the density by `Ratpoints.Sieve.Power.useful`.)

- find_points.c:2283-2284: modelled by Ratpoints.Sieve.setSquareBits (deviation of Ratpoints.Sieve.setSquareBits: the bit array passed and returned, for the local array sq of examine_power; natural numbers for the longs, the squares x x with x < m staying below m ^ 2, with the obligation stated as a lemma (Ratpoints.mul_self_lt_two_pow_63_of_lt_maxPrimeEven).)

- find_points.c:2285-2293: modelled by Ratpoints.Sieve.examinePower (deviation of Ratpoints.Sieve.examinePower: the tables returned as a fresh value, for the entry pw of the buffer of the prime powers that examine_power fills; m as p ^ e, for the product of e factors p; the bit arrays sq, fsq and gsq fresh arrays of Ratpoints.modWords words, which the loop at line 2282 clears, for the local array that the function leaves unset and the arrays of the entry pw; natural numbers for the longs and unsigned longs, the variable t for the multiples of p, where the C code reuses x, and the steps v x + c of Horner's rule, with v, x and c below m, staying below m ^ 2, with the obligation stated as a lemma (Ratpoints.mul_add_lt_two_pow_64_of_lt_maxPrimeEven); the code that examine_power returns read from the density by Ratpoints.Sieve.Power.useful.)

- find_points.c:2294-2300: modelled by Ratpoints.Sieve.setRevBits (deviation of Ratpoints.Sieve.setRevBits: the bit array passed and returned, for the array of the entry pw; the variable t for the multiples of p, where the C code reuses x; a while loop for the loop for with the step p; natural numbers for the longs and unsigned longs.)

- find_points.c:2301-2301: modelled by Ratpoints.Sieve.examinePower (deviation of Ratpoints.Sieve.examinePower: the tables returned as a fresh value, for the entry pw of the buffer of the prime powers that examine_power fills; m as p ^ e, for the product of e factors p; the bit arrays sq, fsq and gsq fresh arrays of Ratpoints.modWords words, which the loop at line 2282 clears, for the local array that the function leaves unset and the arrays of the entry pw; natural numbers for the longs and unsigned longs, the variable t for the multiples of p, where the C code reuses x, and the steps v x + c of Horner's rule, with v, x and c below m, staying below m ^ 2, with the obligation stated as a lemma (Ratpoints.mul_add_lt_two_pow_64_of_lt_maxPrimeEven); the code that examine_power returns read from the density by Ratpoints.Sieve.Power.useful.)

- find_points.c:2302-2303: modelled by Ratpoints.Sieve.unitInverses (deviation of Ratpoints.Sieve.unitInverses: the inverses a fresh array, returned, for the array inv of the entry pw that examine_power fills; natural numbers for the longs and for the unsigned short inverses, which are below m, with the obligation stated as a lemma (Ratpoints.Sieve.getElem!_inv_run_examinePowe

- find_points.c:2304-2304: excluded (other): the count np of a prime power stored in its rp_power, which nothing reads

- find_points.c:2306-2329: modelled by Ratpoints.Sieve.powerDensity (deviation of Ratpoints.Sieve.powerDensity: the density returned, for the field r of the record pw that examine_power sets; natural numbers for the longs; the residues t stepped through by a while loop.)

- find_points.c:2330-2330: modelled by Ratpoints.Sieve.Power.useful (deviation of Ratpoints.Sieve.Power.useful: a Boolean read from the density of the record, for the code 1 or 0 that examine_power returns.)

- find_points.c:2331-2331: modelled by Ratpoints.Sieve.examinePower (deviation of Ratpoints.Sieve.examinePower: the tables returned as a fresh value, for the entry pw of the

buffer of the prime powers that examine_power fills; m as p ^ e, for the product of e factors p; the bit arrays sq, fsq and gsq fresh arrays of Ratpoints.modWords words, which the loop at line 2282 clears, for the local array that the function leaves unset and the arrays of the entry pw; natural numbers for the longs and unsigned longs, the variable t for the multiples of p, where the C code reuses x, and the steps v x + c of Horner's rule, with v, x and c below m, staying below m ^ 2, with the obligation stated as a lemma (Ratpoints.mul_add_lt_two_pow_64_of_lt_maxPrimeEven); the code that examine_power returns read from the density by Ratpoints.Sieve.Power.useful.)

- find_points.c:2337-2337: modelled by Ratpoints.modulusTableMax
- find_points.c:2346-2349: modelled by Ratpoints.addModuli (deviation of Ratpoints.addModuli: the pool passed and returned, for the array prec and the count pnp, and the records of the prime powers returned, for the buffer pws of the arguments; the tables and the densities of the primes of the loop of the tables (primeSe) and whether denominators divisible by them occur (pinf) as arrays by their positions; the estimates of the run as arguments, for the fields of the arguments; a while loop over the candidates; natural numbers for the longs; no bound npw_max (num_powers, line 2351), which never binds.)
- find_points.c:2350-2350: modelled by Ratpoints.PowerTable (deviation of Ratpoints.PowerTable: a structure for the buffer pws, the count npw of its records and the array pw_idx of indices with its markers -1 (not examined) and -2 (useless): an array of the records, and a list of the pairs examined with an option, none for -2, a pair not in the list for -1.)
- find_points.c:2351-2351: modelled by Ratpoints.addModuli (deviation of Ratpoints.addModuli: the pool passed and returned, for the array prec and the count pnp, and the records of the prime powers returned, for the buffer pws of the arguments; the tables and the densities of the primes of the loop of the tables (primeSe) and whether denominators divisible by them occur (pinf) as arrays by their positions; the estimates of the run as arguments, for the fields of the arguments; a while loop over the candidates; natural numbers for the longs; no bound npw_max (num_powers, line 2351), which never binds.)
- find_points.c:2352-2354: modelled by Ratpoints.PowerTable (deviation of Ratpoints.PowerTable: a structure for the buffer pws, the count npw of its records and the array pw_idx of indices with its markers -1 (not examined) and -2 (useless): an array of the records, and a list of the pairs examined with an option, none for -2, a pair not in the list for -1.)
- find_points.c:2355-2356: modelled by Ratpoints.addModuli (deviation of Ratpoints.addModuli: the pool passed and returned, for the array prec and the count pnp, and the records of the prime powers returned, for the buffer pws of the arguments; the tables and the densities of the primes of the loop of the tables (primeSe) and whether denominators divisible by them occur (pinf) as arrays by their positions; the estimates of the run as arguments, for the fields of the arguments; a while loop over the candidates; natural numbers for the longs; no bound npw_max (num_powers, line 2351), which never binds.)
- find_points.c:2358-2359: modelled by Ratpoints.PowerTable (deviation of Ratpoints.PowerTable: a structure for the buffer pws, the count npw of its records and the array pw_idx of indices with its markers -1 (not examined) and -2 (useless): an array of the records, and a list of the pairs examined with an option, none for -2, a pair not in the list for -1.)
- find_points.c:2361-2361: modelled by Ratpoints.addModuli (deviation of Ratpoints.addModuli: the pool passed and returned, for the array prec and the count pnp, and the records of the

prime powers returned, for the buffer `pws` of the arguments; the tables and the densities of the primes of the loop of the tables (`primeSe`) and whether denominators divisible by them occur (`pinf`) as arrays by their positions; the estimates of the run as arguments, for the fields of the arguments; a `while` loop over the candidates; natural numbers for the `long`s; no bound `npw_max` (`num_powers`, line 2351), which never binds.)

- `find_points.c:2362-2362`: modelled by `Ratpoints.FacState` (deviation of `Ratpoints.FacState`: a structure for the local variables of the loop over the candidates.)
- `find_points.c:2363-2363`: modelled by `Ratpoints.offerModulus` (deviation of `Ratpoints.offerModulus`: the candidate returned, for the record `prec[pnp]` that `add_moduli` fills, with the key and the cost `0.0`, which the caller sets, and no sieve entry.)
- `find_points.c:2364-2365`: modelled by `Ratpoints.FacState` (deviation of `Ratpoints.FacState`: a structure for the local variables of the loop over the candidates.)
- `find_points.c:2368-2368`: modelled by `Ratpoints.offerModulus` (deviation of `Ratpoints.offerModulus`: the candidate returned, for the record `prec[pnp]` that `add_moduli` fills, with the key and the cost `0.0`, which the caller sets, and no sieve entry.)
- `find_points.c:2370-2380`: modelled by `Ratpoints.factorCandidate` (deviation of `Ratpoints.factorCandidate`: a `while` loop with the position of the prime in the state, for the loop `for` whose variable `pn` the search for the prime below `p ^ 2` advances; natural numbers for the `long`s.)
- `find_points.c:2381-2386`: modelled by `Ratpoints.takeFactor` (deviation of `Ratpoints.takeFactor`: the densities of the primes from the array `primeSe` of the tables and densities of the loop of the tables, for the entries `prime_se` and their fields `r`.)
- `find_points.c:2387-2394`: modelled by `Ratpoints.powerIndex` (deviation of `Ratpoints.powerIndex`: the table passed and returned, for the buffer `pws`, the count `npw` and the array `pw_idx` of `add_moduli`; no test of room in the buffer (line 2390), which never fails, since `add_moduli` examines each prime power at most once and `num_powers` counts them all.)
- `find_points.c:2395-2398`: modelled by `Ratpoints.takeFactor` (deviation of `Ratpoints.takeFactor`: the densities of the primes from the array `primeSe` of the tables and densities of the loop of the tables, for the entries `prime_se` and their fields `r`.)
- `find_points.c:2399-2399`: modelled by `Ratpoints.factorCandidate` (deviation of `Ratpoints.factorCandidate`: a `while` loop with the position of the prime in the state, for the loop `for` whose variable `pn` the search for the prime below `p ^ 2` advances; natural numbers for the `long`s.)
- `find_points.c:2400-2402`: modelled by `Ratpoints.offerModulus` (deviation of `Ratpoints.offerModulus`: the candidate returned, for the record `prec[pnp]` that `add_moduli` fills, with the key and the cost `0.0`, which the caller sets, and no sieve entry.)
- `find_points.c:2403-2403`: modelled by `Ratpoints.addModuli` (deviation of `Ratpoints.addModuli`: the pool passed and returned, for the array `prec` and the count `pnp`, and the records of the prime powers returned, for the buffer `pws` of the arguments; the tables and the densities of the primes of the loop of the tables (`primeSe`) and whether denominators divisible by them occur (`pinf`) as arrays by their positions; the estimates of the run as arguments, for the fields of the arguments; a `while` loop over the candidates; natural numbers for the `long`s; no bound `npw_max` (`num_powers`, line 2351), which never binds.)

- find_points.c:2404-2406: modelled by Ratpoints.offerModulus (deviation of Ratpoints.offerModulus: the candidate returned, for the record prec[pnp] that add_moduli fills, with the key and the cost 0.0, which the caller sets, and no sieve entry.)

- find_points.c:2407-2411: modelled by Ratpoints.addModuli (deviation of Ratpoints.addModuli: the pool passed and returned, for the array prec and the count pnp, and the records of the prime powers returned, for the buffer pws of the arguments; the tables and the densities of the primes of the loop of the tables (primeSe) and whether denominators divisible by them occur (pinf) as arrays by their positions; the estimates of the run as arguments, for the fields of the arguments; a while loop over the candidates; natural numbers for the longs; no bound npw_max (num_powers, line 2351), which never binds.)

- find_points.c:2414-2426: modelled by Ratpoints.makeModulus (deviation of Ratpoints.makeModulus: the entry returned as a fresh value, for the space that make_power and make_modulus take from the buffer of the sieve entries, and the entry of a prime power made for each modulus that needs it, where make_modulus makes it once and keeps it by its index (power_se), with the density of the first modulus that makes it, the density of the prime power in both cases; the factors as a list (Ratpoints.Sieve.Factor), the tables of a prime from the loop of the tables and the records of the prime powers, for the pointers factor to their entries; the tables of rows, which make_power (line 2427) and make_modulus (line 2471) clear, made in the store of the run after the choice (Ratpoints.Sieve.SieveEntry.addTables).)

- find_points.c:2427-2427: modelled by Ratpoints.Sieve.SieveEntry.addTables (deviation of Ratpoints.Sieve.SieveEntry.addTables: the tables of rows made in the store of the run, by the reference to the entry (Ratpoints.Sieve.EntryRef), for the arrays sieve of the entries, after the choice for every entry, the reserve included (Ratpoints.choiceEnd), where make_power and make_modulus make them with the entry; a table of rows made only when its reference is absent, for the test of power_se (lines 2444 and 2466) by which the entry of a prime power is made once, and for the entry of a prime, which examine_prime makes once, with its table of rows, in the loop of the tables (Ratpoints.examineInLoop) or in the third stage (lines 2994-2995), where the model makes the table of a prime of the third stage after the choice, and none for a prime that the third stage examines and passes over, whose table nothing reads.)

- find_points.c:2428-2429: modelled by Ratpoints.makeModulus (deviation of Ratpoints.makeModulus: the entry returned as a fresh value, for the space that make_power and make_modulus take from the buffer of the sieve entries, and the entry of a prime power made for each modulus that needs it, where make_modulus makes it once and keeps it by its index (power_se), with the density of the first modulus that makes it, the density of the prime power in both cases; the factors as a list (Ratpoints.Sieve.Factor), the tables of a prime from the loop of the tables and the records of the prime powers, for the pointers factor to their entries; the tables of rows, which make_power (line 2427) and make_modulus (line 2471) clear, made in the store of the run after the choice (Ratpoints.Sieve.SieveEntry.addTables).)

- find_points.c:2434-2470: modelled by Ratpoints.makeModulus (deviation of Ratpoints.makeModulus: the entry returned as a fresh value, for the space that make_power and make_modulus take from the buffer of the sieve entries, and the entry of a prime power made for each modulus that needs it, where make_modulus makes it once and keeps it by its index (power_se), with the density of the first modulus that makes it, the density of the prime power in both cases; the factors as a list (Ratpoints.Sieve.Factor), the tables of a prime from the loop of the tables and the records of the prime powers, for the pointers factor to their entries; the

tables of rows, which make_power (line 2427) and make_modulus (line 2471) clear, made in the store of the run after the choice (Ratpoints.Sieve.SieveEntry.addTables).)

- find_points.c:2471-2471: modelled by Ratpoints.Sieve.SieveEntry.addTables (deviation of Ratpoints.Sieve.SieveEntry.addTables: the tables of rows made in the store of the run, by the reference to the entry (Ratpoints.Sieve.EntryRef), for the arrays sieve of the entries, after the choice for every entry, the reserve included (Ratpoints.choiceEnd), where make_power and make_modulus make them with the entry; a table of rows made only when its reference is absent, for the test of power_se (lines 2444 and 2466) by which the entry of a prime power is made once, and for the entry of a prime, which examine_prime makes once, with its table of rows, in the loop of the tables (Ratpoints.examineInLoop) or in the third stage (lines 2994-2995), where the model makes the table of a prime of the third stage after the choice, and none for a prime that the third stage examines and passes over, whose table nothing reads.)

- find_points.c:2472-2475: modelled by Ratpoints.makeModulus (deviation of Ratpoints.makeModulus: the entry returned as a fresh value, for the space that make_power and make_modulus take from the buffer of the sieve entries, and the entry of a prime power made for each modulus that needs it, where make_modulus makes it once and keeps it by its index (power_se), with the density of the first modulus that makes it, the density of the prime power in both cases; the factors as a list (Ratpoints.Sieve.Factor), the tables of a prime from the loop of the tables and the records of the prime powers, for the pointers factor to their entries; the tables of rows, which make_power (line 2427) and make_modulus (line 2471) clear, made in the store of the run after the choice (Ratpoints.Sieve.SieveEntry.addTables).)

- find_points.c:2490-2508: modelled by Ratpoints.takeEntries (deviation of Ratpoints.takeEntries: the pool, the mask and the product passed and returned, for the pointers prec, pnp, used and rate; an integer for the long want and natural numbers for the other longs; the candidate dropped by Array.eraseIdxIfInBounds, for the loop that moves the rest down.)

- find_points.c:2509-2518: modelled by Ratpoints.dropUsed

- find_points.c:2519-2520: modelled by Ratpoints.takeEntries (deviation of Ratpoints.takeEntries: the pool, the mask and the product passed and returned, for the pointers prec, pnp, used and rate; an integer for the long want and natural numbers for the other longs; the candidate dropped by Array.eraseIdxIfInBounds, for the loop that moves the rest down.)

- find_points.c:2522-2532: modelled by Ratpoints.sievingInfo (deviation of Ratpoints.sievingInfo: the result as the prime that sieving_info returns when the curve has no points modulo it, the error of the result, and as a structure (Ratpoints.SievingInfo) otherwise, for the code 0 that it returns and the fields of the arguments that it sets; the arrays prime_se, power_se and pinf of the loop of the tables in its state (Ratpoints.TableLoop), the entries of the prime powers made where a modulus needs them (Ratpoints.makeModulus); no failure for lack of memory (lines 2801 and 3061), which the model cannot meet; no output under RP_PRIME_STATS or DEBUG, and the report under RATPOINTS_VERBOSE at its end (lines 3089-3095) printed by the search after it returns (Ratpoints.Report.choice).)

- find_points.c:2533-2536: modelled by Ratpoints.Sieve.forbidden (deviation of Ratpoints.Sieve.forbidden: the position examined as an argument, for the final pn_lim of the loop of the tables; the Kronecker symbol by the table of the squares modulo p, which marks 0 as a square, for mpz_kronecker_si; the flag RATPOINTS_CHECK_DENOM not tested, since the search computes the forbidden divisors only when the flag is set, for even degree (Ratpoints.sievingInfo);

lists, for the two arrays of the arguments with their terminating zeros; the words of one of the first `Ratpoints.numPrimes` primes an array of the first `p` words of the pattern `sieves0`, for the pointers `start` and `end` into it, and those of a prime beyond them made with its entry, for the words that `sieving_info` makes after this loop in the buffer `forb_words`, with no failure for lack of memory, where `sieving_info` drops those whose words it cannot store; natural numbers for the `long`s and the reciprocals; no output under `DEBUG`.)

- `find_points.c:2537-2539`: modelled by `Ratpoints.TableLoop` (deviation of `Ratpoints.TableLoop`: a structure for the local variables of `sieving_info` that the loop changes: arrays, for `prec` with its count `pnp`, `prime_se`, `pinf` and the arrays of the forbidden divisors with their counts; the tables and the densities of the primes, for the entries `prime_se`; the store of the tables of rows, for the arrays `sieve` of the entries.)

- `find_points.c:2545-2545`: modelled by `Ratpoints.TableLoop` (deviation of `Ratpoints.TableLoop`: a structure for the local variables of `sieving_info` that the loop changes: arrays, for `prec` with its count `pnp`, `prime_se`, `pinf` and the arrays of the forbidden divisors with their counts; the tables and the densities of the primes, for the entries `prime_se`; the store of the tables of rows, for the arrays `sieve` of the entries.)

- `find_points.c:2546-2546`: modelled by `Ratpoints.sievingInfo` (deviation of `Ratpoints.sievingInfo`: the result as the prime that `sieving_info` returns when the curve has no points modulo it, the error of the result, and as a structure (`Ratpoints.SievingInfo`) otherwise, for the code `0` that it returns and the fields of the arguments that it sets; the arrays `prime_se`, `power_se` and `pinf` of the loop of the tables in its state (`Ratpoints.TableLoop`), the entries of the prime powers made where a modulus needs them (`Ratpoints.makeModulus`); no failure for lack of memory (lines 2801 and 3061), which the model cannot meet; no output under `RP_PRIME_STATS` or `DEBUG`, and the report under `RATPOINTS_VERBOSE` at its end (lines 3089-3095) printed by the search after it returns (`Ratpoints.Report.choice`).)

- `find_points.c:2547-2547`: modelled by `Ratpoints.TableLoop` (deviation of `Ratpoints.TableLoop`: a structure for the local variables of `sieving_info` that the loop changes: arrays, for `prec` with its count `pnp`, `prime_se`, `pinf` and the arrays of the forbidden divisors with their counts; the tables and the densities of the primes, for the entries `prime_se`; the store of the tables of rows, for the arrays `sieve` of the entries.)

- `find_points.c:2548-2559`: modelled by `Ratpoints.sievingInfo` (deviation of `Ratpoints.sievingInfo`: the result as the prime that `sieving_info` returns when the curve has no points modulo it, the error of the result, and as a structure (`Ratpoints.SievingInfo`) otherwise, for the code `0` that it returns and the fields of the arguments that it sets; the arrays `prime_se`, `power_se` and `pinf` of the loop of the tables in its state (`Ratpoints.TableLoop`), the entries of the prime powers made where a modulus needs them (`Ratpoints.makeModulus`); no failure for lack of memory (lines 2801 and 3061), which the model cannot meet; no output under `RP_PRIME_STATS` or `DEBUG`, and the report under `RATPOINTS_VERBOSE` at its end (lines 3089-3095) printed by the search after it returns (`Ratpoints.Report.choice`).)

- `find_points.c:2561-2562`: modelled by `Ratpoints.Sieve.forbidden` (deviation of `Ratpoints.Sieve.forbidden`: the position `examined` as an argument, for the final `pn_lim` of the loop of the tables; the Kronecker symbol by the table of the squares modulo `p`, which marks `0` as a square, for `mpz_kronecker_si`; the flag `RATPOINTS_CHECK_DENOM` not tested, since the search computes the forbidden divisors only when the flag is set, for even degree (`Ratpoints.sievingInfo`); lists, for the two arrays of the arguments with their terminating zeros; the words of one of

the first `Ratpoints.numPrimes` primes an array of the first `p` words of the pattern `sieves0`, for the pointers `start` and `end` into it, and those of a prime beyond them made with its entry, for the words that `sieving_info` makes after this loop in the buffer `forb_words`, with no failure for lack of memory, where `sieving_info` drops those whose words it cannot store; natural numbers for the `long`s and the reciprocals; no output under `DEBUG`.)

- `find_points.c:2566-2567`: modelled by `Ratpoints.TableLoop` (deviation of `Ratpoints.TableLoop`: a structure for the local variables of `sieving_info` that the loop changes: arrays, for `prec` with its count `pnp`, `prime_se`, `pinf` and the arrays of the forbidden divisors with their counts; the tables and the densities of the primes, for the entries `prime_se`; the store of the tables of rows, for the arrays `sieve` of the entries.)

- `find_points.c:2569-2575`: modelled by `Ratpoints.SieveOptions.tuning` (deviation of `Ratpoints.SieveOptions.tuning`: the four values as a quadruple, for the local variables of `sieving_info`; the doubles of the options by their bits (`Ratpoints.Double.toFloat`).)

- `find_points.c:2580-2581`: modelled by `Ratpoints.adaptStart` (deviation of `Ratpoints.adaptStart`: a `UInt64` for the `unsigned long`, `2 ^ 64 - 1` for `ULONG_MAX`.)

- `find_points.c:2587-2588`: modelled by `Ratpoints.checkRelOf` (deviation of `Ratpoints.checkRelOf`: the value returned, for the field `check_rel` of the arguments; the double of `-W` by its bits.)

- `find_points.c:2595-2596`: modelled by `Ratpoints.sievingInfo` (deviation of `Ratpoints.sievingInfo`: the result as the prime that `sieving_info` returns when the curve has no points modulo it, the error of the result, and as a structure (`Ratpoints.SievingInfo`) otherwise, for the code `0` that it returns and the fields of the arguments that it sets; the arrays `prime_se`, `power_se` and `pinf` of the loop of the tables in its state (`Ratpoints.TableLoop`), the entries of the prime powers made where a modulus needs them (`Ratpoints.makeModulus`); no failure for lack of memory (lines 2801 and 3061), which the model cannot meet; no output under `RP_PRIME_STATS` or `DEBUG`, and the report under `RATPOINTS_VERBOSE` at its end (lines 3089-3095) printed by the search after it returns (`Ratpoints.Report.choice`).)

- `find_points.c:2597-2599`: modelled by `Ratpoints.choiceEstimate` (deviation of `Ratpoints.choiceEstimate`: one function for the two places of the C code, the values returned.)

- `find_points.c:2601-2605`: modelled by `Ratpoints.tablesLoop` (deviation of `Ratpoints.tablesLoop`: a loop over the first `Ratpoints.numPrimes` positions that ends when the position reaches the number to look at, for the loop `for` whose bound the body raises, which never passes `Ratpoints.numPrimes`; no clearing of the arrays `prime_se`, `power_se` and `pinf` (lines 2601-2602), which the state starts empty.)

- `find_points.c:2606-2626`: modelled by `Ratpoints.examineInLoop` (deviation of `Ratpoints.examineInLoop`: the exit as `none`, for the prime that `sieving_info` returns, which the loop of the tables returns (`Ratpoints.tablesLoop`); the store with the table of rows of the prime, for the array `sieve` of its entry.)

- `find_points.c:2627-2631`: modelled by `Ratpoints.Sieve.forbiddenOfPrime` (deviation of `Ratpoints.Sieve.forbiddenOfPrime`: the entry returned, for the entries that the C code writes into the arrays `args->forb_ba` and `args->forbidden` and counts; the words of the prime an array of the first `p` words of the pattern `sieves0` of the build, for the pointers `start` and `end` into it; natural numbers for the `long`s and the reciprocal; no output under `DEBUG`.)

- find_points.c:2632-2641: modelled by Ratpoints.Sieve.allValuations (deviation of Ratpoints.Sieve.allValuations: a natural number for the unsigned long mask, without the cap m < LONG_LENGTH - 1 of the loop, which never binds for a long b_high and an odd prime, the loop ending before m = 40 (Ratpoints.Sieve.lt_forty_of_pow_le), and for p ≤ 1, which the C code never examines, the loop, once entered, does not end; natural numbers for the longs.)

- find_points.c:2642-2676: modelled by Ratpoints.Sieve.forbiddenOfPrime (deviation of Ratpoints.Sieve.forbiddenOfPrime: the entry returned, for the entries that the C code writes into the arrays args->forb_ba and args->forbidden and counts; the words of the prime an array of the first p words of the pattern sieves0 of the build, for the pointers start and end into it; natural numbers for the longs and the reciprocal; no output under DEBUG.)

- find_points.c:2677-2677: modelled by Ratpoints.examineInLoop (deviation of Ratpoints.examineInLoop: the exit as none, for the prime that sieving_info returns, which the loop of the tables returns (Ratpoints.tablesLoop); the store with the table of rows of the prime, for the array sieve of its entry.)

- find_points.c:2693-2693: modelled by Ratpoints.tablesLoop (deviation of Ratpoints.tablesLoop: a loop over the first Ratpoints.numPrimes positions that ends when the position reaches the number to look at, for the loop for whose bound the body raises, which never passes Ratpoints.numPrimes; no clearing of the arrays prime_se, power_se and pinf (lines 2601-2602), which the state starts empty.)

- find_points.c:2694-2705: modelled by Ratpoints.raiseLimit (deviation of Ratpoints.raiseLimit: the sort by qsort with compare_entries as Lean's stable merge sort (Ratpoints.sortFrom), which gives the order of glibc's qsort, stable when it can allocate its buffer, as always in the model, also on ties, since the keys are finite or +∞; integers for the longs that may be negative or large.)

- find_points.c:2707-2707: modelled by Ratpoints.tablesLoop (deviation of Ratpoints.tablesLoop: a loop over the first Ratpoints.numPrimes positions that ends when the position reaches the number to look at, for the loop for whose bound the body raises, which never passes Ratpoints.numPrimes; no clearing of the arrays prime_se, power_se and pinf (lines 2601-2602), which the state starts empty.)

- find_points.c:2709-2772: modelled by Ratpoints.Sieve.forbidden (deviation of Ratpoints.Sieve.forbidden: the position examined as an argument, for the final pn_lim of the loop of the tables; the Kronecker symbol by the table of the squares modulo p, which marks 0 as a square, for mpz_kronecker_si; the flag RATPOINTS_CHECK_DENOM not tested, since the search computes the forbidden divisors only when the flag is set, for even degree (Ratpoints.sievingInfo); lists, for the two arrays of the arguments with their terminating zeros; the words of one of the first Ratpoints.numPrimes primes an array of the first p words of the pattern sieves0, for the pointers start and end into it, and those of a prime beyond them made with its entry, for the words that sieving_info makes after this loop in the buffer forb_words, with no failure for lack of memory, where sieving_info drops those whose words it cannot store; natural numbers for the longs and the reciprocals; no output under DEBUG.)

- find_points.c:2773-2777: modelled by Ratpoints.Sieve.forbiddenRow (deviation of Ratpoints.Sieve.forbiddenRow: one function for the two codes that make these words, the array work of the generator of find_points.h, for the primes of the table, and the words

that `sieving_info` makes in the buffer `forb_words` for the forbidden divisors beyond the table (`Ratpoints.Sieve.forbidden`), with the positions `p i` for `i < 64`, for the positions `m = 0, p, 2 p, …` below `64 p` there; a fresh array of `p` words, for the array `work` of `RATPOINTS_MAX_PRIME` words and for the space in the buffer; divisions and remainders for the shifts and masks by `LONG_SHIFT` and `LONG_MASK`.)

- `find_points.c:2778-2789`: modelled by `Ratpoints.Sieve.forbidden` (deviation of `Ratpoints.Sieve.forbidden`: the position `examined` as an argument, for the final `pn_lim` of the loop of the tables; the Kronecker symbol by the table of the squares modulo `p`, which marks `0` as a square, for `mpz_kronecker_si`; the flag `RATPOINTS_CHECK_DENOM` not tested, since the search computes the forbidden divisors only when the flag is set, for even degree (`Ratpoints.sievingInfo`); lists, for the two arrays of the arguments with their terminating zeros; the words of one of the first `Ratpoints.numPrimes` primes an array of the first `p` words of the pattern `sieves0`, for the pointers `start` and `end` into it, and those of a prime beyond them made with its entry, for the words that `sieving_info` makes after this loop in the buffer `forb_words`, with no failure for lack of memory, where `sieving_info` drops those whose words it cannot store; natural numbers for the `long`s and the reciprocals; no output under `DEBUG`.)

- `find_points.c:2794-2795`: modelled by `Ratpoints.sievingInfo` (deviation of `Ratpoints.sievingInfo`: the result as the prime that `sieving_info` returns when the curve has no points modulo it, the error of the result, and as a structure (`Ratpoints.SievingInfo`) otherwise, for the code `0` that it returns and the fields of the arguments that it sets; the arrays `prime_se`, `power_se` and `pinf` of the loop of the tables in its state (`Ratpoints.TableLoop`), the entries of the prime powers made where a modulus needs them (`Ratpoints.makeModulus`); no failure for lack of memory (lines 2801 and 3061), which the model cannot meet; no output under `RP_PRIME_STATS` or `DEBUG`, and the report under `RATPOINTS_VERBOSE` at its end (lines 3089-3095) printed by the search after it returns (`Ratpoints.Report.choice`).)

- `find_points.c:2801-2801`: excluded (memory): the enlargement of ba_buffer for the primes that the loop of the tables of sieving_info looked at beyond num_primes, which fails only for lack of memory

- `find_points.c:2808-2812`: modelled by `Ratpoints.choiceTail` (deviation of `Ratpoints.choiceTail`: the sort by `qsort` with `compare_entries` as Lean's stable merge sort (`Ratpoints.sortFrom`), which gives the order of glibc's `qsort`, stable when it can allocate its buffer, as always in the model, also on ties, since the keys are finite or +∞; the result returned as a structure (`Ratpoints.SievingInfo`), for the fields of the arguments that `sieving_info` sets.)

- `find_points.c:2813-2815`: modelled by `Ratpoints.choiceEstimate` (deviation of `Ratpoints.choiceEstimate`: one function for the two places of the C code, the values returned.)

- `find_points.c:2816-2823`: modelled by `Ratpoints.choiceTail` (deviation of `Ratpoints.choiceTail`: the sort by `qsort` with `compare_entries` as Lean's stable merge sort (`Ratpoints.sortFrom`), which gives the order of glibc's `qsort`, stable when it can allocate its buffer, as always in the model, also on ties, since the keys are finite or +∞; the result returned as a structure (`Ratpoints.SievingInfo`), for the fields of the arguments that `sieving_info` sets.)

- `find_points.c:2826-2826`: modelled by `Ratpoints.choiceTail` (deviation of `Ratpoints.choiceTail`: the sort by `qsort` with `compare_entries` as Lean's stable merge sort (`Ratpoints.sortFrom`), which gives the order of glibc's `qsort`, stable when it can allocate its buffer, as always in the model, also on ties, since the keys are finite or +∞; the result returned as a structure (`Ratpoints.SievingInfo`), for the fields of the arguments that `sieving_info` sets.)

- find_points.c:2843-2846: modelled by `Ratpoints.firstStages` (deviation of `Ratpoints.firstStages`: integers for the `long`s that may be negative.)
- find_points.c:2853-2863: modelled by `Ratpoints.rankSecond` (deviation of `Ratpoints.rankSecond`: the sort by `qsort` with `compare_entries` as Lean's stable merge sort (`Ratpoints.sortFrom`), which gives the order of glibc's `qsort`, stable when it can allocate its buffer, as always in the model, also on ties, since the keys are finite or +∞.)
- find_points.c:2864-2871: modelled by `Ratpoints.firstStages` (deviation of `Ratpoints.firstStages`: integers for the `long`s that may be negative.)
- find_points.c:2879-2890: modelled by `Ratpoints.dropComposites`
- find_points.c:2894-2900: modelled by `Ratpoints.stageEntries` (deviation of `Ratpoints.stageEntries`: the entries kept in the candidates, for the array `sieve_list` into which `sieving_info` puts them (line 2898), which the caller reads from the candidates.)
- find_points.c:2922-2922: modelled by `Ratpoints.ThirdState` (deviation of `Ratpoints.ThirdState`: a structure for the local variables of the stage.)
- find_points.c:2926-2932: modelled by `Ratpoints.thirdOf` (deviation of `Ratpoints.thirdOf`: integers for the comparison of the `long`s; the double of `-Q` by its bits.)
- find_points.c:2933-2933: modelled by `Ratpoints.ThirdState` (deviation of `Ratpoints.ThirdState`: a structure for the local variables of the stage.)
- find_points.c:2941-2947: modelled by `Ratpoints.survivorsAtThird` (deviation of `Ratpoints.survivorsAtThird`: `64` for `LONG_LENGTH`.)
- find_points.c:2949-2950: modelled by `Ratpoints.thirdStage` (deviation of `Ratpoints.thirdStage`: a `while` loop over the state, the candidates swapped by `Array.swapIfInBounds`; the sieve entry of the prime taken kept in its candidate, for the array `sieve_list` (line 3014).)
- find_points.c:2955-2961: modelled by `Ratpoints.bestByR`
- find_points.c:2973-2975: modelled by `Ratpoints.thirdStage` (deviation of `Ratpoints.thirdStage`: a `while` loop over the state, the candidates swapped by `Array.swapIfInBounds`; the sieve entry of the prime taken kept in its candidate, for the array `sieve_list` (line 3014).)
- find_points.c:2976-2978: modelled by `Ratpoints.lookFurther`
- find_points.c:2980-2981: modelled by `Ratpoints.thirdStage` (deviation of `Ratpoints.thirdStage`: a `while` loop over the state, the candidates swapped by `Array.swapIfInBounds`; the sieve entry of the prime taken kept in its candidate, for the array `sieve_list` (line 3014).)
- find_points.c:2982-2991: modelled by `Ratpoints.rTyp`
- find_points.c:2992-2993: modelled by `Ratpoints.thirdStage` (deviation of `Ratpoints.thirdStage`: a `while` loop over the state, the candidates swapped by `Array.swapIfInBounds`; the sieve entry of the prime taken kept in its candidate, for the array `sieve_list` (line 3014).)
- find_points.c:2994-3008: modelled by `Ratpoints.lookFurther`
- find_points.c:3009-3016: modelled by `Ratpoints.thirdStage` (deviation of `Ratpoints.thirdStage`: a `while` loop over the state, the candidates swapped by `Array.swapIfInBounds`; the sieve entry of the prime taken kept in its candidate, for the array `sieve_list` (line 3014).)

- find_points.c:3017-3017: modelled by Ratpoints.choiceEnd (deviation of Ratpoints.choiceEnd: the sort by qsort with compare_by_r as Lean's stable merge sort (Ratpoints.sortFrom), from the end of the second stage on, as the C code sorts, which gives the order of glibc's qsort, stable when it can allocate its buffer, as always in the model; the entries, the reciprocals and the store returned, for the arrays sieve_list and magics of the arguments and the arrays sieve of the entries; sp3_max the length of the list of the entries, the larger of sp3 and the size of the pool, which is never smaller; no enlargement of the buffer of the rows (lines 3042-3061), which the model does not have.)
- find_points.c:3024-3034: modelled by Ratpoints.choiceEnd (deviation of Ratpoints.choiceEnd: the sort by qsort with compare_by_r as Lean's stable merge sort (Ratpoints.sortFrom), from the end of the second stage on, as the C code sorts, which gives the order of glibc's qsort, stable when it can allocate its buffer, as always in the model; the entries, the reciprocals and the store returned, for the arrays sieve_list and magics of the arguments and the arrays sieve of the entries; sp3_max the length of the list of the entries, the larger of sp3 and the size of the pool, which is never smaller; no enlargement of the buffer of the rows (lines 3042-3061), which the model does not have.)
- find_points.c:3036-3062: excluded (memory): the enlargement of ba_buffer for the tables of every prime looked at, those that the third stage looked at among them, and of the composite moduli of the first two stages and their prime-power factors, which fails only for lack of memory
- find_points.c:3064-3070: modelled by Ratpoints.choiceEnd (deviation of Ratpoints.choiceEnd: the sort by qsort with compare_by_r as Lean's stable merge sort (Ratpoints.sortFrom), from the end of the second stage on, as the C code sorts, which gives the order of glibc's qsort, stable when it can allocate its buffer, as always in the model; the entries, the reciprocals and the store returned, for the arrays sieve_list and magics of the arguments and the arrays sieve of the entries; sp3_max the length of the list of the entries, the larger of sp3 and the size of the pool, which is never smaller; no enlargement of the buffer of the rows (lines 3042-3061), which the model does not have.)
- find_points.c:3073-3087: excluded (instrumentation): RP_PRIME_STATS, development instrumentation that prints on stderr one line per curve: the primes looked at and those that carry information, the ends of the three stages, the estimates of the run and the density of each candidate
- find_points.c:3089-3095: modelled by Ratpoints.Report.choice (deviation of Ratpoints.Report.choice: the result of the choice as an argument, for the local pn_lim and the fields of the arguments that sieving_info prints, which the search prints after it returns, nothing being printed in between; the double bpw a value of Lean's Float (Ratpoints.Double.ofFloat); the length of the third stage an integer, for the difference of the longs.)
- find_points.c:3098-3100: excluded (instrumentation): the message under DEBUG at the end of sieving_info
- find_points.c:3102-3103: modelled by Ratpoints.sievingInfo (deviation of Ratpoints.sievingInfo: the result as the prime that sieving_info returns when the curve has no points modulo it, the error of the result, and as a structure (Ratpoints.SievingInfo) otherwise, for the code 0 that it returns and the fields of the arguments that it sets; the arrays prime_se, power_se and pinf of the loop of the tables in its state (Ratpoints.TableLoop), the entries

of the prime powers made where a modulus needs them (`Ratpoints.makeModulus`); no failure for lack of memory (lines 2801 and 3061), which the model cannot meet; no output under `RP_PRIME_STATS` or `DEBUG`, and the report under `RATPOINTS_VERBOSE` at its end (lines 3089-3095) printed by the search after it returns (`Ratpoints.Report.choice`).)

- `find_points.c:3112-3114`: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `long`s; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `find_points.c:3119-3121`: modelled by `Ratpoints.Sieve.siftPiece` (deviation of `Ratpoints.Sieve.siftPiece`: the arm of the first stage chosen by a test on the constant `Ratpoints.chunk`, for the conditional compilation `#if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16)` (`sift.c:452`, `sift.c:575`, `sift.c:943`), which compiles one arm only, and `tail_leg` with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words `malloc` leaves uninitialized; the piece by its first word number `w0` and its number `n` of bit arrays, for `w_low` and `w_high`, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list `specs` and the end `sp1` of the first stage, for the sieve specs `sieves` and the number `sp1` that the sieving thread keeps, the reciprocals of their moduli as the list `mgs`, for the array `magics` of the arguments, and the thunk `checks`, for the table `checks` of the third stage and its length `nchecks = sp3 - sp2`; the class `cl` as a value, for the pointer `cls`; integers for the `long`s of the test `small`, which

stay longs, and a Boolean for its int; the counters cnt passed and returned, for the field n_words of the sieving thread, which counts the n Ratpoints.rbaPack words of the piece (line 548), the product of UInt64s as the C code multiplies the unsigned long by RBA_PACK; the thunk w of the products of the check as an argument, where _ratpoints_check_point computes the products when the first survivor of b reaches it; the delivery r passed and returned, for the pointers info and wk->quit and the count total that _ratpoints_sift0 keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (Ratpoints.Sieve.scanPiece); natural numbers and integers for the longs; no measurements of the phases (RP_TIC, RP_TOC, RP_PHASE_TIMING) and no stop after one of them (RP_STOP_AFTER).)

- find_points.c:3122-3122: modelled by Ratpoints.Sieve.checkPoint (deviation of Ratpoints.Sieve.checkPoint: the products w as a thunk, which the sieve of one denominator makes for each denominator (Ratpoints.Sieve.sift) and which is forced when the first numerator of b reaches the check, for the array work of the sieving thread, which worker_init gives it, and its flag compute_bc, which sift sets and _ratpoints_check_point clears when it computes the products; the degree d and the flags fl as arguments, handed down from Ratpoints.Sieve.sieveRun through the components that call the check, for the degree and the bits RATPOINTS_REVERSED, RATPOINTS_NO_CHECK and RATPOINTS_NO_Y of the flags of the search, which _ratpoints_check_point reads through its worker (args->degree, args->flags); the delivery r passed and returned, for the pointers info and wk->quit and the count total that _ratpoints_check_point keeps from zero and returns, and the three branches of the map of the reversal at each call one definition (Ratpoints.Delivery.callMapped); the square root and the test of the remainder by Nat.sqrt on v, for mpz_sqrtrem; integers for the longs a and b (the negations of the map stay longs by Ratpoints.fitsLong_neg); no measurements of the time of the check and of its products (RP_CHECK, RP_BC_TIC, RP_BC_TOC under RP_PHASE_TIMING), and no output under DEBUG.)

- find_points.c:3123-3124: modelled by Ratpoints.Sieve.sift (deviation of Ratpoints.Sieve.sift: the height bound converted to Int64, which is exact for H < 2 ^ 63, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct rp_intervals with its pointers bd and end, which sift reaches through the field iv of the sieving thread, which worker_init clears (line 3130) and run_block sets for each block (line 3913; here Ratpoints.Sieve.runBlock, which passes the array of the run on); one call of Ratpoints.Sieve.setRows, for the two calls with the flag shared a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays sieve of the entries of the arguments, which sift reaches through sieve_list and which keep the rows built for the rest of the run; the end sp1 of the first stage, the entries es and the reciprocals mgs of the first two stages, the entries es3 of the third stage and their reciprocals mgs3 as arguments, for the numbers sp1, sp2 and sp3 that the sieving thread keeps (worker_init), the entries sieve_list[0], …, sieve_list[sp2 - 1], the array magics of the arguments, which _ratpoints_sift0 reads, and the entries sieve_list[sp2], …, sieve_list[sp3 - 1] and the reciprocals magics[sp2], …, magics[sp3 - 1], which fill_checks reads, the cut points of the decision of the block (Ratpoints.Sieve.runBlock); the residues bp and the class cl as values, for the pointers bp_list and cls, and the array offsets of the row offsets of the run as an argument, which the C code reaches through the pointer offset of the class; the thunk of Ratpoints.Sieve.fillChecks, for the table checks of the sieving thread, which fill_checks fills, and its flag stage3_filled, which sift clears; the thunk of the products w, for the flag compute_bc, which sift sets and _ratpoints_check_point clears when it com-

putes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `longs`; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `find_points.c:3127-3128`: modelled by `Ratpoints.Sieve.runBlock` (deviation of `Ratpoints.Sieve.runBlock`: the decision's entries and reciprocals cut as lists for each block, for the numbers `sp2` and `sp3` that the sieving thread keeps and through which `sift` and `fill_bp_list` read the arrays `sieve_list` and `magics`; the report returned with the delivery and the store, for the fields of the sieving thread that `worker_stat` copies into it after the block; the counters and the index of the decision, which `worker_init` clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that `run_block` reads; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `run_block` keeps from zero and returns; the array `bp_list` of `sp3_max` entries on the stack, of which `fill_bp_list` fills the first `sp2`, a list made by `Ratpoints.Sieve.siftDen`; natural numbers for the `longs`.)

- `find_points.c:3129-3130`: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift`

reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `long`s; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `find_points.c:3132-3136`: excluded (memory): the result of worker_init, whether the survivors array could be allocated, and worker_clear, which frees it; the model takes a fresh array for each piece and cannot fail for lack of memory
- `find_points.c:3149-3170`: modelled by `Ratpoints.Sieve.setRows` (deviation of `Ratpoints.Sieve.setRows`: a list of the moduli with their rows and offsets, for the array of sieve specs `ssp` that `set_rows` fills; the empty array for the null pointer of a row not built yet; the start set to `0`, where the C code leaves the field as it is until the first stage sets it; the end `p` also for a row not built yet, where the C code adds `p` to the null pointer; the row read plainly, for the acquire load that the flag `shared` chooses when threads share the tables of rows; the store of the tables of rows of the run as an argument, for the arrays `sieve` of the entries that the pointers in `sieve_list` reach; the array of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class, and a class without a row of offsets (none, for `NULL`), which the run never sifts, reads the offsets from position `0`; natural numbers for the `long`s.)
- `find_points.c:3217-3218`: modelled by `Ratpoints.Sieve.ExactEnd` (deviation of `Ratpoints.Sieve.ExactEnd`: a natural number for the `int s`, which is from `0` to `127`, and a Boolean for the `int neg`.)
- `find_points.c:3219-3221`: modelled by `Ratpoints.Sieve.IntervalBounds` (deviation of `Ratpoints.Sieve.IntervalBounds`: natural numbers for the `int` shifts `lsh` and `ush`, which are from `0` to `63`, and a Boolean for the `int fast`.)
- `find_points.c:3231-3238`: modelled by `Ratpoints.Sieve.exactScale` (deviation of `Ratpoints.Sieve.exactScale`: the 128-bit product, sum and shift on two 64-bit halves, the high half of the product by `Ratpoints.Sieve.mulHi`, for the `unsigned __int128` of the arm that the modelled builds compile (the compiler defines `__SIZEOF_INT128__`, and `RP_NO_INT128` is not set), which Lean core does not have; the other arm (lines 3239-3256), for a build without a 128-bit type, is excluded; the `long b` converted to a 64-bit word, as the C code converts it to an `unsigned long`.)
- `find_points.c:3239-3256`: excluded (other): the arm of exact_scale for a build without a 128-bit type or with RP_NO_INT128, which puts the 128-bit product together from 32-bit halves; the modelled builds compile the arm with unsigned ___int128 (Ratpoints.Sieve.exactScale)
- `find_points.c:3257-3257`: modelled by `Ratpoints.Sieve.exactScale` (deviation of `Ratpoints.Sieve.exactScale`: the 128-bit product, sum and shift on two 64-bit halves, the high half of the product by `Ratpoints.Sieve.mulHi`, for the `unsigned __int128` of the arm that the modelled builds compile (the compiler defines `__SIZEOF_INT128__`, and `RP_NO_INT128` is not set), which Lean core does not have; the other arm (lines 3239-3256), for a build without a 128-bit type, is excluded; the `long b` converted to a 64-bit word, as the C code converts it to an `unsigned long`.)
- `find_points.c:3264-3271`: modelled by `Ratpoints.Sieve.exactEnd` (deviation of `Ratpoints.Sieve.exactEnd`: the double as its bits, with `fabs` and the comparisons of the C language on them (`Ratpoints.Double.fabs`, `Ratpoints.Double.lt`); a Boolean for the `int lower`; the end returned, for the `rp_end` that the C code writes through its pointer.)

- find_points.c:3272-3291: modelled by `Ratpoints.Sieve.exactMantissa` (deviation of `Ratpoints.Sieve.exactMantissa`: an integer for the `int s` while it is computed, and a natural number when it is returned; the lowest set bit by the loop of `RP_CTZL` (`Ratpoints.Sieve.ctz`), where gcc and clang take `__builtin_ctzl`; the mantissa and the exponent returned, for the fields `m` and `s` of the `rp_end` that the C code writes.)

- find_points.c:3292-3299: modelled by `Ratpoints.Sieve.exactEnd` (deviation of `Ratpoints.Sieve.exactEnd`: the double as its bits, with `fabs` and the comparisons of the C language on them (`Ratpoints.Double.fabs`, `Ratpoints.Double.lt`); a Boolean for the `int lower`; the end returned, for the `rp_end` that the C code writes through its pointer.)

- find_points.c:3304-3318: modelled by `Ratpoints.Sieve.endForm` (deviation of `Ratpoints.Sieve.endForm`: the form returned as an option, for the pointers `M`, `C` and `sh` through which the C code writes it and the `int` that says whether it has one; `Int64` for the `long`s, with the division of `Int64`, which truncates as the C division does, on the non-negative `long`s `LONG_MAX - c` and `b_high`; a Boolean for the `int lower`; a natural number for the `int sh`.)

- find_points.c:3323-3340: modelled by `Ratpoints.Sieve.exactBounds` (deviation of `Ratpoints.Sieve.exactBounds`: the search intervals and the bounds as arguments, for the fields `domain`, `num_inter`, `height` and `b_high` of the arguments, and the array of the bounds returned, for the array that `run_setup` allocates and `exact_bounds` fills through its pointer; `Int64` for the `long`s; the conjunction of the tests, whose `&&` makes the form of the upper end only when the lower end has one, as a bind of options, and the forms as the options that `Ratpoints.Sieve.endForm` returns, for the fields that `end_form` writes.)

- find_points.c:3355-3370: modelled by `Ratpoints.Sieve.numeratorsToBits` (deviation of `Ratpoints.Sieve.numeratorsToBits`: `Int64` for the `long`s, in two's complement, the representation on which the and with the mask acts, also on a negative end, and `>>>` on `Int64` the arithmetic shift, for the `>>` of a negative `long`, both of which the C standard leaves to the implementation (the representation before C23) and gcc defines so; the arithmetic of `Int64` wrapping where that of the `long`s overflows, which the C standard leaves undefined: the differences of the plain form for an end within a_0 of `-2 ^ 63`, so beyond `2 ^ 63 - 65` in absolute value, where neither the C code nor `Ratpoints.Sieve.sift` takes the plain form; the stride `k` and the residue a_0 of the class converted to `Int64`; the two ends returned as a pair, for the pointers `low` and `high` through which the function reads and writes them; a Boolean for the `int split`, an argument: clear on the path of the forms of the ends (`Ratpoints.Sieve.intervalBits`), which `exact_bounds` gives an interval only for $H \le$ `2 ^ 63 - 65`, and on the path of the products in 128 bits (`Ratpoints.Sieve.intervalBitsExact`) set exactly for the heights above `2 ^ 63 - 65`.)

- find_points.c:3381-3406: modelled by `Ratpoints.Sieve.intervalBitsExact` (deviation of `Ratpoints.Sieve.intervalBitsExact`: the code and the bits returned as a triple, for the `int` and the pointers `low` and `high` through which the C code writes the bits, the bits `(0, 0)` with the codes `1` and `2`, for which the C code writes nothing; `Int64` for the `long`s, the `unsigned long` products compared with the height bound converted to a 64-bit word, as the C code compares them; no output under `DEBUG`.)

- find_points.c:3410-3434: modelled by `Ratpoints.Sieve.intervalBits` (deviation of `Ratpoints.Sieve.intervalBits`: the code and the bits returned as a triple, for the `int` and the pointers `low` and `high` through which the C code writes the bits, the bits `(0, 0)` with the codes `1` and `2`, for which the C code writes nothing; `Int64` for the `long`s, and `>>>` on

Int64 the arithmetic shift, for the >> of a negative long, which the C standard leaves to the implementation and gcc defines so; the arithmetic of Int64 wrapping where that of the longs overflows, which it does not here, by the bound of Ratpoints.Sieve.endForm on the forms of an interval taken inline and the forms of an empty one otherwise; no output under DEBUG.)

- find_points.c:3435-3440: modelled by Ratpoints.Sieve.sift (deviation of Ratpoints.Sieve.sift: the height bound converted to Int64, which is exact for H < 2 ^ 63, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct rp_intervals with its pointers bd and end, which sift reaches through the field iv of the sieving thread, which worker_init clears (line 3130) and run_block sets for each block (line 3913; here Ratpoints.Sieve.runBlock, which passes the array of the run on); one call of Ratpoints.Sieve.setRows, for the two calls with the flag shared a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays sieve of the entries of the arguments, which sift reaches through sieve_list and which keep the rows built for the rest of the run; the end sp1 of the first stage, the entries es and the reciprocals mgs of the first two stages, the entries es3 of the third stage and their reciprocals mgs3 as arguments, for the numbers sp1, sp2 and sp3 that the sieving thread keeps (worker_init), the entries sieve_list[0], …, sieve_list[sp2 - 1], the array magics of the arguments, which _ratpoints_sift0 reads, and the entries sieve_list[sp2], …, sieve_list[sp3 - 1] and the reciprocals magics[sp2], …, magics[sp3 - 1], which fill_checks reads, the cut points of the decision of the block (Ratpoints.Sieve.runBlock); the residues bp and the class cl as values, for the pointers bp_list and cls, and the array offsets of the row offsets of the run as an argument, which the C code reaches through the pointer offset of the class; the thunk of Ratpoints.Sieve.fillChecks, for the table checks of the sieving thread, which fill_checks fills, and its flag stage3_filled, which sift clears; the thunk of the products w, for the flag compute_bc, which sift sets and _ratpoints_check_point clears when it computes the products; the counters cnt passed and returned, for those of the sieving thread, the count sifts one up at the start (line 3461); the delivery r passed and returned, for the pointers info and wk->quit, which worker_init clears, and the count total that sift keeps from zero and returns; the arguments of the run passed explicitly, for the pointer args of the sieving thread (rp-private.h:574), which worker_init sets and through which sift reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (pool, slot, stop, rp-private.h:584, excluded with the threads); natural numbers and integers for the longs; no measurements of the phases (RP_SIFT_TIC, RP_SETUP_TIC and their TOC) and no output under DEBUG.)
- find_points.c:3442-3488: modelled by Ratpoints.Sieve.sift (deviation of Ratpoints.Sieve.sift: the height bound converted to Int64, which is exact for H < 2 ^ 63, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct rp_intervals with its pointers bd and end, which sift reaches through the field iv of the sieving thread, which worker_init clears (line 3130) and run_block sets for each block (line 3913; here Ratpoints.Sieve.runBlock, which passes the array of the run on); one call of Ratpoints.Sieve.setRows, for the two calls with the flag shared a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays sieve of the entries of the arguments, which sift reaches through sieve_list and which keep the rows built for the rest of the run; the end sp1 of the first stage, the entries es and the reciprocals mgs of the first two stages, the entries es3 of the third stage and their reciprocals mgs3 as arguments,

for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `long`s; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `find_points.c:3490-3521`: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with

the threads); natural numbers and integers for the longs; no measurements of the phases (RP_SIFT_TIC, RP_SETUP_TIC and their TOC) and no output under DEBUG.)

- find_points.c:3522-3554: modelled by Ratpoints.Sieve.buildRows (deviation of Ratpoints.Sieve.buildRows: the specs and the store of the tables of rows of the run passed and returned, for the array of the sieve specs and the arrays sieve of the entries; the empty array for the null pointer of a row not built yet; no lock of the tables (rp_tables_lock, rp_tables_unlock), and the pointer read again plainly, for the acquire load, both for the threads that share the tables of rows, which the model does not have; no measurement of the construction (RP_INIT_TIC, RP_INIT_TOC) and no output under DEBUG.)

- find_points.c:3555-3561: modelled by Ratpoints.Sieve.sift (deviation of Ratpoints.Sieve.sift: the height bound converted to Int64, which is exact for H < 2 ^ 63, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct rp_intervals with its pointers bd and end, which sift reaches through the field iv of the sieving thread, which worker_init clears (line 3130) and run_block sets for each block (line 3913; here Ratpoints.Sieve.runBlock, which passes the array of the run on); one call of Ratpoints.Sieve.setRows, for the two calls with the flag shared a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays sieve of the entries of the arguments, which sift reaches through sieve_list and which keep the rows built for the rest of the run; the end sp1 of the first stage, the entries es and the reciprocals mgs of the first two stages, the entries es3 of the third stage and their reciprocals mgs3 as arguments, for the numbers sp1, sp2 and sp3 that the sieving thread keeps (worker_init), the entries sieve_list[0], …, sieve_list[sp2 - 1], the array magics of the arguments, which _ratpoints_sift0 reads, and the entries sieve_list[sp2], …, sieve_list[sp3 - 1] and the reciprocals magics[sp2], …, magics[sp3 - 1], which fill_checks reads, the cut points of the decision of the block (Ratpoints.Sieve.runBlock); the residues bp and the class cl as values, for the pointers bp_list and cls, and the array offsets of the row offsets of the run as an argument, which the C code reaches through the pointer offset of the class; the thunk of Ratpoints.Sieve.fillChecks, for the table checks of the sieving thread, which fill_checks fills, and its flag stage3_filled, which sift clears; the thunk of the products w, for the flag compute_bc, which sift sets and _ratpoints_check_point clears when it computes the products; the counters cnt passed and returned, for those of the sieving thread, the count sifts one up at the start (line 3461); the delivery r passed and returned, for the pointers info and wk->quit, which worker_init clears, and the count total that sift keeps from zero and returns; the arguments of the run passed explicitly, for the pointer args of the sieving thread (rp-private.h:574), which worker_init sets and through which sift reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (pool, slot, stop, rp-private.h:584, excluded with the threads); natural numbers and integers for the longs; no measurements of the phases (RP_SIFT_TIC, RP_SETUP_TIC and their TOC) and no output under DEBUG.)

- find_points.c:3563-3612: modelled by Ratpoints.Sieve.siftBits (deviation of Ratpoints.Sieve.siftBits: a function of its own for the bits of one interval of the numerators, the part of the loop over the intervals of sift after the set-up; the size A of the pieces as an argument, for args->array_size (line 3571), which sift reads through the pointer args of the sieving thread; natural numbers and integers for the longs, the word numbers and the masks by the division and the remainder, rounding down, by Ratpoints.rbaLength, for the shift by RBA_SHIFT and the mask by RBA_LENGTH - 1 (with Ratpoints.rbaLength

= 2 ^ Ratpoints.rbaShift), which gcc makes the floor division and the Euclidean remainder on a negative long too, the first bits of the bit arrays staying longs for the bits of a height bound below 2 ^ 63 (Ratpoints.Sieve.neg_two_pow_le_rbaLength_mul_ediv, Ratpoints.Sieve.rbaLength_mul_ediv_add_one_le_two_pow); the delivery r passed and returned, for the pointers info and wk->quit and the count total that sift keeps from zero and returns, and the counters cnt passed and returned, for those of the sieving thread; no test of the flag stop of a thread of the pool after a piece (line 3610), which is null on the calling thread (line 5129, with the pool excluded with the threads), and no measurement (RP_SIFT_TOC) on the return at a stop.)

- find_points.c:3613-3617: modelled by Ratpoints.Sieve.sift (deviation of Ratpoints.Sieve.sift: the height bound converted to Int64, which is exact for H < 2 ^ 63, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct rp_intervals with its pointers bd and end, which sift reaches through the field iv of the sieving thread, which worker_init clears (line 3130) and run_block sets for each block (line 3913; here Ratpoints.Sieve.runBlock, which passes the array of the run on); one call of Ratpoints.Sieve.setRows, for the two calls with the flag shared a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays sieve of the entries of the arguments, which sift reaches through sieve_list and which keep the rows built for the rest of the run; the end sp1 of the first stage, the entries es and the reciprocals mgs of the first two stages, the entries es3 of the third stage and their reciprocals mgs3 as arguments, for the numbers sp1, sp2 and sp3 that the sieving thread keeps (worker_init), the entries sieve_list[0], …, sieve_list[sp2 - 1], the array magics of the arguments, which _ratpoints_sift0 reads, and the entries sieve_list[sp2], …, sieve_list[sp3 - 1] and the reciprocals magics[sp2], …, magics[sp3 - 1], which fill_checks reads, the cut points of the decision of the block (Ratpoints.Sieve.runBlock); the residues bp and the class cl as values, for the pointers bp_list and cls, and the array offsets of the row offsets of the run as an argument, which the C code reaches through the pointer offset of the class; the thunk of Ratpoints.Sieve.fillChecks, for the table checks of the sieving thread, which fill_checks fills, and its flag stage3_filled, which sift clears; the thunk of the products w, for the flag compute_bc, which sift sets and _ratpoints_check_point clears when it computes the products; the counters cnt passed and returned, for those of the sieving thread, the count sifts one up at the start (line 3461); the delivery r passed and returned, for the pointers info and wk->quit, which worker_init clears, and the count total that sift keeps from zero and returns; the arguments of the run passed explicitly, for the pointer args of the sieving thread (rp-private.h:574), which worker_init sets and through which sift reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (pool, slot, stop, rp-private.h:584, excluded with the threads); natural numbers and integers for the longs; no measurements of the phases (RP_SIFT_TIC, RP_SETUP_TIC and their TOC) and no output under DEBUG.)

- find_points.c:3639-3679: modelled by Ratpoints.Sieve.fillBpList (deviation of Ratpoints.Sieve.fillBpList: a list, made by a map over the entries and their reciprocals (List.zipWith) in the branch that b chooses and returned, for the array bp_list that the loops over the positions 0, …, sp2 - 1 of sieve_list and magics fill through the pointer argument; the entries of the first two stages and their reciprocals as arguments, for wk->sp2 and the arrays sieve_list and magics of the arguments; natural numbers for the longs, the products natural numbers for the longs that RP_MULMOD converts to unsigned longs, at most Ratpoints.mulModLimit in the first branch (Ratpoints.mul_le_mulModLimit_of_le_div) and below 2 ^ 63 in the others

(Ratpoints.mul_lt_two_pow_63); no measurement of its time (RP_BP_TIC, RP_BP_TOC).)

- find_points.c:3725-3728: modelled by Ratpoints.Sieve.Shape (deviation of Ratpoints.Sieve.Shape: an inductive type with the data of each shape, for the constant shape and the fields of rp_run that the shapes read, with these exceptions: the forbidden divisors as lists, for forb_ba and forbidden with no count nfb, a list having its length; the tables of the Jacobi test as an option, for ji, jtab and fast_jacobi; with the words, the bit RATPOINTS_USE_JACOBI of the flags and the leading coefficient args->cof[args->degree], which run_block reads through the field args, the run taking the rest of what it reads of the arguments as its own arguments (Ratpoints.Sieve.sieveRun); no copies of use_c_long and lcf_long, which only jacobi1 reads (Ratpoints.Sieve.jacobi); the segments and the blocks (Ratpoints.Sieve.BlockPlan) and the counters and the decisions of the correction of the moduli (Ratpoints.Sieve.Correction) structures of their own; the bounds of the search intervals (the struct iv of the run, Ratpoints.Sieve.runSetup), the classes with their row offsets and den_bits passed to the run apart (Ratpoints.Sieve.sieveRun).)
- find_points.c:3729-3731: modelled by Ratpoints.blocksWanted
- find_points.c:3732-3734: modelled by Ratpoints.blockMin
- find_points.c:3736-3749: modelled by Ratpoints.Sieve.Shape (deviation of Ratpoints.Sieve.Shape: an inductive type with the data of each shape, for the constant shape and the fields of rp_run that the shapes read, with these exceptions: the forbidden divisors as lists, for forb_ba and forbidden with no count nfb, a list having its length; the tables of the Jacobi test as an option, for ji, jtab and fast_jacobi; with the words, the bit RATPOINTS_USE_JACOBI of the flags and the leading coefficient args->cof[args->degree], which run_block reads through the field args, the run taking the rest of what it reads of the arguments as its own arguments (Ratpoints.Sieve.sieveRun); no copies of use_c_long and lcf_long, which only jacobi1 reads (Ratpoints.Sieve.jacobi); the segments and the blocks (Ratpoints.Sieve.BlockPlan) and the counters and the decisions of the correction of the moduli (Ratpoints.Sieve.Correction) structures of their own; the bounds of the search intervals (the struct iv of the run, Ratpoints.Sieve.runSetup), the classes with their row offsets and den_bits passed to the run apart (Ratpoints.Sieve.sieveRun).)
- find_points.c:3750-3754: modelled by Ratpoints.Sieve.BlockPlan (deviation of Ratpoints.Sieve.BlockPlan: the segments as an array of pairs, for the pointers seg_lo and seg_hi; the first blocks with the number of the blocks at the end of the array, for seg_block and nblocks, which the C code keeps apart with the same value; no count nseg of the segments, an array having its size; natural numbers for the longs.)
- find_points.c:3755-3762: modelled by Ratpoints.Sieve.Correction (deviation of Ratpoints.Sieve.Correction: the decisions an array of structures, for the three arrays dec_block, dec_sp2 and dec_sp3 and their count ndec; the fields sp2, sp3 and adapt_at of the arguments, which adapt_primes writes and run_fold reads, kept here, since they are read only within the run, by run_fold and adapt_primes, after which line 5163 overwrites sp2 and sp3 with the ends of the last block folded; natural numbers for the longs and a UInt64 for the unsigned long.)
- find_points.c:3763-3763: modelled by Ratpoints.Sieve.Shape (deviation of Ratpoints.Sieve.Shape: an inductive type with the data of each shape, for the constant shape and the fields of rp_run that the shapes read, with these exceptions: the forbidden divisors as lists, for forb_ba and forbidden with no count nfb, a list having its length; the tables of the

Jacobi test as an option, for `ji`, `jtab` and `fast_jacobi`; with the words, the bit `RATPOINTS_USE_JACOBI` of the flags and the leading coefficient `args->cof[args->degree]`, which `run_block` reads through the field `args`, the run taking the rest of what it reads of the arguments as its own arguments (`Ratpoints.Sieve.sieveRun`); no copies of `use_c_long` and `lcf_long`, which only `jacobi1` reads (`Ratpoints.Sieve.jacobi`); the segments and the blocks (`Ratpoints.Sieve.BlockPlan`) and the counters and the decisions of the correction of the moduli (`Ratpoints.Sieve.Correction`) structures of their own; the bounds of the search intervals (the struct `iv` of the run, `Ratpoints.Sieve.runSetup`), the classes with their row offsets and `den_bits` passed to the run apart (`Ratpoints.Sieve.sieveRun`).)

- `find_points.c:3766-3774`: modelled by `Ratpoints.Sieve.floorSqrt` (deviation of `Ratpoints.Sieve.floorSqrt`: the exact square root of the natural numbers (`Nat.sqrt`), for the square root in `double` corrected by a step either way; natural numbers for the `long`s, so that `0` is the only case `n < 1`.)
- `find_points.c:3778-3806`: modelled by `Ratpoints.Sieve.runSetup` (deviation of `Ratpoints.Sieve.runSetup`: the shape returned with its data, for the fields of the run that `run_setup` fills, and Booleans and an option for the bits `RATPOINTS_USE_SQUARES`, `RATPOINTS_USE_SQUARES1`, `RATPOINTS_CHECK_DENOM` and `RATPOINTS_USE_JACOBI` of the flags; the bounds of the search intervals returned as an array with the shape, for the struct `iv` of the run with its pointers `bd` and `end`; the search intervals, the height bound and the largest denominator as arguments, for the fields of the arguments that `exact_bounds` reads; no count `nfb`, a list having its length; no copies of `use_c_long` and `lcf_long`, which only `jacobi1` reads (`Ratpoints.Sieve.jacobi`); the segments and the blocks made by `Ratpoints.Sieve.Shape.blockPlan`, and the counters and the decisions of the correction by `Ratpoints.Sieve.Correction.start`; no failure for lack of memory.)
- `find_points.c:3807-3811`: excluded (memory): the arrays of the segments of the run (their bounds and first blocks) in one allocation, which fails only for lack of memory, and before it the pointers to the other two arrays that run_clear frees set to NULL
- `find_points.c:3812-3837`: modelled by `Ratpoints.Sieve.Shape.segments` (deviation of `Ratpoints.Sieve.Shape.segments`: the segments as an array of pairs, for the arrays `seg_lo` and `seg_hi` of the run; quotients by 64 for the shifts by `LONG_SHIFT`; natural numbers for the `long`s, the steps of a segment counted only when `lo ≤ hi`, as the C code counts them; the count of the segments read from the shape, for `nseg`, which `run_setup` counts before (line 3794).)
- `find_points.c:3838-3842`: modelled by `Ratpoints.Sieve.blockLength` (deviation of `Ratpoints.Sieve.blockLength`: the conditional compilation `#ifdef RP_BLOCK_LEN` a constant of type `Option Nat` (`Ratpoints.blockLen`), none in the default build.)
- `find_points.c:3843-3850`: modelled by `Ratpoints.Sieve.Shape.blockPlan` (deviation of `Ratpoints.Sieve.Shape.blockPlan`: the blocks returned as a structure (`Ratpoints.Sieve.BlockPlan`), for the fields of the run that `run_setup` fills; natural numbers for the `long`s.)
- `find_points.c:3853-3856`: modelled by `Ratpoints.Sieve.Correction.start` (deviation of `Ratpoints.Sieve.Correction.start`: the state returned, for the fields of the run that `run_setup` sets.)
- `find_points.c:3857-3860`: excluded (memory): the arrays of the decisions of the correction of the moduli between the blocks in one allocation, which fails only for lack of memory

- find_points.c:3861-3863: modelled by Ratpoints.Sieve.Correction.start (deviation of Ratpoints.Sieve.Correction.start: the state returned, for the fields of the run that run_setup sets.)
- find_points.c:3866-3868: excluded (memory): the array of the search intervals of the run on the heap, whose allocation fails only for lack of memory
- find_points.c:3869-3870: modelled by Ratpoints.Sieve.runSetup (deviation of Ratpoints.Sieve.runSetup: the shape returned with its data, for the fields of the run that run_setup fills, and Booleans and an option for the bits RATPOINTS_USE_SQUARES, RATPOINTS_USE_SQUARES1, RATPOINTS_CHECK_DENOM and RATPOINTS_USE_JACOBI of the flags; the bounds of the search intervals returned as an array with the shape, for the struct iv of the run with its pointers bd and end; the search intervals, the height bound and the largest denominator as arguments, for the fields of the arguments that exact_bounds reads; no count nfb, a list having its length; no copies of use_c_long and lcf_long, which only jacobi1 reads (Ratpoints.Sieve.jacobi); the segments and the blocks made by Ratpoints.Sieve.Shape.blockPlan, and the counters and the decisions of the correction by Ratpoints.Sieve.Correction.start; no failure for lack of memory.)
- find_points.c:3871-3875: excluded (instrumentation): the report of the loop over the denominators under DEBUG at the end of run_setup: its shape, its segments and its blocks
- find_points.c:3876-3877: modelled by Ratpoints.Sieve.runSetup (deviation of Ratpoints.Sieve.runSetup: the shape returned with its data, for the fields of the run that run_setup fills, and Booleans and an option for the bits RATPOINTS_USE_SQUARES, RATPOINTS_USE_SQUARES1, RATPOINTS_CHECK_DENOM and RATPOINTS_USE_JACOBI of the flags; the bounds of the search intervals returned as an array with the shape, for the struct iv of the run with its pointers bd and end; the search intervals, the height bound and the largest denominator as arguments, for the fields of the arguments that exact_bounds reads; no count nfb, a list having its length; no copies of use_c_long and lcf_long, which only jacobi1 reads (Ratpoints.Sieve.jacobi); the segments and the blocks made by Ratpoints.Sieve.Shape.blockPlan, and the counters and the decisions of the correction by Ratpoints.Sieve.Correction.start; no failure for lack of memory.)
- find_points.c:3879-3883: excluded (memory): run_clear, which releases the arrays of the run
- find_points.c:3889-3898: modelled by Ratpoints.Sieve.runBlock (deviation of Ratpoints.Sieve.runBlock: the decision's entries and reciprocals cut as lists for each block, for the numbers sp2 and sp3 that the sieving thread keeps and through which sift and fill_bp_list read the arrays sieve_list and magics; the report returned with the delivery and the store, for the fields of the sieving thread that worker_stat copies into it after the block; the counters and the index of the decision, which worker_init clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that run_block reads; the delivery r passed and returned, for the pointers info and wk->quit and the count total that run_block keeps from zero and returns; the array bp_list of sp3_max entries on the stack, of which fill_bp_list fills the first sp2, a list made by Ratpoints.Sieve.siftDen; natural numbers for the longs.)

- find_points.c:3900-3904: modelled by Ratpoints.Sieve.BlockPlan.steps (deviation of Ratpoints.Sieve.BlockPlan.steps: the segment and the steps returned, for the locals of run_block; natural numbers for the longs, the difference hi - a truncated at 0, where the C code compares a long that is not negative for a block of the run.)
- find_points.c:3908-3910: modelled by Ratpoints.Sieve.findDecision (deviation of Ratpoints.Sieve.findDecision: the decisions an array of structures (Ratpoints.Sieve.Decision), for the three arrays dec_block, dec_sp2 and dec_sp3 and their count ndec, which the C code reads with an acquire load, for the threads that publish their decisions (RP_ACQUIRE), here a plain read; the index returned, for the local d; natural numbers for the longs.)
- find_points.c:3911-3911: modelled by Ratpoints.Sieve.runBlock (deviation of Ratpoints.Sieve.runBlock: the decision's entries and reciprocals cut as lists for each block, for the numbers sp2 and sp3 that the sieving thread keeps and through which sift and fill_bp_list read the arrays sieve_list and magics; the report returned with the delivery and the store, for the fields of the sieving thread that worker_stat copies into it after the block; the counters and the index of the decision, which worker_init clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that run_block reads; the delivery r passed and returned, for the pointers info and wk->quit and the count total that run_block keeps from zero and returns; the array bp_list of sp3_max entries on the stack, of which fill_bp_list fills the first sp2, a list made by Ratpoints.Sieve.siftDen; natural numbers for the longs.)
- find_points.c:3912-3912: modelled by Ratpoints.Sieve.findDecision (deviation of Ratpoints.Sieve.findDecision: the decisions an array of structures (Ratpoints.Sieve.Decision), for the three arrays dec_block, dec_sp2 and dec_sp3 and their count ndec, which the C code reads with an acquire load, for the threads that publish their decisions (RP_ACQUIRE), here a plain read; the index returned, for the local d; natural numbers for the longs.)
- find_points.c:3913-3915: modelled by Ratpoints.Sieve.runBlock (deviation of Ratpoints.Sieve.runBlock: the decision's entries and reciprocals cut as lists for each block, for the numbers sp2 and sp3 that the sieving thread keeps and through which sift and fill_bp_list read the arrays sieve_list and magics; the report returned with the delivery and the store, for the fields of the sieving thread that worker_stat copies into it after the block; the counters and the index of the decision, which worker_init clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that run_block reads; the delivery r passed and returned, for the pointers info and wk->quit and the count total that run_block keeps from zero and returns; the array bp_list of sp3_max entries on the stack, of which fill_bp_list fills the first sp2, a list made by Ratpoints.Sieve.siftDen; natural numbers for the longs.)
- find_points.c:3917-3918: modelled by Ratpoints.Sieve.Shape.blockLoop (deviation of Ratpoints.Sieve.Shape.blockLoop: a function of its own for the switch of run_block over the shapes, the step an argument, for the two lines of each shape that sift a denominator (Ratpoints.Sieve.siftDen); the first and last words of the run, which the masks of the word shape compare with, read from the first segment of plan, as the C code reads seg_lo[0] and seg_hi[0]; natural numbers for the longs.)
- find_points.c:3919-3925: modelled by Ratpoints.Sieve.squaresBlock (deviation of Ratpoints.Sieve.squaresBlock: the denominator sifted by the argument step, for the two calls

of each shape (Ratpoints.Sieve.siftDen); the pattern of the class read as its one word, for EXT0(cl->bits); the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators k ^ 2 being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_squaresDens); no output under DEBUG.)

- find_points.c:3926-3927: modelled by Ratpoints.Sieve.siftDen (deviation of Ratpoints.Sieve.siftDen: a function of its own for the two calls that each shape makes for a denominator, the residues a list for the array bp_list, which sift reads through its pointer; the state of the block passed and returned (Ratpoints.Sieve.BlockState), for the pointers info and wk->quit, the count total that run_block keeps, the arrays sieve of the entries and the counters of the sieving thread.)

- find_points.c:3928-3935: modelled by Ratpoints.Sieve.squaresBlock (deviation of Ratpoints.Sieve.squaresBlock: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen); the pattern of the class read as its one word, for EXT0(cl->bits); the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators k ^ 2 being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_squaresDens); no output under DEBUG.)

- find_points.c:3936-3936: modelled by Ratpoints.Sieve.Shape.blockLoop (deviation of Ratpoints.Sieve.Shape.blockLoop: a function of its own for the switch of run_block over the shapes, the step an argument, for the two lines of each shape that sift a denominator (Ratpoints.Sieve.siftDen); the first and last words of the run, which the masks of the word shape compare with, read from the first segment of plan, as the C code reads seg_lo[0] and seg_hi[0]; natural numbers for the longs.)

- find_points.c:3937-3943: modelled by Ratpoints.Sieve.squares1Block (deviation of Ratpoints.Sieve.squares1Block: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen); the valuation test a function of its own; the pattern of the class read as its one word; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators dv k ^ 2 being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_squares1Dens); no output under DEBUG.)

- find_points.c:3944-3953: modelled by Ratpoints.Sieve.denInfoTest (deviation of Ratpoints.Sieve.denInfoTest: the test returned, for the flag flag; natural numbers for the ints and longs.)

- find_points.c:3954-3954: modelled by Ratpoints.Sieve.squares1Block (deviation of Ratpoints.Sieve.squares1Block: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen); the valuation test a function of its own; the pattern of the class read as its one word; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators dv k ^ 2 being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_squares1Dens); no output under DEBUG.)

- find_points.c:3955-3956: modelled by Ratpoints.Sieve.siftDen (deviation of Ratpoints.Sieve.siftDen: a function of its own for the two calls that each shape makes for a denominator, the residues a list for the array bp_list, which sift reads through its pointer; the state of the block passed and returned (Ratpoints.Sieve.BlockState), for the pointers info and wk->quit, the count total that run_block keeps, the arrays sieve of the entries and the counters of the sieving thread.)

- find_points.c:3957-3965: modelled by Ratpoints.Sieve.squares1Block (deviation of Ratpoints.Sieve.squares1Block: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen); the valuation test a function of its own; the pattern of the class read as its one word; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators dv k ^ 2 being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_squares1Dens); no output under DEBUG.)

- find_points.c:3966-3966: modelled by Ratpoints.Sieve.Shape.blockLoop (deviation of Ratpoints.Sieve.Shape.blockLoop: a function of its own for the switch of run_block over the shapes, the step an argument, for the two lines of each shape that sift a denominator (Ratpoints.Sieve.siftDen); the first and last words of the run, which the masks of the word shape compare with, read from the first segment of plan, as the C code reads seg_lo[0] and seg_hi[0]; natural numbers for the longs.)

- find_points.c:3967-3980: modelled by Ratpoints.Sieve.wordsBlock (deviation of Ratpoints.Sieve.wordsBlock: the denominator sifted by the argument step, for the two calls of each shape, and the tests of the word shape the argument test (Ratpoints.Sieve.wordsTest); the forbidden divisors an array, for the pointer run->forb_ba and their count nfb; the product 64 w for the shift by LONG_SHIFT; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the products 64 w and the denominators being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens).)

- find_points.c:3981-3981: modelled by Ratpoints.Sieve.forbStarts (deviation of Ratpoints.Sieve.forbStarts: indices into the words of the divisors, for the pointers cur into the arrays from start to end; a Boolean for the int small; natural numbers for the longs.)

- find_points.c:3982-3982: modelled by Ratpoints.Sieve.wordsBlock (deviation of Ratpoints.Sieve.wordsBlock: the denominator sifted by the argument step, for the two calls of each shape, and the tests of the word shape the argument test (Ratpoints.Sieve.wordsTest); the forbidden divisors an array, for the pointer run->forb_ba and their count nfb; the product 64 w for the shift by LONG_SHIFT; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the products 64 w and the denominators being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens).)

- find_points.c:3983-3988: modelled by Ratpoints.Sieve.forbStarts (deviation of Ratpoints.Sieve.forbStarts: indices into the words of the divisors, for the pointers cur into the arrays from start to end; a Boolean for the int small; natural numbers for the longs.)

- find_points.c:3989-3989: modelled by Ratpoints.Sieve.wordsBlock (deviation of Ratpoints.Sieve.wordsBlock: the denominator sifted by the argument step, for the two calls of each shape, and the tests of the word shape the argument test (Ratpoints.Sieve.wordsTest); the forbidden divisors an array, for the pointer run->forb_ba and their count nfb; the product 64 w for the shift by LONG_SHIFT; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the products 64 w and the denominators being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens).)

- find_points.c:3990-3990: modelled by Ratpoints.Sieve.denWord (deviation of Ratpoints.Sieve.denWord: indices into the words of the divisors, for the pointers cur, and the end of the words of a divisor its prime p, for the pointer end; the remainders by 64 for the masks by LONG_MASK; the positions returned with the word; natural numbers for the longs; no output under DEBUG.)

- find_points.c:3991-3992: modelled by Ratpoints.Sieve.wordsBlock (deviation of Ratpoints.Sieve.wordsBlock: the denominator sifted by the argument step, for the two calls of each shape, and the tests of the word shape the argument test (Ratpoints.Sieve.wordsTest); the forbidden divisors an array, for the pointer run->forb_ba and their count nfb; the product 64 w for the shift by LONG_SHIFT; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the products 64 w and the denominators being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens).)
- find_points.c:3993-4006: modelled by Ratpoints.Sieve.denWord (deviation of Ratpoints.Sieve.denWord: indices into the words of the divisors, for the pointers cur, and the end of the words of a divisor its prime p, for the pointer end; the remainders by 64 for the masks by LONG_MASK; the positions returned with the word; natural numbers for the longs; no output under DEBUG.)
- find_points.c:4007-4009: modelled by Ratpoints.Sieve.walkDens (deviation of Ratpoints.Sieve.walkDens: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen), and the tests of the word shape the argument test, for the code of Ratpoints.Sieve.wordsTest in the loop; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); the class of the denominator read by Ratpoints.Sieve.siftDen; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators base + j being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens); no output under DEBUG.)
- find_points.c:4010-4011: modelled by Ratpoints.Sieve.wordsTest (deviation of Ratpoints.Sieve.wordsTest: the binary algorithm of jacobi also where the run calls jacobi1 on the long lcf_long (Ratpoints.Sieve.jacobi); bit v_p(b) of the mask of a natural number, false beyond its bits (Nat.testBit), for the shift and the mask of the unsigned long; the test returned, where the loop goes on to the sieve or to the next denominator; natural numbers for the longs; no output under DEBUG.)
- find_points.c:4012-4012: modelled by Ratpoints.Sieve.walkDens (deviation of Ratpoints.Sieve.walkDens: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen), and the tests of the word shape the argument test, for the code of Ratpoints.Sieve.wordsTest in the loop; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); the class of the denominator read by Ratpoints.Sieve.siftDen; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators base + j being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens); no output under DEBUG.)
- find_points.c:4013-4041: modelled by Ratpoints.Sieve.wordsTest (deviation of Ratpoints.Sieve.wordsTest: the binary algorithm of jacobi also where the run calls jacobi1 on the long lcf_long (Ratpoints.Sieve.jacobi); bit v_p(b) of the mask of a natural number, false beyond its bits (Nat.testBit), for the shift and the mask of the unsigned long; the test returned, where the loop goes on to the sieve or to the next denominator; natural numbers for the longs; no output under DEBUG.)
- find_points.c:4042-4043: modelled by Ratpoints.Sieve.siftDen (deviation of Ratpoints.Sieve.siftDen: a function of its own for the two calls that each shape makes for a denominator, the residues a list for the array bp_list, which sift reads through its pointer; the state of the block passed and returned (Ratpoints.Sieve.BlockState), for the pointers info and wk->quit, the count total that run_block keeps, the arrays sieve of the entries and the counters of the sieving thread.)

- find_points.c:4044-4053: modelled by Ratpoints.Sieve.walkDens (deviation of Ratpoints.Sieve.walkDens: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen), and the tests of the word shape the argument test, for the code of Ratpoints.Sieve.wordsTest in the loop; the lowest set bit by the loop of RP_CTZL (Ratpoints.Sieve.ctz); the class of the denominator read by Ratpoints.Sieve.siftDen; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the denominators base + j being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens); no output under DEBUG.)
- find_points.c:4054-4058: modelled by Ratpoints.Sieve.wordsBlock (deviation of Ratpoints.Sieve.wordsBlock: the denominator sifted by the argument step, for the two calls of each shape, and the tests of the word shape the argument test (Ratpoints.Sieve.wordsTest); the forbidden divisors an array, for the pointer run->forb_ba and their count nfb; the product 64 w for the shift by LONG_SHIFT; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs, the products 64 w and the denominators being at most b_high and so longs for a long b_high (Ratpoints.Sieve.fitsLong_of_mem_wordsDens).)
- find_points.c:4059-4059: modelled by Ratpoints.Sieve.Shape.blockLoop (deviation of Ratpoints.Sieve.Shape.blockLoop: a function of its own for the switch of run_block over the shapes, the step an argument, for the two lines of each shape that sift a denominator (Ratpoints.Sieve.siftDen); the first and last words of the run, which the masks of the word shape compare with, read from the first segment of plan, as the C code reads seg_lo[0] and seg_hi[0]; natural numbers for the longs.)
- find_points.c:4060-4065: modelled by Ratpoints.Sieve.allBlock (deviation of Ratpoints.Sieve.allBlock: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen); the pattern of the class read as its one word; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs; no output under DEBUG.)
- find_points.c:4066-4067: modelled by Ratpoints.Sieve.siftDen (deviation of Ratpoints.Sieve.siftDen: a function of its own for the two calls that each shape makes for a denominator, the residues a list for the array bp_list, which sift reads through its pointer; the state of the block passed and returned (Ratpoints.Sieve.BlockState), for the pointers info and wk->quit, the count total that run_block keeps, the arrays sieve of the entries and the counters of the sieving thread.)
- find_points.c:4068-4074: modelled by Ratpoints.Sieve.allBlock (deviation of Ratpoints.Sieve.allBlock: the denominator sifted by the argument step, for the two calls of each shape (Ratpoints.Sieve.siftDen); the pattern of the class read as its one word; the state of the block passed and returned (Ratpoints.Sieve.BlockState); natural numbers for the longs; no output under DEBUG.)
- find_points.c:4075-4076: modelled by Ratpoints.Sieve.Shape.blockLoop (deviation of Ratpoints.Sieve.Shape.blockLoop: a function of its own for the switch of run_block over the shapes, the step an argument, for the two lines of each shape that sift a denominator (Ratpoints.Sieve.siftDen); the first and last words of the run, which the masks of the word shape compare with, read from the first segment of plan, as the C code reads seg_lo[0] and seg_hi[0]; natural numbers for the longs.)
- find_points.c:4077-4078: modelled by Ratpoints.Sieve.runBlock (deviation of Ratpoints.Sieve.runBlock: the decision's entries and reciprocals cut as lists for each block, for the numbers sp2 and sp3 that the sieving thread keeps and through which sift and fill_bp_list read the arrays sieve_list and magics; the report returned with the delivery and the store, for the fields

of the sieving thread that `worker_stat` copies into it after the block; the counters and the index of the decision, which `worker_init` clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that `run_block` reads; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `run_block` keeps from zero and returns; the array `bp_list` of `sp3_max` entries on the stack, of which `fill_bp_list` fills the first `sp2`, a list made by `Ratpoints.Sieve.siftDen`; natural numbers for the `longs`.)

- `find_points.c:4082-4086`: modelled by `Ratpoints.Sieve.BlockStat` (deviation of `Ratpoints.Sieve.BlockStat`: the counters as a structure, for the six fields; natural numbers for the `longs`.)

- `find_points.c:4088-4093`: modelled by `Ratpoints.Sieve.runBlock` (deviation of `Ratpoints.Sieve.runBlock`: the decision's entries and reciprocals cut as lists for each block, for the numbers `sp2` and `sp3` that the sieving thread keeps and through which `sift` and `fill_bp_list` read the arrays `sieve_list` and `magics`; the report returned with the delivery and the store, for the fields of the sieving thread that `worker_stat` copies into it after the block; the counters and the index of the decision, which `worker_init` clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that `run_block` reads; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `run_block` keeps from zero and returns; the array `bp_list` of `sp3_max` entries on the stack, of which `fill_bp_list` fills the first `sp2`, a list made by `Ratpoints.Sieve.siftDen`; natural numbers for the `longs`.)

- `find_points.c:4102-4106`: modelled by `Ratpoints.Sieve.runFold` (deviation of `Ratpoints.Sieve.runFold`: the state of the correction passed and returned (`Ratpoints.Sieve.Correction`), for the fields of the run and of the arguments; the decision appended to the array, for the store into the three arrays and the release store of their count (`RP_PUBLISH`), for the threads that read them, here a plain store; natural numbers for the `longs`.)

- `find_points.c:4107-4113`: modelled by `Ratpoints.Sieve.RunCounts.fold` (deviation of `Ratpoints.Sieve.RunCounts.fold`: a function of its own for this part of `run_fold`; the end `sp2` passed, for the entry `dec_sp2[st->dec]` that `run_fold` reads; the sums of `UInt64`s, wrapping as the `unsigned longs` do.)

- `find_points.c:4114-4127`: modelled by `Ratpoints.Sieve.runFold` (deviation of `Ratpoints.Sieve.runFold`: the state of the correction passed and returned (`Ratpoints.Sieve.Correction`), for the fields of the run and of the arguments; the decision appended to the array, for the store into the three arrays and the release store of their count (`RP_PUBLISH`), for the threads that read them, here a plain store; natural numbers for the `longs`.)

- `find_points.c:4129-4519`: excluded (threads): the pool of threads (the lock of the tables of rows, the workers, the collection of the points of a block, run_threaded) and its stubs for a build without threads; the pool sieves the blocks apart and delivers them in their order, a block not starting before the block RP_ADAPT_LAG + 1 before it is folded, so that the decision of each block and the output are those of the blocks sifted in turn on one thread, which the model does, but for the failure of the search (RATPOINTS_NO_MEMORY) when a thread of the pool has no memory to store a point of its block before the delivery reaches it (lines 4245-4250, 4482, 4501), a failure that the model, allocating nothing, cannot meet

- find_points.c:4523-4528: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- find_points.c:4542-4545: excluded (interface): the input fields of the arguments that find_points_work saves, so that a caller who reuses the arguments for the next curve does not find the normalizations and choices of the search in them; the model writes nothing back

- find_points.c:4546-4546: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS;

RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- find_points.c:4551-4552: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2

`^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4557-4559`: modelled by `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the

working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- find_points.c:4560-4563: excluded (interface): the input fields of the arguments that find_points_work puts back

- find_points.c:4564-4565: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args-

`>flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4567-4571`: modelled by `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4572-4572`: excluded (threads): the worker of the calling thread, the state with which a thread sieves its blocks (its survivors, its third stage); the model sieves the blocks in turn, without threads, and passes what the sieve needs as arguments

- `find_points.c:4573-4573`: modelled by `Ratpoints.sieving` (deviation of `Ratpoints.sieving`: a function of its own for the block of the sieving in `find_points_work_1`; the range of the

denominators and the search intervals as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments that `run_setup` and the run read; the flags of the denominators as Booleans and an option, for the bits of `args->flags` and the arrays `args->den_info` and `args->divisors` that `run_setup` reads; the array of the row offsets made with zeros, for the array on the stack, whose entries the C code leaves unset until `class_offsets` writes them, with one offset in each row for each of the `sp3_max` sieve entries of the run, the reserve of the correction included; the delivery passed and returned, for the pointers `info` and `wk.quit` and the count `total`, and the ends of the second and the third stage of the last block returned with it, for the fields `sp2` and `sp3` of the arguments that the run writes; no sieving thread made (`worker_init`, `worker_clear`), whose fields the components that the run calls take as arguments, and no pool of threads (excluded with the threads); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4577-4585`: modelled by `Ratpoints.findPointsWork` (deviation of `Ratpoints.findPointsWork`: a function of its own for the part of `find_points_work_1` after the test of the leading zeros; the refusals returned apart from the outcome, with the data of the callback, for the negative codes that the C code returns, the caller keeping its data behind the pointer `info`; the outcome of the return of line 4889, which leaves the flags of the squares clear and no interval, with the data `info`, the count `0`, the flag of the reversal, no interval and no moduli used, the flag being in fact clear there (the cases 1 and 3 of the reversal check leave an odd degree, and in case 2 the reversed polynomial takes the value `c_d > 0` at `0`, in the default interval, which the Sturm step keeps); the test of the `longs` (`Ratpoints.Setup.useCLong`), which the C code makes between the reversal check and the search intervals, made by the search, where it is used; the report under `RATPOINTS_VERBOSE` printed through the print channel, that of the reversal check and of the Sturm step after them, from their results, where the C code prints in their course; no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4588-4589`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4595-4596`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`:

the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4599-4600`: modelled by `Ratpoints.Setup.useCLong` (deviation of `Ratpoints.Setup.useCLong`: no array `c_long`: the coefficients stay integers, which `Ratpoints.Sieve.modLong` reads as `long`s, as `mpz_get_si` gives them exactly when they fit; the flag is returned, for the `int` that the loop sets, and the loop, which stops at the first coefficient that is not a `long`, is `List.all`.)

- `find_points.c:4602-4602`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4603-4604`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag

of find_points_work_1; the flags of the denominators as Booleans, for the bits of args->flags; the divisors of the leading coefficient with their records as lists, for the arrays args->den_info and args->divisors of the arguments; the square test lcfsq of line 4906 made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), for the bits that find_points_work_1 sets in args->flags, the intervals that it leaves in args->domain and the fields sp1_used, sp2_used and sp3_used that the part after the choice sets (Ratpoints.chooseAndSearch); the report under RATPOINTS_VERBOSE printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under DEBUG.)

- find_points.c:4606-4606: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the

standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4607-4607`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4610-4612`: excluded (memory): the reset of the bump allocators of the buffers that find_points_init allocates, from which the set-up and the sieve take their memory; the model allocates nothing itself

- `find_points.c:4614-4616`: excluded (instrumentation): the message under DEBUG at the start of find_points_work_1

- `find_points.c:4618-4623`: excluded (interface): the checks of the coefficient array and of the work array that find_points_init sets up, which only a caller of the library can fail (the work array is missing when find_points_init refused the degree and the caller then gave an admissible one, since find_points_work refuses an inadmissible degree first at lines 4557-4558, or for lack of memory, which the model cannot meet)

- `find_points.c:4624-4633`: modelled by `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them,

where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:4634-4634`: modelled by `Ratpoints.Sieve.pointsAtInfinity` (deviation of `Ratpoints.Sieve.pointsAtInfinity`: the flags `fl` as an argument, for the bits `RATPOINTS_REVERSED`, `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the flags of the search (`args->flags`); the delivery `r` passed and returned, for the pointer `info`, the local flag `quit` and the count `total` of `find_points_work_1`; the return when the callback stops the search (lines 5100-5103) made by the caller (`Ratpoints.chooseAndSearch`), which also prints the line before them under `RATPOINTS_VERBOSE` (`Ratpoints.Report.atInfinity`), the line that there are points at infinity coming with the report of the tests on the denominators (`Ratpoints.Report.setup`); no output under `DEBUG`; the square test of `c_d` and the decision whether there are points at infinity are made on the list that it is given, after the reversal check, when the points are delivered, where the C code computes `lcfsq` before the check (line 4634), which clears it in case 2 (line 4770), the only reversal after which the degree is even, and leaves it, unread, with the odd degree that cases 1 and 3 leave, and sets `point_at_infty` before the exits of the search (lines 4894-4899); the square root by `Nat.sqrt`, for `mpz_sqrt`.)
- `find_points.c:4636-4638`: excluded (instrumentation): the message under DEBUG before the checks of the parameters
- `find_points.c:4640-4641`: modelled by `Ratpoints.Options.normalize` (deviation of `Ratpoints.Options.normalize`: the normalized options returned, for the fields of the arguments

that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563), which the caller does not see either; no clamp of the number of the search intervals, which a list cannot make negative; the bit `RATPOINTS_NO_REVERSE_AUTO`, which the C code sets in the flags and the reversal check tests with `RATPOINTS_NO_REVERSE` (line 4739), as the switch `reverse` cleared; integers for the `long`s.)

- `find_points.c:4643-4649`: modelled by `Ratpoints.SieveOptions.normalize` (deviation of `Ratpoints.SieveOptions.normalize`: the normalized options returned, for the fields of the arguments that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563); the array size normalized with the height bound where it is read (`Ratpoints.arraySizeAt`); no enlargement of the buffer of the rows (line 4654), which the model does not have; integers for the `long`s.)

- `find_points.c:4654-4654`: excluded (memory): the enlargement of ba_buffer for more primes than find_points_init sized it for, which fails only for lack of memory

- `find_points.c:4655-4656`: modelled by `Ratpoints.SieveOptions.normalize` (deviation of `Ratpoints.SieveOptions.normalize`: the normalized options returned, for the fields of the arguments that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563); the array size normalized with the height bound where it is read (`Ratpoints.arraySizeAt`); no enlargement of the buffer of the rows (line 4654), which the model does not have; integers for the `long`s.)

- `find_points.c:4658-4658`: modelled by `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and

its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- find_points.c:4659-4661: modelled by `Ratpoints.Options.normalize` (deviation of `Ratpoints.Options.normalize`: the normalized options returned, for the fields of the arguments that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563), which the caller does not see either; no clamp of the number of the search intervals, which a list cannot make negative; the bit `RATPOINTS_NO_REVERSE_AUTO`, which the C code sets in the flags and the reversal check tests with `RATPOINTS_NO_REVERSE` (line 4739), as the switch `reverse` cleared; integers for the `long`s.)

- find_points.c:4662-4665: modelled by `Ratpoints.SieveOptions.normalize` (deviation of `Ratpoints.SieveOptions.normalize`: the normalized options returned, for the fields of the arguments that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563); the array size normalized with the height bound where it is read (`Ratpoints.arraySizeAt`); no enlargement of the buffer of the rows (line 4654), which the model does not have; integers for the `long`s.)

- find_points.c:4666-4677: modelled by `Ratpoints.arraySizeAt` (deviation of `Ratpoints.arraySizeAt`: a `Nat` for the `long` field, so that only `0` stands for the default, which the library takes for every value `≤ 0`; `64` for `LONG_LENGTH`; and the macro `CEIL` of a natural number `x` by `d` as `(x + d - 1) / d`, which is its value for every `x ≥ 0`.)

- find_points.c:4678-4689: modelled by `Ratpoints.Options.normalize` (deviation of `Ratpoints.Options.normalize`: the normalized options returned, for the fields of the arguments that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563), which the caller does not see either; no clamp of the number of the search intervals, which a list cannot make negative; the bit `RATPOINTS_NO_REVERSE_AUTO`, which the C code sets in the flags and the reversal check tests with `RATPOINTS_NO_REVERSE` (line 4739), as the switch `reverse` cleared; integers for the `long`s.)

- find_points.c:4691-4725: modelled by `Ratpoints.Report.arguments` (deviation of `Ratpoints.Report.arguments`: the fields of the arguments as arguments, as the search normalizes them: the options of the sieve as `Ratpoints.SieveOptions.normalize` gives them, the range of the denominators, the depth and the switch of the reversal as `Ratpoints.Options.normalize` gives them, the switch for the bits `RATPOINTS_NO_REVERSE` and `RATPOINTS_NO_REVERSE_AUTO` of the flags, and the bits `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_JACOBI` as Booleans; integers for the `long`s.)

- find_points.c:4727-4730: excluded (instrumentation): the message under DEBUG before the reversal check

- find_points.c:4732-4739: modelled by `Ratpoints.Setup.reversal` (deviation of `Ratpoints.Setup.reversal`: a new list for the reversal in place, and nothing written back to the caller: the reversed coefficients, the degree and the flag `RATPOINTS_REVERSED` are returned instead; the switch

for the test of the flags RATPOINTS_NO_REVERSE and RATPOINTS_NO_REVERSE_AUTO, which is on when neither is set, as the normalized options give it (Ratpoints.Options.normalize: off for a search region or a range of the denominators); the square test of c_d made here, where the C code reads the flag lcfsq that the set-up computes before (line 4634), and no flag lcfsq for the reversed list, which case 2 clears, since Ratpoints.Sieve.pointsAtInfinity tests the leading coefficient of the list it is given; integers and their absolute values for the GMP integers, with no scratch integers work[0] and work[1]; the report under RATPOINTS_VERBOSE printed by the caller from the result (Ratpoints.Report.reversal).)

- find_points.c:4740-4743: modelled by Ratpoints.Report.reversal (deviation of Ratpoints.Report.reversal: one text, from the result of the check, for the messages that the C code prints in its course, the case of a reversal read from the degree and the constant term as the check decides it: case 1 for even d and $c_0 = 0$, case 2 for even d otherwise and case 3 for odd d.)

- find_points.c:4744-4756: modelled by Ratpoints.Setup.reversal (deviation of Ratpoints.Setup.reversal: a new list for the reversal in place, and nothing written back to the caller: the reversed coefficients, the degree and the flag RATPOINTS_REVERSED are returned instead; the switch for the test of the flags RATPOINTS_NO_REVERSE and RATPOINTS_NO_REVERSE_AUTO, which is on when neither is set, as the normalized options give it (Ratpoints.Options.normalize: off for a search region or a range of the denominators); the square test of c_d made here, where the C code reads the flag lcfsq that the set-up computes before (line 4634), and no flag lcfsq for the reversed list, which case 2 clears, since Ratpoints.Sieve.pointsAtInfinity tests the leading coefficient of the list it is given; integers and their absolute values for the GMP integers, with no scratch integers work[0] and work[1]; the report under RATPOINTS_VERBOSE printed by the caller from the result (Ratpoints.Report.reversal).)

- find_points.c:4757-4758: modelled by Ratpoints.Report.reversal (deviation of Ratpoints.Report.reversal: one text, from the result of the check, for the messages that the C code prints in its course, the case of a reversal read from the degree and the constant term as the check decides it: case 1 for even d and $c_0 = 0$, case 2 for even d otherwise and case 3 for odd d.)

- find_points.c:4759-4770: modelled by Ratpoints.Setup.reversal (deviation of Ratpoints.Setup.reversal: a new list for the reversal in place, and nothing written back to the caller: the reversed coefficients, the degree and the flag RATPOINTS_REVERSED are returned instead; the switch for the test of the flags RATPOINTS_NO_REVERSE and RATPOINTS_NO_REVERSE_AUTO, which is on when neither is set, as the normalized options give it (Ratpoints.Options.normalize: off for a search region or a range of the denominators); the square test of c_d made here, where the C code reads the flag lcfsq that the set-up computes before (line 4634), and no flag lcfsq for the reversed list, which case 2 clears, since Ratpoints.Sieve.pointsAtInfinity tests the leading coefficient of the list it is given; integers and their absolute values for the GMP integers, with no scratch integers work[0] and work[1]; the report under RATPOINTS_VERBOSE printed by the caller from the result (Ratpoints.Report.reversal).)

- find_points.c:4771-4774: modelled by Ratpoints.Report.reversal (deviation of Ratpoints.Report.reversal: one text, from the result of the check, for the messages that the C code prints in its course, the case of a reversal read from the degree and the constant term as the check decides it: case 1 for even d and $c_0 = 0$, case 2 for even d otherwise and case 3 for odd d.)

- find_points.c:4775-4790: modelled by Ratpoints.Setup.reversal (deviation of Ratpoints.Setup.reversal: a new list for the reversal in place, and nothing written back to the caller: the reversed coefficients, the degree and the flag RATPOINTS_REVERSED are returned instead; the switch

for the test of the flags `RATPOINTS_NO_REVERSE` and `RATPOINTS_NO_REVERSE_AUTO`, which is on when neither is set, as the normalized options give it (`Ratpoints.Options.normalize`: off for a search region or a range of the denominators); the square test of `c_d` made here, where the C code reads the flag `lcfsq` that the set-up computes before (line 4634), and no flag `lcfsq` for the reversed list, which case 2 clears, since `Ratpoints.Sieve.pointsAtInfinity` tests the leading coefficient of the list it is given; integers and their absolute values for the GMP integers, with no scratch integers `work[0]` and `work[1]`; the report under `RATPOINTS_VERBOSE` printed by the caller from the result (`Ratpoints.Report.reversal`).)

- `find_points.c:4791-4801`: modelled by `Ratpoints.Report.reversal` (deviation of `Ratpoints.Report.reversal`: one text, from the result of the check, for the messages that the C code prints in its course, the case of a reversal read from the degree and the constant term as the check decides it: case 1 for even `d` and $c_0 = 0$, case 2 for even `d` otherwise and case 3 for odd `d`.)
- `find_points.c:4802-4803`: modelled by `Ratpoints.Setup.reversal` (deviation of `Ratpoints.Setup.reversal`: a new list for the reversal in place, and nothing written back to the caller: the reversed coefficients, the degree and the flag `RATPOINTS_REVERSED` are returned instead; the switch for the test of the flags `RATPOINTS_NO_REVERSE` and `RATPOINTS_NO_REVERSE_AUTO`, which is on when neither is set, as the normalized options give it (`Ratpoints.Options.normalize`: off for a search region or a range of the denominators); the square test of `c_d` made here, where the C code reads the flag `lcfsq` that the set-up computes before (line 4634), and no flag `lcfsq` for the reversed list, which case 2 clears, since `Ratpoints.Sieve.pointsAtInfinity` tests the leading coefficient of the list it is given; integers and their absolute values for the GMP integers, with no scratch integers `work[0]` and `work[1]`; the report under `RATPOINTS_VERBOSE` printed by the caller from the result (`Ratpoints.Report.reversal`).)
- `find_points.c:4804-4807`: modelled by `Ratpoints.Report.reversal` (deviation of `Ratpoints.Report.reversal`: one text, from the result of the check, for the messages that the C code prints in its course, the case of a reversal read from the degree and the constant term as the check decides it: case 1 for even `d` and $c_0 = 0$, case 2 for even `d` otherwise and case 3 for odd `d`.)
- `find_points.c:4809-4812`: excluded (instrumentation): the message under DEBUG after the reversal check when it reverses
- `find_points.c:4814-4823`: modelled by `Ratpoints.Setup.useCLong` (deviation of `Ratpoints.Setup.useCLong`: no array `c_long`: the coefficients stay integers, which `Ratpoints.Sieve.modLong` reads as `long`s, as `mpz_get_si` gives them exactly when they fit; the flag is returned, for the `int` that the loop sets, and the loop, which stops at the first coefficient that is not a `long`, is `List.all`.)
- `find_points.c:4825-4827`: excluded (instrumentation): the message under DEBUG before the search region and the Sturm step
- `find_points.c:4829-4829`: modelled by `Ratpoints.searchIntervals` (deviation of `Ratpoints.searchIntervals`: the intervals returned as a list, for the array `domain` and the number `num_inter` of the arguments, which the C code sets to the default interval in place; none for the code `RATPOINTS_BAD_ARGS` that the C code returns (`Ratpoints.Refusal.badArgs`); the test of a null pointer `domain` (line 4830) excluded, the interface of the library; natural numbers for the `long` index.)
- `find_points.c:4830-4830`: excluded (interface): the check of the array of search intervals, which only a caller of the library can fail

- find_points.c:4831-4855: modelled by Ratpoints.searchIntervals (deviation of Ratpoints.searchIntervals: the intervals returned as a list, for the array domain and the number num_inter of the arguments, which the C code sets to the default interval in place; none for the code RATPOINTS_BAD_ARGS that the C code returns (Ratpoints.Refusal.badArgs); the test of a null pointer domain (line 4830) excluded, the interface of the library; natural numbers for the long index.)
- find_points.c:4857-4859: modelled by Ratpoints.sturmRegion (deviation of Ratpoints.sturmRegion: the result as a refusal, none, or the intervals with the flag sturm_empty, for the codes that the C code returns at lines 4879 and 4889, the intervals that it leaves in domain with their number num_inter, and its local flag sturm_empty; the square test lcfsq made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; no failure for lack of memory (RATPOINTS_NO_MEMORY), which the model cannot meet; the report under RATPOINTS_VERBOSE printed by the caller from the result (Ratpoints.Report.sturmStep).)
- find_points.c:4860-4861: modelled by Ratpoints.Report.sturmStep (deviation of Ratpoints.Report.sturmStep: one text, from the result of the step, for the line before the step and the message after it, which the C code decides by the code that _ratpoints_compute_sturm returns: negative for the refusal, 0 when the step keeps no interval, and the number of the intervals kept otherwise.)
- find_points.c:4862-4863: modelled by Ratpoints.sturmRegion (deviation of Ratpoints.sturmRegion: the result as a refusal, none, or the intervals with the flag sturm_empty, for the codes that the C code returns at lines 4879 and 4889, the intervals that it leaves in domain with their number num_inter, and its local flag sturm_empty; the square test lcfsq made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; no failure for lack of memory (RATPOINTS_NO_MEMORY), which the model cannot meet; the report under RATPOINTS_VERBOSE printed by the caller from the result (Ratpoints.Report.sturmStep).)
- find_points.c:4864-4878: modelled by Ratpoints.Report.sturmStep (deviation of Ratpoints.Report.sturmStep: one text, from the result of the step, for the line before the step and the message after it, which the C code decides by the code that _ratpoints_compute_sturm returns: negative for the refusal, 0 when the step keeps no interval, and the number of the intervals kept otherwise.)
- find_points.c:4879-4892: modelled by Ratpoints.sturmRegion (deviation of Ratpoints.sturmRegion: the result as a refusal, none, or the intervals with the flag sturm_empty, for the codes that the C code returns at lines 4879 and 4889, the intervals that it leaves in domain with their number num_inter, and its local flag sturm_empty; the square test lcfsq made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; no failure for lack of memory (RATPOINTS_NO_MEMORY), which the model cannot meet; the report under RATPOINTS_VERBOSE printed by the caller from the result (Ratpoints.Report.sturmStep).)
- find_points.c:4894-4896: modelled by Ratpoints.Sieve.pointsAtInfinity (deviation of Ratpoints.Sieve.pointsAtInfinity: the flags fl as an argument, for the bits RATPOINTS_REVERSED,

RATPOINTS_NO_CHECK and RATPOINTS_NO_Y of the flags of the search (args->flags); the delivery r passed and returned, for the pointer info, the local flag quit and the count total of find_points_work_1; the return when the callback stops the search (lines 5100-5103) made by the caller (Ratpoints.chooseAndSearch), which also prints the line before them under RATPOINTS_VERBOSE (Ratpoints.Report.atInfinity), the line that there are points at infinity coming with the report of the tests on the denominators (Ratpoints.Report.setup); no output under DEBUG; the square test of c_d and the decision whether there are points at infinity are made on the list that it is given, after the reversal check, when the points are delivered, where the C code computes lcfsq before the check (line 4634), which clears it in case 2 (line 4770), the only reversal after which the degree is even, and leaves it, unread, with the odd degree that cases 1 and 3 leave, and sets point_at_infty before the exits of the search (lines 4894-4899); the square root by Nat.sqrt, for mpz_sqrt.)

- find_points.c:4897-4898: modelled by Ratpoints.Report.setup (deviation of Ratpoints.Report.setup: one text for the messages between the Sturm step and the 2-adic information, the lines before the divisor set-up printed with those after it, from its result: the set-up prints nothing; the divisors as a list, for the zero-terminated array divisors.)

- find_points.c:4899-4899: modelled by Ratpoints.Sieve.pointsAtInfinity (deviation of Ratpoints.Sieve.pointsAtInfinity: the flags fl as an argument, for the bits RATPOINTS_REVERSED, RATPOINTS_NO_CHECK and RATPOINTS_NO_Y of the flags of the search (args->flags); the delivery r passed and returned, for the pointer info, the local flag quit and the count total of find_points_work_1; the return when the callback stops the search (lines 5100-5103) made by the caller (Ratpoints.chooseAndSearch), which also prints the line before them under RATPOINTS_VERBOSE (Ratpoints.Report.atInfinity), the line that there are points at infinity coming with the report of the tests on the denominators (Ratpoints.Report.setup); no output under DEBUG; the square test of c_d and the decision whether there are points at infinity are made on the list that it is given, after the reversal check, when the points are delivered, where the C code computes lcfsq before the check (line 4634), which clears it in case 2 (line 4770), the only reversal after which the degree is even, and leaves it, unread, with the odd degree that cases 1 and 3 leave, and sets point_at_infty before the exits of the search (lines 4894-4899); the square root by Nat.sqrt, for mpz_sqrt.)

- find_points.c:4900-4914: modelled by Ratpoints.searchPoints (deviation of Ratpoints.searchPoints: the range of the denominators, the search intervals and the flag sturm_empty as arguments, for the fields b_low, b_high, domain and num_inter of the arguments and the local flag of find_points_work_1; the flags of the denominators as Booleans, for the bits of args->flags; the divisors of the leading coefficient with their records as lists, for the arrays args->den_info and args->divisors of the arguments; the square test lcfsq of line 4906 made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), for the bits that find_points_work_1 sets in args->flags, the intervals that it leaves in args->domain and the fields sp1_used, sp2_used and sp3_used that the part after the choice

sets (Ratpoints.chooseAndSearch); the report under RATPOINTS_VERBOSE printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under DEBUG.)

- find_points.c:4915-4918: modelled by Ratpoints.Report.setup (deviation of Ratpoints.Report.setup: one text for the messages between the Sturm step and the 2-adic information, the lines before the divisor set-up printed with those after it, from its result: the set-up prints nothing; the divisors as a list, for the zero-terminated array divisors.)

- find_points.c:4919-4920: modelled by Ratpoints.searchPoints (deviation of Ratpoints.searchPoints: the range of the denominators, the search intervals and the flag sturm_empty as arguments, for the fields b_low, b_high, domain and num_inter of the arguments and the local flag of find_points_work_1; the flags of the denominators as Booleans, for the bits of args->flags; the divisors of the leading coefficient with their records as lists, for the arrays args->den_info and args->divisors of the arguments; the square test lcfsq of line 4906 made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), for the bits that find_points_work_1 sets in args->flags, the intervals that it leaves in args->domain and the fields sp1_used, sp2_used and sp3_used that the part after the choice sets (Ratpoints.chooseAndSearch); the report under RATPOINTS_VERBOSE printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under DEBUG.)

- find_points.c:4921-4926: modelled by Ratpoints.Report.setup (deviation of Ratpoints.Report.setup: one text for the messages between the Sturm step and the 2-adic information, the lines before the divisor set-up printed with those after it, from its result: the set-up prints nothing; the divisors as a list, for the zero-terminated array divisors.)

- find_points.c:4927-4927: modelled by Ratpoints.searchPoints (deviation of Ratpoints.searchPoints: the range of the denominators, the search intervals and the flag sturm_empty as arguments, for the fields b_low, b_high, domain and num_inter of the arguments and the local flag of find_points_work_1; the flags of the denominators as Booleans, for the bits of args->flags; the divisors of the leading coefficient with their records as lists, for the arrays args->den_info and args->divisors of the arguments; the square test lcfsq of line 4906 made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), for the bits that find_points_work_1 sets in args->flags, the intervals that it leaves in

`args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4928-4941`: modelled by `Ratpoints.Report.setup` (deviation of `Ratpoints.Report.setup`: one text for the messages between the Sturm step and the 2-adic information, the lines before the divisor set-up printed with those after it, from its result: the set-up prints nothing; the divisors as a list, for the zero-terminated array `divisors`.)

- `find_points.c:4942-4943`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4946-4947`: modelled by `Ratpoints.Report.setup` (deviation of `Ratpoints.Report.setup`: one text for the messages between the Sturm step and the 2-adic information, the lines before the divisor set-up printed with those after it, from its result: the set-up prints nothing; the divisors as a list, for the zero-terminated array `divisors`.)

- `find_points.c:4948-4948`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`),

for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `find_points.c:4954-4959`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)
- `find_points.c:4960-4961`: modelled by `Ratpoints.Report.noClasses`
- `find_points.c:4962-4963`: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906 made on the polynomial searched (`Ratpoints.Setup.isSquare`), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (`Ratpoints.arraySizeAt`); the test on `point_at_infty` of the 2-adic exit recomputed, as `Ratpoints.Sieve.pointsAtInfinity` decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where `find_points_work_1` returns the count `0`; the delivery passed, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)
- `find_points.c:4965-4968`: excluded (instrumentation): the denominators of the 2-adic in-

formation under DEBUG, where the report under RATPOINTS_VERBOSE has the packing of the numerators

- find_points.c:4969-4985: modelled by Ratpoints.Report.packing (deviation of Ratpoints.Report.packing: the pattern of a class, one word in the model, for the first word of its bit array (EXT0), which repeats the pattern; a list for the array rep.)
- find_points.c:4988-4991: modelled by Ratpoints.Report.packing (deviation of Ratpoints.Report.packing: the pattern of a class, one word in the model, for the first word of its bit array (EXT0), which repeats the pattern; a list for the array rep.)
- find_points.c:4992-4995: modelled by Ratpoints.searchPoints (deviation of Ratpoints.searchPoints: the range of the denominators, the search intervals and the flag sturm_empty as arguments, for the fields b_low, b_high, domain and num_inter of the arguments and the local flag of find_points_work_1; the flags of the denominators as Booleans, for the bits of args->flags; the divisors of the leading coefficient with their records as lists, for the arrays args->den_info and args->divisors of the arguments; the square test lcfsq of line 4906 made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), for the bits that find_points_work_1 sets in args->flags, the intervals that it leaves in args->domain and the fields sp1_used, sp2_used and sp3_used that the part after the choice sets (Ratpoints.chooseAndSearch); the report under RATPOINTS_VERBOSE printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under DEBUG.)
- find_points.c:4996-5006: modelled by Ratpoints.chooseAndSearch (deviation of Ratpoints.chooseAndSearch: a function of its own for the part of find_points_work_1 from the call of sieving_info on; the shape of the run as an argument, for the fields of the arguments and the local variables of find_points_work_1 that the C code passes on (the range of the denominators, the search intervals, the flags of the denominators, the classes and the word den_bits); the exit returns the delivery with the message, nothing delivered, where sieving_info returns the prime and find_points_work_1 the count 0; the delivery passed and returned, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the numbers of the moduli used returned with it, for the fields sp1_used, sp2_used and sp3_used of the arguments, which find_points_work zeroes on entry; the report under RATPOINTS_VERBOSE printed through the print channel, that of the choice after it returns, where sieving_info prints it at its end; no failure for lack of memory, which the model cannot meet; no output under DEBUG.)
- find_points.c:5007-5009: modelled by Ratpoints.Report.noPoints
- find_points.c:5010-5012: modelled by Ratpoints.chooseAndSearch (deviation of Ratpoints.chooseAndSearch: a function of its own for the part of find_points_work_1 from the call of sieving_info on; the shape of the run as an argument, for the fields of the arguments and the local variables

of `find_points_work_1` that the C code passes on (the range of the denominators, the search intervals, the flags of the denominators, the classes and the word `den_bits`); the exit returns the delivery with the message, nothing delivered, where `sieving_info` returns the prime and `find_points_work_1` the count `0`; the delivery passed and returned, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the numbers of the moduli used returned with it, for the fields `sp1_used`, `sp2_used` and `sp3_used` of the arguments, which `find_points_work` zeroes on entry; the report under `RATPOINTS_VERBOSE` printed through the print channel, that of the choice after it returns, where `sieving_info` prints it at its end; no failure for lack of memory, which the model cannot meet; no output under `DEBUG`.)

- `find_points.c:5014-5026`: excluded (instrumentation): the moduli of the first two stages under DEBUG, where the report under RATPOINTS_VERBOSE has those of the three stages, the cost of a check and the forbidden divisors

- `find_points.c:5027-5042`: modelled by `Ratpoints.Report.moduli` (deviation of `Ratpoints.Report.moduli`: the moduli of the stages as the moduli of the sieve entries in their order, for the fields `p` of the entries of `sieve_list`; the doubles as values of Lean's `Float` (`Ratpoints.Double.ofFloat`), the product with the reference computed in its arithmetic; the numbers of the moduli of the second and the third stage integers, for the differences of the `long`s.)

- `find_points.c:5043-5048`: modelled by `Ratpoints.Report.excluded` (deviation of `Ratpoints.Report.excluded`: loops over the lists, for the loops over the zero-terminated arrays `forb_ba` and `forbidden`; the switch of the Jacobi test as a Boolean, for the bit `RATPOINTS_USE_JACOBI` of the flags.)

- `find_points.c:5049-5058`: modelled by `Ratpoints.Report.excludedValuations` (deviation of `Ratpoints.Report.excludedValuations`: natural numbers for the `long`s and for the mask, a `while` loop for each of the two loops `for`, the first with the power and the valuation in the order of its body.)

- `find_points.c:5059-5063`: modelled by `Ratpoints.Report.excluded` (deviation of `Ratpoints.Report.excluded`: loops over the lists, for the loops over the zero-terminated arrays `forb_ba` and `forbidden`; the switch of the Jacobi test as a Boolean, for the bit `RATPOINTS_USE_JACOBI` of the flags.)

- `find_points.c:5064-5066`: modelled by `Ratpoints.Report.moduli` (deviation of `Ratpoints.Report.moduli`: the moduli of the stages as the moduli of the sieve entries in their order, for the fields `p` of the entries of `sieve_list`; the doubles as values of Lean's `Float` (`Ratpoints.Double.ofFloat`), the product with the reference computed in its arithmetic; the numbers of the moduli of the second and the third stage integers, for the differences of the `long`s.)

- `find_points.c:5068-5074`: modelled by `Ratpoints.Sieve.pointsAtInfinity` (deviation of `Ratpoints.Sieve.pointsAtInfinity`: the flags `fl` as an argument, for the bits `RATPOINTS_REVERSED`, `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the flags of the search (`args->flags`); the delivery `r` passed and returned, for the pointer `info`, the local flag `quit` and the count `total` of `find_points_work_1`; the return when the callback stops the search (lines 5100-5103) made by the caller (`Ratpoints.chooseAndSearch`), which also prints the line before them under `RATPOINTS_VERBOSE` (`Ratpoints.Report.atInfinity`), the line that there are points at infinity coming with the report of the tests on the denominators (`Ratpoints.Report.setup`); no output under `DEBUG`; the square test of `c_d` and the decision whether there are points at infinity are made on the list that it is given, after the reversal check, when the points are delivered, where the C code computes `lcfsq` before the check (line 4634), which clears it in case 2 (line 4770), the only reversal after which the degree is even, and leaves it, unread,

with the odd degree that cases 1 and 3 leave, and sets `point_at_infty` before the exits of the search (lines 4894-4899); the square root by `Nat.sqrt`, for `mpz_sqrt`.)

- find_points.c:5075-5077: modelled by `Ratpoints.Report.atInfinity`
- find_points.c:5078-5099: modelled by `Ratpoints.Sieve.pointsAtInfinity` (deviation of `Ratpoints.Sieve.pointsAtInfinity`: the flags `fl` as an argument, for the bits `RATPOINTS_REVERSED`, `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the flags of the search (`args->flags`); the delivery `r` passed and returned, for the pointer `info`, the local flag `quit` and the count `total` of `find_points_work_1`; the return when the callback stops the search (lines 5100-5103) made by the caller (`Ratpoints.chooseAndSearch`), which also prints the line before them under `RATPOINTS_VERBOSE` (`Ratpoints.Report.atInfinity`), the line that there are points at infinity coming with the report of the tests on the denominators (`Ratpoints.Report.setup`); no output under `DEBUG`; the square test of `c_d` and the decision whether there are points at infinity are made on the list that it is given, after the reversal check, when the points are delivered, where the C code computes `lcfsq` before the check (line 4634), which clears it in case 2 (line 4770), the only reversal after which the degree is even, and leaves it, unread, with the odd degree that cases 1 and 3 leave, and sets `point_at_infty` before the exits of the search (lines 4894-4899); the square root by `Nat.sqrt`, for `mpz_sqrt`.)
- find_points.c:5100-5103: modelled by `Ratpoints.chooseAndSearch` (deviation of `Ratpoints.chooseAndSearch`: a function of its own for the part of `find_points_work_1` from the call of `sieving_info` on; the shape of the run as an argument, for the fields of the arguments and the local variables of `find_points_work_1` that the C code passes on (the range of the denominators, the search intervals, the flags of the denominators, the classes and the word `den_bits`); the exit returns the delivery with the message, nothing delivered, where `sieving_info` returns the prime and `find_points_work_1` the count `0`; the delivery passed and returned, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the numbers of the moduli used returned with it, for the fields `sp1_used`, `sp2_used` and `sp3_used` of the arguments, which `find_points_work` zeroes on entry; the report under `RATPOINTS_VERBOSE` printed through the print channel, that of the choice after it returns, where `sieving_info` prints it at its end; no failure for lack of memory, which the model cannot meet; no output under `DEBUG`.)
- find_points.c:5104-5104: modelled by `Ratpoints.Report.sieving` (deviation of `Ratpoints.Report.sieving`: one text for the line feed after the points at infinity and the line before the sieve, between which the C code returns when the Sturm step has left no interval.)
- find_points.c:5105-5106: modelled by `Ratpoints.chooseAndSearch` (deviation of `Ratpoints.chooseAndSearch`: a function of its own for the part of `find_points_work_1` from the call of `sieving_info` on; the shape of the run as an argument, for the fields of the arguments and the local variables of `find_points_work_1` that the C code passes on (the range of the denominators, the search intervals, the flags of the denominators, the classes and the word `den_bits`); the exit returns the delivery with the message, nothing delivered, where `sieving_info` returns the prime and `find_points_work_1` the count `0`; the delivery passed and returned, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the numbers of the moduli used returned with it, for the fields `sp1_used`, `sp2_used` and `sp3_used` of the arguments, which `find_points_work` zeroes on entry; the report under `RATPOINTS_VERBOSE` printed through the print channel, that of the choice after it returns, where `sieving_info` prints it at its end; no failure for lack of memory, which the model cannot meet; no output under `DEBUG`.)

- find_points.c:5108-5110: excluded (instrumentation): the message before the sieve under DEBUG
- find_points.c:5111-5113: modelled by Ratpoints.Report.sieving (deviation of Ratpoints.Report.sieving: one text for the line feed after the points at infinity and the line before the sieve, between which the C code returns when the Sturm step has left no interval.)
- find_points.c:5116-5116: modelled by Ratpoints.sieving (deviation of Ratpoints.sieving: a function of its own for the block of the sieving in find_points_work_1; the range of the denominators and the search intervals as arguments, for the fields b_low, b_high, domain and num_inter of the arguments that run_setup and the run read; the flags of the denominators as Booleans and an option, for the bits of args->flags and the arrays args->den_info and args->divisors that run_setup reads; the array of the row offsets made with zeros, for the array on the stack, whose entries the C code leaves unset until class_offsets writes them, with one offset in each row for each of the sp3_max sieve entries of the run, the reserve of the correction included; the delivery passed and returned, for the pointers info and wk.quit and the count total, and the ends of the second and the third stage of the last block returned with it, for the fields sp2 and sp3 of the arguments that the run writes; no sieving thread made (worker_init, worker_clear), whose fields the components that the run calls take as arguments, and no pool of threads (excluded with the threads); no failure for lack of memory, which the model cannot meet.)
- find_points.c:5119-5119: modelled by Ratpoints.sieving (deviation of Ratpoints.sieving: a function of its own for the block of the sieving in find_points_work_1; the range of the denominators and the search intervals as arguments, for the fields b_low, b_high, domain and num_inter of the arguments that run_setup and the run read; the flags of the denominators as Booleans and an option, for the bits of args->flags and the arrays args->den_info and args->divisors that run_setup reads; the array of the row offsets made with zeros, for the array on the stack, whose entries the C code leaves unset until class_offsets writes them, with one offset in each row for each of the sp3_max sieve entries of the run, the reserve of the correction included; the delivery passed and returned, for the pointers info and wk.quit and the count total, and the ends of the second and the third stage of the last block returned with it, for the fields sp2 and sp3 of the arguments that the run writes; no sieving thread made (worker_init, worker_clear), whose fields the components that the run calls take as arguments, and no pool of threads (excluded with the threads); no failure for lack of memory, which the model cannot meet.)
- find_points.c:5120-5120: modelled by Ratpoints.Sieve.sieveRun (deviation of Ratpoints.Sieve.sieveRun: the blocks on the calling thread alone, without the pool of threads, which sifts them apart and delivers them in the same order, the decision of each block depending only on the folds of the blocks at least Ratpoints.adaptLag + 1 before it (excluded with the threads); the ends of the last block returned, for the fields sp2 and sp3 of the arguments that line 5163 writes; the store of the tables of rows passed from one block to the next, for the arrays sieve of the entries of the arguments, its last state not returned, as nothing reads the tables of rows after the run; the delivery r passed and returned, for the pointers info and wk.quit and the count total; natural numbers for the longs.)
- find_points.c:5122-5125: excluded (instrumentation): the message under DEBUG before the allocation of the survivors
- find_points.c:5127-5128: excluded (memory): worker_init, which allocates the survivors

array of the worker of the calling thread and sets up its state, and fails only for lack of memory; the model passes what the sieve needs as arguments

- find_points.c:5129-5129: excluded (threads): the worker of the calling thread without a pool: the calling thread sieves; the model has no threads

- find_points.c:5132-5132: modelled by `Ratpoints.sieving` (deviation of `Ratpoints.sieving`: a function of its own for the block of the sieving in `find_points_work_1`; the range of the denominators and the search intervals as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments that `run_setup` and the run read; the flags of the denominators as Booleans and an option, for the bits of `args->flags` and the arrays `args->den_info` and `args->divisors` that `run_setup` reads; the array of the row offsets made with zeros, for the array on the stack, whose entries the C code leaves unset until `class_offsets` writes them, with one offset in each row for each of the `sp3_max` sieve entries of the run, the reserve of the correction included; the delivery passed and returned, for the pointers `info` and `wk.quit` and the count `total`, and the ends of the second and the third stage of the last block returned with it, for the fields `sp2` and `sp3` of the arguments that the run writes; no sieving thread made (`worker_init`, `worker_clear`), whose fields the components that the run calls take as arguments, and no pool of threads (excluded with the threads); no failure for lack of memory, which the model cannot meet.)

- find_points.c:5133-5136: excluded (instrumentation): the end of the message under DEBUG about the allocation of the survivors

- find_points.c:5137-5139: modelled by `Ratpoints.sieving` (deviation of `Ratpoints.sieving`: a function of its own for the block of the sieving in `find_points_work_1`; the range of the denominators and the search intervals as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments that `run_setup` and the run read; the flags of the denominators as Booleans and an option, for the bits of `args->flags` and the arrays `args->den_info` and `args->divisors` that `run_setup` reads; the array of the row offsets made with zeros, for the array on the stack, whose entries the C code leaves unset until `class_offsets` writes them, with one offset in each row for each of the `sp3_max` sieve entries of the run, the reserve of the correction included; the delivery passed and returned, for the pointers `info` and `wk.quit` and the count `total`, and the ends of the second and the third stage of the last block returned with it, for the fields `sp2` and `sp3` of the arguments that the run writes; no sieving thread made (`worker_init`, `worker_clear`), whose fields the components that the run calls take as arguments, and no pool of threads (excluded with the threads); no failure for lack of memory, which the model cannot meet.)

- find_points.c:5140-5151: excluded (threads): the pool of threads, when asked for, with a survivors array for each thread, and the threaded run with its failure; the model has no threads, which deliver the blocks in their order

- find_points.c:5152-5160: modelled by `Ratpoints.Sieve.sieveRun` (deviation of `Ratpoints.Sieve.sieveRun`: the blocks on the calling thread alone, without the pool of threads, which sifts them apart and delivers them in the same order, the decision of each block depending only on the folds of the blocks at least `Ratpoints.adaptLag + 1` before it (excluded with the threads); the ends of the last block returned, for the fields `sp2` and `sp3` of the arguments that line 5163 writes; the store of the tables of rows passed from one block to the next, for the arrays `sieve` of the entries of the arguments, its last state not returned, as nothing reads the tables of rows after the run; the delivery `r` passed and returned, for the pointers `info` and `wk.quit` and the count `total`; natural numbers for the `long`s.)

- find_points.c:5161-5161: excluded (threads): the end of the block of the pool of threads
- find_points.c:5163-5163: modelled by `Ratpoints.Sieve.sieveRun` (deviation of `Ratpoints.Sieve.sieveRun`: the blocks on the calling thread alone, without the pool of threads, which sifts them apart and delivers them in the same order, the decision of each block depending only on the folds of the blocks at least `Ratpoints.adaptLag + 1` before it (excluded with the threads); the ends of the last block returned, for the fields `sp2` and `sp3` of the arguments that line 5163 writes; the store of the tables of rows passed from one block to the next, for the arrays `sieve` of the entries of the arguments, its last state not returned, as nothing reads the tables of rows after the run; the delivery `r` passed and returned, for the pointers `info` and `wk.quit` and the count `total`; natural numbers for the `long`s.)
- find_points.c:5164-5164: excluded (memory): the release of the run (run_clear)
- find_points.c:5166-5166: excluded (memory): the release of the worker of the calling thread (worker_clear)
- find_points.c:5167-5167: modelled by `Ratpoints.sieving` (deviation of `Ratpoints.sieving`: a function of its own for the block of the sieving in `find_points_work_1`; the range of the denominators and the search intervals as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments that `run_setup` and the run read; the flags of the denominators as Booleans and an option, for the bits of `args->flags` and the arrays `args->den_info` and `args->divisors` that `run_setup` reads; the array of the row offsets made with zeros, for the array on the stack, whose entries the C code leaves unset until `class_offsets` writes them, with one offset in each row for each of the `sp3_max` sieve entries of the run, the reserve of the correction included; the delivery passed and returned, for the pointers `info` and `wk.quit` and the count `total`, and the ends of the second and the third stage of the last block returned with it, for the fields `sp2` and `sp3` of the arguments that the run writes; no sieving thread made (`worker_init`, `worker_clear`), whose fields the components that the run calls take as arguments, and no pool of threads (excluded with the threads); no failure for lack of memory, which the model cannot meet.)
- find_points.c:5169-5208: excluded (instrumentation): the report on the shape of the run under RP_PRIME_STATS and RP_PHASE_TIMING, which prints on stderr what run_shape predicted against what the run did
- find_points.c:5210-5213: modelled by `Ratpoints.chooseAndSearch` (deviation of `Ratpoints.chooseAndSearch`: a function of its own for the part of `find_points_work_1` from the call of `sieving_info` on; the shape of the run as an argument, for the fields of the arguments and the local variables of `find_points_work_1` that the C code passes on (the range of the denominators, the search intervals, the flags of the denominators, the classes and the word `den_bits`); the exit returns the delivery with the message, nothing delivered, where `sieving_info` returns the prime and `find_points_work_1` the count `0`; the delivery passed and returned, for the pointer `info`, the local flag `quit` of `find_points_work_1` and the count `total`, and the numbers of the moduli used returned with it, for the fields `sp1_used`, `sp2_used` and `sp3_used` of the arguments, which `find_points_work` zeroes on entry; the report under `RATPOINTS_VERBOSE` printed through the print channel, that of the choice after it returns, where `sieving_info` prints it at its end; no failure for lack of memory, which the model cannot meet; no output under `DEBUG`.)
- find_points.c:5215-5217: excluded (instrumentation): the message under DEBUG at the end of find_points_work_1

- find_points.c:5219-5220: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- find_points.c:5226-5229: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree

0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `find_points.c:5232-5232`: excluded (memory): find_points_init, which allocates the work array and the buffers of the search; its refusal of a degree outside 0, ..., RATPOINTS_MAX_DEGREE never takes effect, since find_points_work refuses the same degrees first

- `find_points.c:5235-5235`: modelled by `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height

(Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- find_points.c:5238-5238: excluded (memory): find_points_clear, which releases them
- find_points.c:5241-5242: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the

working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- find_points.c:5250-5338: modelled by Ratpoints.Sieve.checkPoint (deviation of Ratpoints.Sieve.checkPoint: the products w as a thunk, which the sieve of one denominator makes for each denominator (Ratpoints.Sieve.sift) and which is forced when the first numerator of b reaches the check, for the array work of the sieving thread, which worker_init gives it, and its flag compute_bc, which sift sets and _ratpoints_check_point clears when it computes the products; the degree d and the flags fl as arguments, handed down from Ratpoints.Sieve.sieveRun through the components that call the check, for the degree and the bits RATPOINTS_REVERSED, RATPOINTS_NO_CHECK and RATPOINTS_NO_Y of the flags of the search, which _ratpoints_check_point reads through its worker (args->degree, args->flags); the delivery r passed and returned, for the pointers info and wk->quit and the count total that _ratpoints_check_point keeps from zero and returns, and the three branches of the map of the reversal at each call one definition (Ratpoints.Delivery.callMapped); the square root and the test of the remainder by Nat.sqrt on v, for mpz_sqrtrem; integers for the longs a and b (the negations of the map stay longs by Ratpoints.fitsLong_neg); no measurements of the time of the check and of its products (RP_CHECK, RP_BC_TIC, RP_BC_TOC under RP_PHASE_TIMING), and no output under DEBUG.)

sift.c

- sift.c:30-30: excluded (other): the include of the private header; the model has no headers
- sift.c:55-176: excluded (instrumentation): RP_PHASE_TIMING and RP_PHASE_COUNTS, development instrumentation that times the parts of _ratpoints_sift0 with rdtsc, counts what survives each and prints one line on stderr at exit, with the arm of the production build, where the timers RP_TIC and RP_TOC are empty and RP_CHECK is the call it wraps
- sift.c:194-201: excluded (instrumentation): RP_STOP_AFTER, development instrumentation that cuts the pipeline short after a step so that its cost can be measured, 0 in the production build, with the exact check of its level 4 replaced by a count
- sift.c:202-205: modelled by Ratpoints.Sieve.checkPoint (deviation of Ratpoints.Sieve.checkPoint: the products w as a thunk, which the sieve of one denominator makes for each denominator (Ratpoints.Sieve.sift) and which is forced when the first numerator of b reaches the check, for the array work of the sieving thread, which worker_init gives it, and its flag compute_bc, which sift sets and _ratpoints_check_point clears when it computes the products; the degree d and the flags fl as arguments, handed down from Ratpoints.Sieve.sieveRun through

the components that call the check, for the degree and the bits `RATPOINTS_REVERSED`, `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the flags of the search, which `_ratpoints_check_point` reads through its worker (`args->degree`, `args->flags`); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_check_point` keeps from zero and returns, and the three branches of the map of the reversal at each call one definition (`Ratpoints.Delivery.callMapped`); the square root and the test of the remainder by `Nat.sqrt` on `v`, for `mpz_sqrtrem`; integers for the `long`s `a` and `b` (the negations of the map stay `long`s by `Ratpoints.fitsLong_neg`); no measurements of the time of the check and of its products (`RP_CHECK`, `RP_BC_TIC`, `RP_BC_TOC` under `RP_PHASE_TIMING`), and no output under `DEBUG`.)

- `sift.c:207-213`: excluded (instrumentation): the sink into which the levels of RP_STOP_AFTER accumulate the work they leave out, printed on stderr at exit
- `sift.c:225-238`: excluded (instrumentation): RP_MOD_CHOICE, a development switch that chooses the reduction of the word numbers of the chunked first stage at run time, from the environment variable RP_MOD_MUL, so that the two reductions can be timed in one binary; in the production build RP_USE_MOD_MUL is the constant 1, the reduction by the reciprocal wherever it is exact
- `sift.c:243-247`: modelled by `Ratpoints.Sieve.modMul` (deviation of `Ratpoints.Sieve.modMul`: integers for the `long`s `a` and the result, and natural numbers for the `long p` and the `unsigned long` reciprocal `m`; `|a|` a natural number, for the `long` that `RP_MULMOD` converts to an `unsigned long`.)
- `sift.c:262-264`: modelled by `Ratpoints.Sieve.walkWord` (deviation of `Ratpoints.Sieve.walkWord`: the lowest set bit by the loop of `RP_CTZL` for a compiler without the builtin (`Ratpoints.Sieve.ctz`); one definition for the body at the three places of the extraction, the first word of a bit array, the others, and a word of the arm of the second stage for one word at a time; natural numbers and integers for the `long`s; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, to which `RP_CHECK_POINT` adds the count of the check (lines 1168, 1240, 1250, 1279, 1287); the return after a stop (lines 1169, 1241, 1251, 1280, 1288) a return from the walk, which its callers pass on, testing the flag after it (`Ratpoints.Sieve.siftBitArrayByWords`, `Ratpoints.Sieve.siftBitArrayWhole`, then `Ratpoints.Sieve.scanPiece` after the bit array), where the C code returns from `_ratpoints_sift0`; the counters passed and returned, for the field `n_bits` of the sieving thread (lines 1166, 1233, 1271); no counts under `RP_PHASE_COUNTS`; no output under `DEBUG`.)
- `sift.c:291-301`: modelled by `Ratpoints.Sieve.stage3` (deviation of `Ratpoints.Sieve.stage3`: a list for the array `csp` and its length `n`; the residue computed by `Ratpoints.Sieve.CheckSpec.index`; a Boolean for the `int`.)
- `sift.c:302-308`: modelled by `Ratpoints.Sieve.CheckSpec.index` (deviation of `Ratpoints.Sieve.CheckSpec.index`: a function of its own for the lines of the loop of `stage3` that compute the residue; integers for the `long`s, `Int.tmod` for C's `%`, which truncates, and the value made a natural number, for the `long` that `RP_MULMOD` converts to an `unsigned long`, and for the `long` residue, both non-negative; the obligations that the `long`s `a binv + bias` and `am binv` stay in range left to lemmas: for `|a| ≤ H` and `binv < p`, `a binv + p H` lies between `H` and `2 p H - H` (`Ratpoints.Sieve.le_mul_add_natCast_mul`, `Ratpoints.Sieve.mul_add_natCast_mul_le_two_mul_mul_sub`), below `2 ^ 32` with the bias, and `am binv < p ^ 2` (`Ratpoints.Sieve.toNat_emod_mul_lt_sq`).)

- sift.c:309-312: modelled by `Ratpoints.Sieve.stage3` (deviation of `Ratpoints.Sieve.stage3`: a list for the array `csp` and its length `n`; the residue computed by `Ratpoints.Sieve.CheckSpec.index`; a Boolean for the `int`.)
- sift.c:318-327: modelled by `Ratpoints.Sieve.relprime` (deviation of `Ratpoints.Sieve.relprime`: `Int64` for `long`, `toUInt64` for the cast to `unsigned long`, and the `toInt64` of the count for the `long` that `RP_CTZL` returns; the negation of a negative `m` as a conditional expression, for the statement `if(m < 0)`; for `m = -2 ^ 63`, whose negation overflows in the C code, the model wraps; the contract excludes that value, which the search never passes, its numerators being at most `H < 2 ^ 63` in absolute value; `m &&& 1 == 0` for `!(m & 1)`; the lowest set bit by the loop of `RP_CTZL` for a compiler without the builtin (`Ratpoints.Sieve.ctz`), where gcc and clang take `__builtin_ctzl`; the builtin is undefined on `0` and the loop does not end there, which the contract excludes, with `n ≥ 1` and `m ≠ 0` at the calls; the loop of the binary algorithm a function of its own (`Ratpoints.Sieve.relprimeOdd`); a Boolean for the `int`.)
- sift.c:328-342: modelled by `Ratpoints.Sieve.relprimeOdd` (deviation of `Ratpoints.Sieve.relprimeOdd`: `Int64` for `long`, `toUInt64` for the cast to `unsigned long`, and the `toInt64` of the count for the `long` that `RP_CTZL` returns; `63` for `LONG_LENGTH - 1`; `Int64` in two's complement, the representation of the negative `long`s on which the and `d &&& msk` and the exclusive or `d ^^^ msk` of the step act, and `>>>` on `Int64` the arithmetic shift, for the `>>` of a negative `long`, both of which the C standard leaves to the implementation (the representation before C23) and gcc defines so; the lowest set bit by the loop of `RP_CTZL` for a compiler without the builtin (`Ratpoints.Sieve.ctz`), where gcc and clang take `__builtin_ctzl`; the loop a function of its own, for the end of the function `relprime`.)
- sift.c:343-343: modelled by `Ratpoints.Sieve.relprime` (deviation of `Ratpoints.Sieve.relprime`: `Int64` for `long`, `toUInt64` for the cast to `unsigned long`, and the `toInt64` of the count for the `long` that `RP_CTZL` returns; the negation of a negative `m` as a conditional expression, for the statement `if(m < 0)`; for `m = -2 ^ 63`, whose negation overflows in the C code, the model wraps; the contract excludes that value, which the search never passes, its numerators being at most `H < 2 ^ 63` in absolute value; `m &&& 1 == 0` for `!(m & 1)`; the lowest set bit by the loop of `RP_CTZL` for a compiler without the builtin (`Ratpoints.Sieve.ctz`), where gcc and clang take `__builtin_ctzl`; the builtin is undefined on `0` and the loop does not end there, which the contract excludes, with `n ≥ 1` and `m ≠ 0` at the calls; the loop of the binary algorithm a function of its own (`Ratpoints.Sieve.relprimeOdd`); a Boolean for the `int`.)
- sift.c:349-367: excluded (instrumentation): RP_MOD_COUNTS, development instrumentation that counts how often the helper mod divides and prints the counts on stderr at exit, with the arm of the production build, where the counters RP_MOD_TICK and RP_MOD_DIV are empty
- sift.c:370-389: modelled by `Ratpoints.Sieve.modLong` (deviation of `Ratpoints.Sieve.modLong`: one definition for the two copies of the function, in `find_points.c` and in `sift.c`, which are the same but for the hooks of the counts of the reductions in `sift.c` (`RP_MOD_TICK`, `RP_MOD_DIV`), empty unless `RP_MOD_COUNTS` is set; integers for the `long`s, with the obligations that the values it computes stay in the range of a `long` stated as lemmas: for `0 ≤ b < 2 ^ 59`, `16 b` (`Ratpoints.fitsLong_sixteen_mul`), its negation (`Ratpoints.fitsLong_neg`), and `a + 16 b` for `-16 b ≤ a < 0` (`Ratpoints.fitsLong_add_sixteen_mul`), its other values lying between `-b` and `16 b`; `Int.tmod` for C's division of `long`s, which truncates; the multiples `16 b`, `8 b`, `4 b` and `2 b` as products, for the shifts of `b1`.)

- sift.c:405-413: modelled by Ratpoints.Sieve.fillChecks (deviation of Ratpoints.Sieve.fillChecks: a list, made by a map over the entries of the third stage and their reciprocals (List.zipWith), for the loop over the positions sp2, …, sp3 - 1 of sieve_list and magics that fills the array checks of the sieving thread; no bound RATPOINTS_NUM_PRIMES on its length (stage3_list and the arrays of the pool, find_points.c:339, find_points.c:4372), which ratpoints never reaches since each entry of sieve_list owns its own prime (find_points.c:329-331), where the model takes any entries, which may repeat a prime; the thunk that the sieve of one denominator makes of it (Ratpoints.Sieve.sift), for the flag stage3_filled of the sieving thread, which sift clears and fill_checks sets.)

- sift.c:414-428: modelled by Ratpoints.Sieve.checkOf (deviation of Ratpoints.Sieve.checkOf: the entry of one prime, for one pass of the loop of fill_checks (Ratpoints.Sieve.fillChecks), with the reciprocal magics[n] as the argument mg; the fields of the prime read from its table, the single factor of its sieve entry, where the C code reads them from the sieve entry of the prime, which has them itself; the exact comparison 2 p H < stage3Limit of natural numbers, for the comparison of the double product p · 2.0 · H with the double RP_STAGE3_LIMIT, which has the same outcome for every H < 2 ^ 63, the product being exact below 2 ^ 53 and rounded monotonically above, beyond the bound; natural numbers for the longs, the product p H computed only when it is below 2 ^ 31; the inverse 0 for an entry of another shape, for which the C code has no case.)

- sift.c:429-431: modelled by Ratpoints.Sieve.fillChecks (deviation of Ratpoints.Sieve.fillChecks: a list, made by a map over the entries of the third stage and their reciprocals (List.zipWith), for the loop over the positions sp2, …, sp3 - 1 of sieve_list and magics that fills the array checks of the sieving thread; no bound RATPOINTS_NUM_PRIMES on its length (stage3_list and the arrays of the pool, find_points.c:339, find_points.c:4372), which ratpoints never reaches since each entry of sieve_list owns its own prime (find_points.c:329-331), where the model takes any entries, which may repeat a prime; the thunk that the sieve of one denominator makes of it (Ratpoints.Sieve.sift), for the flag stage3_filled of the sieving thread, which sift clears and fill_checks sets.)

- sift.c:438-446: modelled by Ratpoints.Sieve.accepted (deviation of Ratpoints.Sieve.accepted: the integer a and the natural number b of the loops of the model converted to the 64-bit words that they are in the C code (Int.toInt64, Nat.toInt64), which is exact for |a| < 2 ^ 63 and b < 2 ^ 63, as the numerators and the denominators of a search with a height bound below 2 ^ 63 are (Ratpoints.findPoints); the thunk checks, which the sieve of one denominator makes for each denominator (Ratpoints.Sieve.sift), for the array of the sieving thread and its flag stage3_filled, which accepted tests before it fills the array when the stage has primes (n ≠ 0); the counters passed and returned, for the fields n_coprime and n_checks of the sieving thread; a Boolean for the int.)

- sift.c:452-465: modelled by Ratpoints.Sieve.tailLeg (deviation of Ratpoints.Sieve.tailLeg: the registers a vector of 8 bit arrays, made of zeros, for the array reg[8]; the pointer surv as the array surv and the index base of the first bit array of the tail, past the whole chunks; indices into the rows for the pointers; the size of the local array of the sieve specs for the number sp1 of the moduli; no pragmas that make the compiler unroll the loops over the registers (RP_UNROLL_REGS, sift.c:466-472, excluded); natural numbers for the longs w and off.)

- sift.c:466-472: excluded (other): RP_UNROLL_REGS, the pragma GCC unroll 8 with which tail_leg makes gcc and clang unroll its loops over the registers before they decide

what to keep in registers, empty on other compilers or when given empty; it carries no meaning

- sift.c:473-496: modelled by Ratpoints.Sieve.tailLeg (deviation of Ratpoints.Sieve.tailLeg: the registers a vector of 8 bit arrays, made of zeros, for the array reg[8]; the pointer surv as the array surv and the index base of the first bit array of the tail, past the whole chunks; indices into the rows for the pointers; the size of the local array of the sieve specs for the number sp1 of the moduli; no pragmas that make the compiler unroll the loops over the registers (RP_UNROLL_REGS, sift.c:466-472, excluded); natural numbers for the longs w and off.)

- sift.c:498-531: modelled by Ratpoints.Sieve.siftPiece (deviation of Ratpoints.Sieve.siftPiece: the arm of the first stage chosen by a test on the constant Ratpoints.chunk, for the conditional compilation #if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16) (sift.c:452, sift.c:575, sift.c:943), which compiles one arm only, and tail_leg with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words malloc leaves uninitialized; the piece by its first word number w0 and its number n of bit arrays, for w_low and w_high, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list specs and the end sp1 of the first stage, for the sieve specs sieves and the number sp1 that the sieving thread keeps, the reciprocals of their moduli as the list mgs, for the array magics of the arguments, and the thunk checks, for the table checks of the third stage and its length nchecks = sp3 - sp2; the class cl as a value, for the pointer cls; integers for the longs of the test small, which stay longs, and a Boolean for its int; the counters cnt passed and returned, for the field n_words of the sieving thread, which counts the n Ratpoints.rbaPack words of the piece (line 548), the product of UInt64s as the C code multiplies the unsigned long by RBA_PACK; the thunk w of the products of the check as an argument, where _ratpoints_check_point computes the products when the first survivor of b reaches it; the delivery r passed and returned, for the pointers info and wk->quit and the count total that _ratpoints_sift0 keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (Ratpoints.Sieve.scanPiece); natural numbers and integers for the longs; no measurements of the phases (RP_TIC, RP_TOC, RP_PHASE_TIMING) and no stop after one of them (RP_STOP_AFTER).)

- sift.c:533-544: excluded (instrumentation): the output under DEBUG at the start of _ratpoints_sift0: the denominator, the number of bit arrays, the 2-adic pattern and the masks

- sift.c:548-548: modelled by Ratpoints.Sieve.siftPiece (deviation of Ratpoints.Sieve.siftPiece: the arm of the first stage chosen by a test on the constant Ratpoints.chunk, for the conditional compilation #if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16) (sift.c:452, sift.c:575, sift.c:943), which compiles one arm only, and tail_leg with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words malloc leaves uninitialized; the piece by its first word number w0 and its number n of bit arrays, for w_low and w_high, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list specs and the end sp1 of the first stage, for the sieve specs sieves and the number sp1 that the sieving thread keeps, the reciprocals of their moduli as the list mgs, for the array magics of the arguments, and the thunk checks, for the table checks of the third stage and its length nchecks = sp3 - sp2;

the class `cl` as a value, for the pointer `cls`; integers for the `long`s of the test `small`, which stay `long`s, and a Boolean for its `int`; the counters `cnt` passed and returned, for the field `n_words` of the sieving thread, which counts the `n Ratpoints.rbaPack` words of the piece (line 548), the product of `UInt64`s as the C code multiplies the `unsigned long` by `RBA_PACK`; the thunk `w` of the products of the check as an argument, where `_ratpoints_check_point` computes the products when the first survivor of `b` reaches it; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_sift0` keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (`Ratpoints.Sieve.scanPiece`); natural numbers and integers for the `long`s; no measurements of the phases (`RP_TIC`, `RP_TOC`, `RP_PHASE_TIMING`) and no stop after one of them (`RP_STOP_AFTER`).)

- `sift.c:550-573`: excluded (instrumentation): the counts of the calls, of the bit arrays and of the bits under RP_PHASE_TIMING and RP_PHASE_COUNTS, the start of the timing of the first stage (RP_TIC) and the output of the cut points under DEBUG, before the first stage

- `sift.c:575-575`: modelled by `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRowsChunked`: no moduli tested by the size of the local array of the sieve specs, for `sp1 == 0`; the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`); the end of the whole chunks the index `full`, for the word number `w_high_full`, which is also the index of the first bit array of the tail, for the pointer `surv` after the chunks; natural numbers and integers for the `long`s.)

- `sift.c:596-606`: modelled by `Ratpoints.Sieve.chunkStarts` (deviation of `Ratpoints.Sieve.chunkStarts`: a local array of the sieve specs, in which the starts are set, for the sieve specs `sieves` of the sieving thread, whose field `start` the C code writes and nothing reads after the first stage; the moduli of the stage with their rows and their reciprocals as two lists, for the first `sp1` sieve specs and the array `magics` from position `0` on; the reduction by the reciprocal under `small` alone, for `small && RP_USE_MOD_MUL`, with `RP_USE_MOD_MUL` the constant 1 of the production build (`sift.c:236-238`, excluded with the switch of the development builds); the start an index into the row, for the pointer `ptr` plus the remainder; integers for the `long`s, the sum `a = w_low + offset` staying a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_add`); a Boolean for the `int small`.)

- `sift.c:608-620`: modelled by `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRowsChunked`: no moduli tested by the size of the local array of the sieve specs, for `sp1 == 0`; the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`); the end of the whole chunks the index `full`, for the word number `w_high_full`, which is also the index of the first bit array of the tail, for the pointer `surv` after the chunks; natural numbers and integers for the `long`s.)

- `sift.c:624-624`: modelled by `Ratpoints.Sieve.andChunks` (deviation of `Ratpoints.Sieve.andChunks`: the index `c` of the first bit array of the chunk in `surv`, from `0`, for the word number `w_low_new`, from `w_low`, and the pointer `surv` into the survivors, which the loop steps on together, and

full the index of the end of the whole chunks, for the word number w_high_full; the stores a loop over the registers, for the code that the build unrolls under #if (RATPOINTS_CHUNK >= k), for k up to 16; the local array of the sieve specs, for the sieve specs of the sieving thread.)

- sift.c:625-681: modelled by Ratpoints.Sieve.chunkSieve (deviation of Ratpoints.Sieve.chunkSieve: the registers reg0, …, reg15 a vector of chunk bit arrays, made of zeros and then loaded, and the loads and the ands of the registers loops over its bit arrays, for the variables that the build declares and the code that it unrolls under #if (RATPOINTS_CHUNK >= k), for k up to 16; the pointers siv0 and siv1 after the loads and the ands as the start plus chunk, for the pointers that each load and each and steps on by one bit array; indices into the rows for the pointers start and end; the local array of the sieve specs, for the sieve specs of the sieving thread, and its size for the number sp1 of the moduli of the stage; no output under DEBUG.)
- sift.c:683-683: modelled by Ratpoints.Sieve.wrapStart (deviation of Ratpoints.Sieve.wrapStart: one definition for the loop at its two places, after the first modulus of a chunk and after each further one; indices into the row for the pointers.)
- sift.c:684-804: modelled by Ratpoints.Sieve.chunkSieve (deviation of Ratpoints.Sieve.chunkSieve: the registers reg0, …, reg15 a vector of chunk bit arrays, made of zeros and then loaded, and the loads and the ands of the registers loops over its bit arrays, for the variables that the build declares and the code that it unrolls under #if (RATPOINTS_CHUNK >= k), for k up to 16; the pointers siv0 and siv1 after the loads and the ands as the start plus chunk, for the pointers that each load and each and steps on by one bit array; indices into the rows for the pointers start and end; the local array of the sieve specs, for the sieve specs of the sieving thread, and its size for the number sp1 of the moduli of the stage; no output under DEBUG.)
- sift.c:805-805: modelled by Ratpoints.Sieve.wrapStart (deviation of Ratpoints.Sieve.wrapStart: one definition for the loop at its two places, after the first modulus of a chunk and after each further one; indices into the row for the pointers.)
- sift.c:806-863: modelled by Ratpoints.Sieve.chunkSieve (deviation of Ratpoints.Sieve.chunkSieve: the registers reg0, …, reg15 a vector of chunk bit arrays, made of zeros and then loaded, and the loads and the ands of the registers loops over its bit arrays, for the variables that the build declares and the code that it unrolls under #if (RATPOINTS_CHUNK >= k), for k up to 16; the pointers siv0 and siv1 after the loads and the ands as the start plus chunk, for the pointers that each load and each and steps on by one bit array; indices into the rows for the pointers start and end; the local array of the sieve specs, for the sieve specs of the sieving thread, and its size for the number sp1 of the moduli of the stage; no output under DEBUG.)
- sift.c:864-913: modelled by Ratpoints.Sieve.andChunks (deviation of Ratpoints.Sieve.andChunks: the index c of the first bit array of the chunk in surv, from 0, for the word number w_low_new, from w_low, and the pointer surv into the survivors, which the loop steps on together, and full the index of the end of the whole chunks, for the word number w_high_full; the stores a loop over the registers, for the code that the build unrolls under #if (RATPOINTS_CHUNK >= k), for k up to 16; the local array of the sieve specs, for the sieve specs of the sieving thread.)

- sift.c:915-940: modelled by Ratpoints.Sieve.tailLegs (deviation of Ratpoints.Sieve.tailLegs: the conditions on the chunk tested with those on t, for the conditional compilation #if (RATPOINTS_CHUNK > 8), > 4 and > 2; no check at compile time that the chunk is at most 16, the #error of the C code, which its place inside the conditional compilation of the chunked first stage for a chunk up to 16 makes dead code: a chunk above 16 compiles the first stage one row at a time, and the model chooses the arm by the same test (Ratpoints.Sieve.siftPiece, Ratpoints.Sieve.andRows); natural numbers for the longs.)
- sift.c:941-941: modelled by Ratpoints.Sieve.andRowsChunked (deviation of Ratpoints.Sieve.andRowsChunked: no moduli tested by the size of the local array of the sieve specs, for sp1 == 0; the bit arrays set to the class word by a loop that counts up, bit array n - 1 - i at the step i, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector RBA(packed) that get_2adic_info makes once per class (find_points.c:1098) and _ratpoints_sift0 copies into each bit array (bits64, sift.c:519); the end of the whole chunks the index full, for the word number w_high_full, which is also the index of the first bit array of the tail, for the pointer surv after the chunks; natural numbers and integers for the longs.)
- sift.c:943-1022: modelled by Ratpoints.Sieve.andRows (deviation of Ratpoints.Sieve.andRows: the bit arrays set to the class word by a loop that counts up, bit array n - 1 - i at the step i, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector RBA(packed) that get_2adic_info makes once per class (find_points.c:1098) and _ratpoints_sift0 copies into each bit array (bits64, sift.c:519, sift.c:954); the moduli of the stage with their rows and their reciprocals as two lists, for the first sp1 sieve specs and the array magics from position 0 on; the value -w_low - offset written out in both reductions, for the long a, which stays a long where the helper mod reduces it outside small (Ratpoints.Sieve.fitsLong_neg_sub); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the longs; a Boolean for the int small; no output under DEBUG.)
- sift.c:1024-1024: excluded (instrumentation): the end of the timing of the first stage (RP_TOC), empty in the production build
- sift.c:1030-1041: modelled by Ratpoints.Sieve.siftPiece (deviation of Ratpoints.Sieve.siftPiece: the arm of the first stage chosen by a test on the constant Ratpoints.chunk, for the conditional compilation #if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16) (sift.c:452, sift.c:575, sift.c:943), which compiles one arm only, and tail_leg with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words malloc leaves uninitialized; the piece by its first word number w0 and its number n of bit arrays, for w_low and w_high, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list specs and the end sp1 of the first stage, for the sieve specs sieves and the number sp1 that the sieving thread keeps, the reciprocals of their moduli as the list mgs, for the array magics of the arguments, and the thunk checks, for the table checks of the third stage and its length nchecks = sp3 - sp2; the class cl as a value, for the pointer cls; integers for the longs of the test small, which stay longs, and a Boolean for its int; the counters cnt passed and returned, for the field n_words of the sieving thread, which counts the n Ratpoints.rbaPack words of the piece (line 548), the product of UInt64s as the C code multiplies the unsigned long by RBA_PACK; the thunk w of the products of the check as an argument, where _ratpoints_check_point

computes the products when the first survivor of `b` reaches it; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_sift0` keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (`Ratpoints.Sieve.scanPiece`); natural numbers and integers for the `longs`; no measurements of the phases (`RP_TIC`, `RP_TOC`, `RP_PHASE_TIMING`) and no stop after one of them (`RP_STOP_AFTER`).)

- `sift.c:1043-1054`: excluded (instrumentation): the bit arrays of the piece printed under DEBUG after the first stage and the masks

- `sift.c:1056-1058`: modelled by `Ratpoints.Sieve.siftPiece` (deviation of `Ratpoints.Sieve.siftPiece`: the arm of the first stage chosen by a test on the constant `Ratpoints.chunk`, for the conditional compilation `#if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16)` (`sift.c:452`, `sift.c:575`, `sift.c:943`), which compiles one arm only, and `tail_leg` with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words `malloc` leaves uninitialized; the piece by its first word number `w0` and its number `n` of bit arrays, for `w_low` and `w_high`, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list `specs` and the end `sp1` of the first stage, for the sieve specs `sieves` and the number `sp1` that the sieving thread keeps, the reciprocals of their moduli as the list `mgs`, for the array `magics` of the arguments, and the thunk `checks`, for the table `checks` of the third stage and its length `nchecks = sp3 - sp2`; the class `cl` as a value, for the pointer `cls`; integers for the `longs` of the test `small`, which stay `longs`, and a Boolean for its `int`; the counters `cnt` passed and returned, for the field `n_words` of the sieving thread, which counts the `n Ratpoints.rbaPack` words of the piece (line 548), the product of `UInt64`s as the C code multiplies the `unsigned long` by `RBA_PACK`; the thunk `w` of the products of the check as an argument, where `_ratpoints_check_point` computes the products when the first survivor of `b` reaches it; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_sift0` keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (`Ratpoints.Sieve.scanPiece`); natural numbers and integers for the `longs`; no measurements of the phases (`RP_TIC`, `RP_TOC`, `RP_PHASE_TIMING`) and no stop after one of them (`RP_STOP_AFTER`).)

- `sift.c:1061-1082`: modelled by `Ratpoints.Sieve.scanPiece` (deviation of `Ratpoints.Sieve.scanPiece`: indices for the pointers into the survivors, and a scan that holds its position as an index, where `ratpoints` recovers the word number from a pointer difference; the copy `nums` of the bit array a vector of `Ratpoints.rbaPack` words; the sentinel `~zero` of every build as `Ratpoints.rbaPack` words `~~~0`, written one at a time; natural numbers and integers for the `longs`; the counters `cnt` passed and returned, for the field `n_arrays` of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each bit array, where the C code returns from `_ratpoints_sift0` within the walk through the set bits of a word (`Ratpoints.Sieve.walkWord`).)

- `sift.c:1083-1083`: modelled by `Ratpoints.Sieve.secondStage` (deviation of `Ratpoints.Sieve.secondStage`: the moduli of the stage with their rows and their reciprocals as two lists, for the pointers `ssp` and `mg` that step through the sieve specs and the array `magics` from position `sp1` on and the count `sp2 - sp1`, with no bound `2 RATPOINTS_NUM_PRIMES` on their number (the length of `magics`, `find_points.c:347`), which `ratpoints` never reaches since each entry of `sieve_list` owns its own prime (`find_points.c:329-331`), where the model takes any moduli, which

may repeat; one bit array of `Ratpoints.rbaPack` words, for the vector `nums`; natural numbers and integers for the `long`s, and the sum `v` a natural number for the `long` that `RP_MULMOD` converts to an `unsigned long`, which is not negative under `small`, and which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_add`); a Boolean for the `int` `small`; no count of the ands of the second stage under `RP_PHASE_COUNTS`, and no output under `DEBUG`.)

- `sift.c:1084-1084`: modelled by `Ratpoints.useLongInPhase2` (deviation of `Ratpoints.useLongInPhase2`: a constant of the model, set here, for the macro that the compiler is given or not (`-DUSE_LONG_IN_PHASE_2`), which the C code tests with `#ifdef` and `#ifndef`.)
- `sift.c:1085-1085`: modelled by `Ratpoints.Sieve.secondStage` (deviation of `Ratpoints.Sieve.secondStage`: the moduli of the stage with their rows and their reciprocals as two lists, for the pointers `ssp` and `mg` that step through the sieve specs and the array `magics` from position `sp1` on and the count `sp2 - sp1`, with no bound `2 RATPOINTS_NUM_PRIMES` on their number (the length of `magics`, `find_points.c:347`), which `ratpoints` never reaches since each entry of `sieve_list` owns its own prime (`find_points.c:329-331`), where the model takes any moduli, which may repeat; one bit array of `Ratpoints.rbaPack` words, for the vector `nums`; natural numbers and integers for the `long`s, and the sum `v` a natural number for the `long` that `RP_MULMOD` converts to an `unsigned long`, which is not negative under `small`, and which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_add`); a Boolean for the `int` `small`; no count of the ands of the second stage under `RP_PHASE_COUNTS`, and no output under `DEBUG`.)
- `sift.c:1086-1086`: modelled by `Ratpoints.useLongInPhase2` (deviation of `Ratpoints.useLongInPhase2`: a constant of the model, set here, for the macro that the compiler is given or not (`-DUSE_LONG_IN_PHASE_2`), which the C code tests with `#ifdef` and `#ifndef`.)
- `sift.c:1088-1092`: modelled by `Ratpoints.Sieve.scanPiece` (deviation of `Ratpoints.Sieve.scanPiece`: indices for the pointers into the survivors, and a scan that holds its position as an index, where `ratpoints` recovers the word number from a pointer difference; the copy `nums` of the bit array a vector of `Ratpoints.rbaPack` words; the sentinel `~zero` of every build as `Ratpoints.rbaPack` words `~~~0`, written one at a time; natural numbers and integers for the `long`s; the counters `cnt` passed and returned, for the field `n_arrays` of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each bit array, where the C code returns from `_ratpoints_sift0` within the walk through the set bits of a word (`Ratpoints.Sieve.walkWord`).)
- `sift.c:1094-1111`: modelled by `Ratpoints.Sieve.siftBitArray` (deviation of `Ratpoints.Sieve.siftBitArray`: the arm chosen by the argument `useLong`, so that the guards run both in the same build, for the arm that the build compiles (`#ifdef USE_LONG_IN_PHASE_2`), each arm a function of its own; a function of its own for the body of the loop of the scan over one bit array; no counts under `RP_PHASE_COUNTS`, no stop after a phase (`RP_STOP_AFTER`), and no output under `DEBUG`.)
- `sift.c:1113-1113`: modelled by `Ratpoints.useLongInPhase2` (deviation of `Ratpoints.useLongInPhase2`: a constant of the model, set here, for the macro that the compiler is given or not (`-DUSE_LONG_IN_PHASE_2`), which the C code tests with `#ifdef` and `#ifndef`.)
- `sift.c:1126-1134`: modelled by `Ratpoints.Sieve.siftBitArrayByWords` (deviation of `Ratpoints.Sieve.siftBitArrayByWords`: the stride, the first numerator and the steps as nat-

ural numbers and integers, the powers `2 ^ k` for the shifts `1L << k`, `RBA_LENGTH << k` and `LONG_LENGTH << k`, the last two of which stay `int`s, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below `2 ^ 63`, stay `long`s (`Ratpoints.Sieve.NumClass.fitsLong_numerator`); the words of the bit array read as entries of the vector, for `EXT`; natural numbers and integers for the `long`s; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each walk through the set bits of a word, where the C code returns from `_ratpoints_sift0` within the walk (`Ratpoints.Sieve.walkWord`); no counts under `RP_PHASE_COUNTS` and no stop after a phase (`RP_STOP_AFTER`).)

- sift.c:1135-1136: modelled by `Ratpoints.Sieve.secondStageWord` (deviation of `Ratpoints.Sieve.secondStageWo`
the moduli of the stage with their rows and their reciprocals as two lists, for the pointers `sspk` and `mg` that step through the sieve specs and the array `magics` from position `sp1` on and the count `sp2 - sp1`, with no bound `2 RATPOINTS_NUM_PRIMES` on their number (the length of `magics`, `find_points.c:347`), which `ratpoints` never reaches since each entry of `sieve_list` owns its own prime (`find_points.c:329-331`), where the model takes any moduli, which may repeat; natural numbers and integers for the `long`s, and the sum `v` a natural number for the `long` that `RP_MULMOD` converts to an `unsigned long`, which is not negative under `small`, and which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_add`); a Boolean for the `int small`; no count of the ands of the second stage under `RP_PHASE_COUNTS`.)

- sift.c:1137-1139: modelled by `Ratpoints.Sieve.siftBitArrayByWords` (deviation of `Ratpoints.Sieve.siftBitArrayByWords`: the stride, the first numerator and the steps as natural numbers and integers, the powers `2 ^ k` for the shifts `1L << k`, `RBA_LENGTH << k` and `LONG_LENGTH << k`, the last two of which stay `int`s, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below `2 ^ 63`, stay `long`s (`Ratpoints.Sieve.NumClass.fitsLong_numerator`); the words of the bit array read as entries of the vector, for `EXT`; natural numbers and integers for the `long`s; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each walk through the set bits of a word, where the C code returns from `_ratpoints_sift0` within the walk (`Ratpoints.Sieve.walkWord`); no counts under `RP_PHASE_COUNTS` and no stop after a phase (`RP_STOP_AFTER`).)

- sift.c:1141-1152: modelled by `Ratpoints.Sieve.secondStageWord` (deviation of `Ratpoints.Sieve.secondStageWo`
the moduli of the stage with their rows and their reciprocals as two lists, for the pointers `sspk` and `mg` that step through the sieve specs and the array `magics` from position `sp1` on and the count `sp2 - sp1`, with no bound `2 RATPOINTS_NUM_PRIMES` on their number (the length of `magics`, `find_points.c:347`), which `ratpoints` never reaches since each entry of `sieve_list` owns its own prime (`find_points.c:329-331`), where the model takes any moduli, which may repeat; natural numbers and integers for the `long`s, and the sum `v` a natural number for the `long` that `RP_MULMOD` converts to an `unsigned long`, which is not negative under `small`, and which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_add`); a Boolean for the `int small`; no count of the ands of the second stage under `RP_PHASE_COUNTS`.)

- sift.c:1154-1159: modelled by `Ratpoints.Sieve.siftBitArrayByWords` (deviation of `Ratpoints.Sieve.siftBitArrayByWords`: the stride, the first numerator and the steps as natural numbers and integers, the powers `2 ^ k` for the shifts `1L << k`, `RBA_LENGTH << k` and `LONG_LENGTH << k`, the last two of which stay `int`s, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below `2 ^ 63`, stay `long`s

(Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- sift.c:1161-1171: modelled by Ratpoints.Sieve.walkWord (deviation of Ratpoints.Sieve.walkWord: the lowest set bit by the loop of RP_CTZL for a compiler without the builtin (Ratpoints.Sieve.ctz); one definition for the body at the three places of the extraction, the first word of a bit array, the others, and a word of the arm of the second stage for one word at a time; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, to which RP_CHECK_POINT adds the count of the check (lines 1168, 1240, 1250, 1279, 1287); the return after a stop (lines 1169, 1241, 1251, 1280, 1288) a return from the walk, which its callers pass on, testing the flag after it (Ratpoints.Sieve.siftBitArrayByWords, Ratpoints.Sieve.siftBitArrayWhole, then Ratpoints.Sieve.scanPiece after the bit array), where the C code returns from _ratpoints_sift0; the counters passed and returned, for the field n_bits of the sieving thread (lines 1166, 1233, 1271); no counts under RP_PHASE_COUNTS; no output under DEBUG.)

- sift.c:1172-1173: modelled by Ratpoints.Sieve.siftBitArrayByWords (deviation of Ratpoints.Sieve.siftBitArrayByWords: the stride, the first numerator and the steps as natural numbers and integers, the powers 2 ^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- sift.c:1174-1174: modelled by Ratpoints.useLongInPhase2 (deviation of Ratpoints.useLongInPhase2: a constant of the model, set here, for the macro that the compiler is given or not (-DUSE_LONG_IN_PHASE_2), which the C code tests with #ifdef and #ifndef.)

- sift.c:1175-1202: modelled by Ratpoints.Sieve.secondStage (deviation of Ratpoints.Sieve.secondStage: the moduli of the stage with their rows and their reciprocals as two lists, for the pointers ssp and mg that step through the sieve specs and the array magics from position sp1 on and the count sp2 - sp1, with no bound 2 RATPOINTS_NUM_PRIMES on their number (the length of magics, find_points.c:347), which ratpoints never reaches since each entry of sieve_list owns its own prime (find_points.c:329-331), where the model takes any moduli, which may repeat; one bit array of Ratpoints.rbaPack words, for the vector nums; natural numbers and integers for the longs, and the sum v a natural number for the long that RP_MULMOD converts to an unsigned long, which is not negative under small, and which stays a long where the helper mod reduces it outside small (Ratpoints.Sieve.fitsLong_add); a Boolean for the int small; no count of the ands of the second stage under RP_PHASE_COUNTS, and no output under DEBUG.)

- sift.c:1204-1210: modelled by Ratpoints.Sieve.siftBitArrayWhole (deviation of Ratpoints.Sieve.siftBitArray
the stride, the first numerator and the steps as natural numbers and integers, the powers 2

^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT0 and EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- sift.c:1213-1226: modelled by Ratpoints.Sieve.siftBitArrayWhole (deviation of Ratpoints.Sieve.siftBitArray
the stride, the first numerator and the steps as natural numbers and integers, the powers 2 ^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT0 and EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- sift.c:1227-1255: modelled by Ratpoints.Sieve.walkWord (deviation of Ratpoints.Sieve.walkWord: the lowest set bit by the loop of RP_CTZL for a compiler without the builtin (Ratpoints.Sieve.ctz); one definition for the body at the three places of the extraction, the first word of a bit array, the others, and a word of the arm of the second stage for one word at a time; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, to which RP_CHECK_POINT adds the count of the check (lines 1168, 1240, 1250, 1279, 1287); the return after a stop (lines 1169, 1241, 1251, 1280, 1288) a return from the walk, which its callers pass on, testing the flag after it (Ratpoints.Sieve.siftBitArrayByWords, Ratpoints.Sieve.siftBitArrayWhole, then Ratpoints.Sieve.scanPiece after the bit array), where the C code returns from _ratpoints_sift0; the counters passed and returned, for the field n_bits of the sieving thread (lines 1166, 1233, 1271); no counts under RP_PHASE_COUNTS; no output under DEBUG.)

- sift.c:1257-1264: modelled by Ratpoints.Sieve.siftBitArrayWhole (deviation of Ratpoints.Sieve.siftBitArray
the stride, the first numerator and the steps as natural numbers and integers, the powers 2 ^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT0 and EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- sift.c:1265-1292: modelled by Ratpoints.Sieve.walkWord (deviation of Ratpoints.Sieve.walkWord: the lowest set bit by the loop of RP_CTZL for a compiler without the builtin (Ratpoints.Sieve.ctz); one definition for the body at the three places of the extraction, the first word of a bit array, the others, and a word of the arm of the second stage for one word at a time; natural

numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, to which RP_CHECK_POINT adds the count of the check (lines 1168, 1240, 1250, 1279, 1287); the return after a stop (lines 1169, 1241, 1251, 1280, 1288) a return from the walk, which its callers pass on, testing the flag after it (Ratpoints.Sieve.siftBitArrayByWords, Ratpoints.Sieve.siftBitArrayWhole, then Ratpoints.Sieve.scanPiece after the bit array), where the C code returns from _ratpoints_sift0; the counters passed and returned, for the field n_bits of the sieving thread (lines 1166, 1233, 1271); no counts under RP_PHASE_COUNTS; no output under DEBUG.)

- sift.c:1293-1296: modelled by Ratpoints.Sieve.siftBitArrayWhole (deviation of Ratpoints.Sieve.siftBitArray the stride, the first numerator and the steps as natural numbers and integers, the powers 2 ^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT0 and EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- sift.c:1297-1297: modelled by Ratpoints.useLongInPhase2 (deviation of Ratpoints.useLongInPhase2: a constant of the model, set here, for the macro that the compiler is given or not (-DUSE_LONG_IN_PHASE_2), which the C code tests with #ifdef and #ifndef.)

- sift.c:1298-1299: modelled by Ratpoints.Sieve.scanPiece (deviation of Ratpoints.Sieve.scanPiece: indices for the pointers into the survivors, and a scan that holds its position as an index, where ratpoints recovers the word number from a pointer difference; the copy nums of the bit array a vector of Ratpoints.rbaPack words; the sentinel ~zero of every build as Ratpoints.rbaPack words ~~~0, written one at a time; natural numbers and integers for the longs; the counters cnt passed and returned, for the field n_arrays of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each bit array, where the C code returns from _ratpoints_sift0 within the walk through the set bits of a word (Ratpoints.Sieve.walkWord).)

- sift.c:1300-1305: modelled by Ratpoints.Sieve.siftPiece (deviation of Ratpoints.Sieve.siftPiece: the arm of the first stage chosen by a test on the constant Ratpoints.chunk, for the conditional compilation #if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16) (sift.c:452, sift.c:575, sift.c:943), which compiles one arm only, and tail_leg with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words malloc leaves uninitialized; the piece by its first word number w0 and its number n of bit arrays, for w_low and w_high, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list specs and the end sp1 of the first stage, for the sieve specs sieves and the number sp1 that the sieving thread keeps, the reciprocals of their moduli as the list mgs, for the array magics of the arguments, and the thunk checks, for the table checks of the third stage and its length nchecks = sp3 - sp2; the class cl as a value, for the pointer cls; integers for the longs of the test small, which stay longs, and a Boolean for its int; the counters cnt passed and returned, for the field n_words of the sieving thread, which counts the n Ratpoints.rbaPack words of the piece

(line 548), the product of `UInt64`s as the C code multiplies the `unsigned long` by `RBA_PACK`; the thunk `w` of the products of the check as an argument, where `_ratpoints_check_point` computes the products when the first survivor of `b` reaches it; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_sift0` keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (`Ratpoints.Sieve.scanPiece`); natural numbers and integers for the `long`s; no measurements of the phases (`RP_TIC`, `RP_TOC`, `RP_PHASE_TIMING`) and no stop after one of them (`RP_STOP_AFTER`).)

init.c

- `init.c:30-30`: excluded (other): the include of the private header; the model has no headers
- `init.c:42-46`: modelled by `Ratpoints.Sieve.initBit` (deviation of `Ratpoints.Sieve.initBit`: the form of the production builds, where `RP_INIT_BRANCH` is not set; a Boolean for the `int` entry of `is_f_square`, which is `0` or `1`.)
- `init.c:62-63`: modelled by `Ratpoints.Sieve.initFourDecl` (deviation of `Ratpoints.Sieve.initFourDecl`: the form of the production builds, where `RP_INIT_ONEWAY` is not set; the residues returned, for the variables that the macro declares in its caller; natural numbers for the `long`s.)
- `init.c:64-64`: modelled by `Ratpoints.Sieve.initFourLoop` (deviation of `Ratpoints.Sieve.initFourLoop`: the form of the production builds, where `RP_INIT_ONEWAY` is not set; the table, the prime, the step, the word, the residues and the bounds as arguments and the word, the bit and the residues returned, for the variables of its caller that the macro reads and writes; Booleans for the `int` entries of `is_f_square`; the steps `ab += d4; if (ab >= p) ab -= p;` as conditional values.)
- `init.c:65-70`: modelled by `Ratpoints.Sieve.initFourDecl` (deviation of `Ratpoints.Sieve.initFourDecl`: the form of the production builds, where `RP_INIT_ONEWAY` is not set; the residues returned, for the variables that the macro declares in its caller; natural numbers for the `long`s.)
- `init.c:72-83`: modelled by `Ratpoints.Sieve.initFourLoop` (deviation of `Ratpoints.Sieve.initFourLoop`: the form of the production builds, where `RP_INIT_ONEWAY` is not set; the table, the prime, the step, the word, the residues and the bounds as arguments and the word, the bit and the residues returned, for the variables of its caller that the macro reads and writes; Booleans for the `int` entries of `is_f_square`; the steps `ab += d4; if (ab >= p) ab -= p;` as conditional values.)
- `init.c:88-93`: excluded (instrumentation): RP_INIT_NOACC, a probe for bench_init that leaves out computing the pattern of a row; the tables it produces are wrong
- `init.c:97-98`: excluded (instrumentation): RP_INIT_NOREP, a probe for bench_init that leaves out replicating a row over the words of a bit array; the tables it produces are wrong
- `init.c:99-111`: modelled by `Ratpoints.Sieve.initReplicate` (deviation of `Ratpoints.Sieve.initReplicate`: the row passed and returned, for the pointer `si` of its caller.)
- `init.c:118-125`: modelled by `Ratpoints.Sieve.sieveInit1` (deviation of `Ratpoints.Sieve.sieveInit1`: one function for every prime below 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables

of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; no copy `pattern` of `help0`, which the C code shifts for `help1`.)

- `init.c:126-129`: modelled by `Ratpoints.Sieve.sieve1Rows` (deviation of `Ratpoints.Sieve.sieve1Rows`: `ldp`, `p1`, `diff_shift` and `diff` computed here, for the variables at the head of the function; the mask `p1 & LONG_MASK` as the remainder by 64; a fresh vector of `(p + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`); natural numbers for the `long`s.)
- `init.c:130-130`: modelled by `Ratpoints.Sieve.sieveInit1` (deviation of `Ratpoints.Sieve.sieveInit1`: one function for every prime below 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; no copy `pattern` of `help0`, which the C code shifts for `help1`.)
- `init.c:131-143`: modelled by `Ratpoints.Sieve.sieve1Pattern` (deviation of `Ratpoints.Sieve.sieve1Pattern`: the steps `ab += d; if (ab >= p) ab -= p;` as conditional values; the results of the loop of four bits read as the components of its tuple; natural numbers for the `long`s.)
- `init.c:144-159`: modelled by `Ratpoints.Sieve.sieveInit1` (deviation of `Ratpoints.Sieve.sieveInit1`: one function for every prime below 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; no copy `pattern` of `help0`, which the C code shifts for `help1`.)
- `init.c:160-177`: modelled by `Ratpoints.Sieve.sieve1Rows` (deviation of `Ratpoints.Sieve.sieve1Rows`: `ldp`, `p1`, `diff_shift` and `diff` computed here, for the variables at the head of the function; the mask `p1 & LONG_MASK` as the remainder by 64; a fresh vector of `(p + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`); natural numbers for the `long`s.)
- `init.c:178-183`: modelled by `Ratpoints.Sieve.sieveInit1` (deviation of `Ratpoints.Sieve.sieveInit1`: one function for every prime below 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; no copy `pattern` of `help0`, which the C code shifts for `help1`.)

- init.c:186-194: modelled by `Ratpoints.Sieve.sieveInit2` (deviation of `Ratpoints.Sieve.sieveInit2`: one function for every prime above 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the words `help` a fresh vector, which the loop clears from the last, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)
- init.c:195-196: modelled by `Ratpoints.Sieve.sieve2Rows` (deviation of `Ratpoints.Sieve.sieve2Rows`: the words `help` passed, for the array of the function; `diff_shift` and `diff` computed here, for the variables at the head of the function, and `wp` where it is used, for the variable of the function; a fresh vector of `(p + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`); the shift by `LONG_SHIFT` and the mask by `LONG_MASK` as the division and the remainder by 64.)
- init.c:197-203: modelled by `Ratpoints.Sieve.sieveInit2` (deviation of `Ratpoints.Sieve.sieveInit2`: one function for every prime above 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the words `help` a fresh vector, which the loop clears from the last, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)
- init.c:204-208: modelled by `Ratpoints.Sieve.sieve2Pattern` (deviation of `Ratpoints.Sieve.sieve2Pattern`: the words `help` passed and returned, for the array of the function; `⌊p / 64⌋` computed where it is used, for the variable `wp`; natural numbers for the `long`s.)
- init.c:209-216: modelled by `Ratpoints.Sieve.sieve2Word` (deviation of `Ratpoints.Sieve.sieve2Word`: the residues of the lanes passed and returned, for the variables of the function; the steps `ab += d; if (ab >= p) ab -= p;` as conditional values; the results of the loop of four bits read as the components of its tuple; natural numbers for the `long`s.)
- init.c:217-219: modelled by `Ratpoints.Sieve.sieve2Pattern` (deviation of `Ratpoints.Sieve.sieve2Pattern`: the words `help` passed and returned, for the array of the function; `⌊p / 64⌋` computed where it is used, for the variable `wp`; natural numbers for the `long`s.)
- init.c:220-238: modelled by `Ratpoints.Sieve.sieve2Rows` (deviation of `Ratpoints.Sieve.sieve2Rows`: the words `help` passed, for the array of the function; `diff_shift` and `diff` computed here, for the variables at the head of the function, and `wp` where it is used, for the variable of the function; a fresh vector of `(p + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`); the shift by `LONG_SHIFT` and the mask by `LONG_MASK` as the division and the remainder by 64.)
- init.c:239-244: modelled by `Ratpoints.Sieve.sieveInit2` (deviation of `Ratpoints.Sieve.sieveInit2`: one function for every prime above 64, the prime that of its tables, for the function that the build generates for each prime with the prime a constant (`init_sieve.h`); the tables of the prime and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` from which the C code reads them and into which it stores the row, and the

store of the tables of the rows of the run passed and returned, for the arrays `sieve` of the entries; the words `help` a fresh vector, which the loop clears from the last, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)

- `init.c:246-246`: modelled by `Ratpoints.Sieve.sieveInit` (deviation of `Ratpoints.Sieve.sieveInit`: a function of `p`, for the entry of `p` in the table of the pointers to the functions that the build generates, one for each prime, and that `init.c` includes; the kind of the function, for the pointer to it.)
- `init.c:259-267`: modelled by `Ratpoints.Sieve.layOutPattern` (deviation of `Ratpoints.Sieve.layOutPattern`: the words `rep` a fresh vector of `RP_MODWORDS + 2` zeros, for the array whose first `words` words `lay_out_pattern` clears and whose other words it leaves unset and never reads; a fresh vector of `(m + chunk - 1) rbaPack` words for the row, for the space that it takes from the buffer of the tables (`ba_next`); the step of the offset as a conditional value; the shifts by `LONG_SHIFT` and the masks by `LONG_MASK` as divisions and remainders by 64.)
- `init.c:268-279`: modelled by `Ratpoints.Sieve.repeatPattern` (deviation of `Ratpoints.Sieve.repeatPattern`: the words `rep` passed and returned, for the array of `lay_out_pattern`; the two loops as `while` loops, for the `for` loops that step `o` by `m` and test `i` against two bounds; the or-ing into the next word as a conditional value; the shifts by `LONG_SHIFT` and the masks by `LONG_MASK` as divisions and remainders by 64.)
- `init.c:280-297`: modelled by `Ratpoints.Sieve.layOutPattern` (deviation of `Ratpoints.Sieve.layOutPattern`: the words `rep` a fresh vector of `RP_MODWORDS + 2` zeros, for the array whose first `words` words `lay_out_pattern` clears and whose other words it leaves unset and never reads; a fresh vector of `(m + chunk - 1) rbaPack` words for the row, for the space that it takes from the buffer of the tables (`ba_next`); the step of the offset as a conditional value; the shifts by `LONG_SHIFT` and the masks by `LONG_MASK` as divisions and remainders by 64.)
- `init.c:299-353`: excluded (instrumentation): RP_VERIFY_MODULI, a debugging check, compiled only with -DRP_VERIFY_MODULI: verify_row, which compares a row of a prime power or a product with F(N, b), factor by factor, and aborts on a mismatch
- `init.c:359-368`: modelled by `Ratpoints.Sieve.sieveInitPower` (deviation of `Ratpoints.Sieve.sieveInitPower`: the tables of the prime power and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` whose field `pw` points to the tables and into which the C code stores the row, and the store of the tables of rows of the run passed and returned, for the arrays `sieve` of the entries; the pattern a fresh array, which the loop clears, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)
- `init.c:369-375`: modelled by `Ratpoints.Sieve.powerPatternUnit` (deviation of `Ratpoints.Sieve.powerPatternUn`: the pattern passed and returned, for the array of the function; the step `r += binv; if (r >= m) r -= m;` as a conditional value; natural numbers for the `long`s.)
- `init.c:376-377`: modelled by `Ratpoints.Sieve.sieveInitPower` (deviation of `Ratpoints.Sieve.sieveInitPower`: the tables of the prime power and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` whose field `pw` points to the tables and into which the C code stores the row, and the store of the tables of rows of the run passed and returned, for the arrays `sieve` of the entries; the pattern a fresh array, which the loop clears, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)

- init.c:378-385: modelled by `Ratpoints.Sieve.powerPatternNonunit` (deviation of `Ratpoints.Sieve.powerPatter` the pattern passed and returned, for the array of the function; natural numbers for the `long`s and `unsigned long`s, the product `b inv[x]` below `m ^ 2`.)

- init.c:386-387: modelled by `Ratpoints.Sieve.sieveInitPower` (deviation of `Ratpoints.Sieve.sieveInitPower`: the tables of the prime power and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` whose field `pw` points to the tables and into which the C code stores the row, and the store of the tables of rows of the run passed and returned, for the arrays `sieve` of the entries; the pattern a fresh array, which the loop clears, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)

- init.c:388-390: excluded (instrumentation): RP_VERIFY_MODULI, a debugging check, compiled only with -DRP_VERIFY_MODULI, that compares every row of a prime power with F(N, b) (verify_row) and aborts on a mismatch

- init.c:391-393: modelled by `Ratpoints.Sieve.sieveInitPower` (deviation of `Ratpoints.Sieve.sieveInitPower`: the tables of the prime power and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` whose field `pw` points to the tables and into which the C code stores the row, and the store of the tables of rows of the run passed and returned, for the arrays `sieve` of the entries; the pattern a fresh array, which the loop clears, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)

- init.c:399-406: modelled by `Ratpoints.Sieve.sieveInitProduct` (deviation of `Ratpoints.Sieve.sieveInitProdu` the store of the tables of rows passed and returned, for the entries of the factors that the C code reaches through the pointers `factor` and for the entry `se` into which it stores the row; a fresh vector of `(m + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`) after the rows of the factors; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; the number of the factors the length of their list, for the count `nf`.)

- init.c:407-414: modelled by `Ratpoints.Sieve.factorRows` (deviation of `Ratpoints.Sieve.factorRows`: the rows of the factors an array returned with the store, for the array `frow` of the function and the tables of rows in the entries of the factors; the row read plainly, for the acquire load `RP_ACQUIRE` of a build with threads, which the model does not have; the number of the factors the length of their list, for the count `nf`.)

- init.c:415-418: modelled by `Ratpoints.Sieve.sieveInitProduct` (deviation of `Ratpoints.Sieve.sieveInitProdu` the store of the tables of rows passed and returned, for the entries of the factors that the C code reaches through the pointers `factor` and for the entry `se` into which it stores the row; a fresh vector of `(m + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`) after the rows of the factors; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; the number of the factors the length of their list, for the count `nf`.)

- init.c:419-423: modelled by `Ratpoints.Sieve.productCopy` (deviation of `Ratpoints.Sieve.productCopy`: the row passed and returned, for the pointer of the function; the loop over `x` as a `while` loop, for the `for` loop that steps `x` by `q`; the row of the factor as the vector of its words, which the store of a bit array reads.)

- init.c:424-429: modelled by Ratpoints.Sieve.productAnd (deviation of Ratpoints.Sieve.productAnd: the row passed and returned, for the pointer of the function; the loop over x as a while loop, for the for loop that steps x by q; the number of the factors the length of their list, for the count nf.)
- init.c:430-430: modelled by Ratpoints.Sieve.sieveInitProduct (deviation of Ratpoints.Sieve.sieveInitProdu the store of the tables of rows passed and returned, for the entries of the factors that the C code reaches through the pointers factor and for the entry se into which it stores the row; a fresh vector of (m + chunk - 1) rbaPack words, for the space that the builder takes from the buffer of the tables (ba_next) after the rows of the factors; the row stored plainly, for the release store of RP_PUBLISH in a build with threads, which the model does not have; the number of the factors the length of their list, for the count nf.)
- init.c:431-442: excluded (instrumentation): RP_VERIFY_MODULI, a debugging check, compiled only with -DRP_VERIFY_MODULI, that compares every row of a product with F(N, b), factor by factor (verify_row), and aborts on a mismatch
- init.c:443-445: modelled by Ratpoints.Sieve.sieveInitProduct (deviation of Ratpoints.Sieve.sieveInitProdu the store of the tables of rows passed and returned, for the entries of the factors that the C code reaches through the pointers factor and for the entry se into which it stores the row; a fresh vector of (m + chunk - 1) rbaPack words, for the space that the builder takes from the buffer of the tables (ba_next) after the rows of the factors; the row stored plainly, for the release store of RP_PUBLISH in a build with threads, which the model does not have; the number of the factors the length of their list, for the count nf.)

sturm.c

- sturm.c:29-32: excluded (other): the includes of math.h, stdlib.h and the header of the library; the model has no headers
- sturm.c:67-84: modelled by Ratpoints.Sturm.evalSign (deviation of Ratpoints.Sturm.evalSign: the row as an array of integers, for the GMP integers cofs and the long degree, which is the size of the array less one; integers for the GMP integers work[0] and work[1], the multiplication by a power of 2 for mpz_mul_2exp, and Int.sign for mpz_sgn; Int64 for the longs num, denexp and e; the steps counted up from 0 to deg - 1, for the index n that counts down; no scratch integers of the arguments.)
- sturm.c:86-97: modelled by Ratpoints.Sturm.computeSturm (deviation of Ratpoints.Sturm.computeSturm: the coefficients, the degree, the depth and the search intervals as arguments, for the fields cof, degree, sturm, domain and num_inter of the arguments; natural numbers for the longs degree and iter, which is from 0 to 55 (the search clamps it, find_points.c:4682); the result as an option of an array, for the long that the C code returns and the intervals it writes into domain with their number num_inter: none for -1, the empty array for 0, with num_inter = 0, and else the intervals, their number the return value; the table of the chain made by Ratpoints.Sturm.chain, whose rows the step reads, for the table on the stack with the degrees, which _ratpoints_compute_sturm fills itself, the index of the last row the size of the chain less one, for the k of line 163; the array of the intervals kept made by Ratpoints.Sturm.iterate, for the array ivlocal on the stack of 1 + ⌊degree / 2⌋ intervals, which suffice assuming Sturm's theorem (Ratpoints.Sturm.size_run_iterate_le_of_chain_eq_some); integers for the GMP integers and Int.sign for mpz_sgn; no mpz_init and mpz_clear, no copy of

the caller's intervals on the heap and no failure for lack of memory (RATPOINTS_NO_MEMORY), which the model cannot meet.)

- sturm.c:99-163: modelled by Ratpoints.Sturm.chain (deviation of Ratpoints.Sturm.chain: the rows as arrays of their coefficients up to their degree, for the table of (d + 1)^2 GMP integers and the array of degrees, with no mpz_init and mpz_clear (a chain has at most d + 1 rows, Ratpoints.Sturm.IsChain.size_le, and its row k + 1 at most d - k entries, Ratpoints.Sturm.IsChain.size_getElem!_add_one_add_le): an array has one entry more than the degree of its row, and a remainder that becomes zero is the empty array, where the degree in the C code reaches -1; a reduction step drops the leading coefficient, which it makes 0, where the C code lowers the degree; the rows are handed back, and none for the return of -1, where _ratpoints_compute_sturm keeps them in its table for the rest of the Sturm step (Ratpoints.Sturm.computeSturm), and the index of the last row, which the C code leaves in k and sets to d after the loop for d = 1, is the size of the array less one; a list of integers for the array of GMP integers of the coefficients, local values for the scratch integers work[0], work[1] and work[2], and Lean's / for mpz_fdiv_q, which agree on the divisions of the chain, all exact.)

- sturm.c:165-175: modelled by Ratpoints.Sturm.countsAtInfinity (deviation of Ratpoints.Sturm.countsAtInfin: natural numbers for the long counts, which stay at most k; the rows as arrays, their degrees the sizes less one, for the arrays sturm and sturm_degs, and Int.sign for mpz_sgn; the counts returned, for the locals of _ratpoints_compute_sturm.)

- sturm.c:177-190: modelled by Ratpoints.Sturm.computeSturm (deviation of Ratpoints.Sturm.computeSturm: the coefficients, the degree, the depth and the search intervals as arguments, for the fields cof, degree, sturm, domain and num_inter of the arguments; natural numbers for the longs degree and iter, which is from 0 to 55 (the search clamps it, find_points.c:4682); the result as an option of an array, for the long that the C code returns and the intervals it writes into domain with their number num_inter: none for -1, the empty array for 0, with num_inter = 0, and else the intervals, their number the return value; the table of the chain made by Ratpoints.Sturm.chain, whose rows the step reads, for the table on the stack with the degrees, which _ratpoints_compute_sturm fills itself, the index of the last row the size of the chain less one, for the k of line 163; the array of the intervals kept made by Ratpoints.Sturm.iterate, for the array ivlocal on the stack of 1 + ⌊degree / 2⌋ intervals, which suffice assuming Sturm's theorem (Ratpoints.Sturm.size_run_iterate_le_of_chain_eq_some); integers for the GMP integers and Int.sign for mpz_sgn; no mpz_init and mpz_clear, no copy of the caller's intervals on the heap and no failure for lack of memory (RATPOINTS_NO_MEMORY), which the model cannot meet.)

- sturm.c:191-192: modelled by Ratpoints.Sturm.longMax (deviation of Ratpoints.Sturm.longMax: one constant for the two locals max and min = -max.)

- sturm.c:193-194: modelled by Ratpoints.Sturm.computeSturm (deviation of Ratpoints.Sturm.computeSturm: the coefficients, the degree, the depth and the search intervals as arguments, for the fields cof, degree, sturm, domain and num_inter of the arguments; natural numbers for the longs degree and iter, which is from 0 to 55 (the search clamps it, find_points.c:4682); the result as an option of an array, for the long that the C code returns and the intervals it writes into domain with their number num_inter: none for -1, the empty array for 0, with num_inter = 0, and else the intervals, their number the return value; the table of the chain made by Ratpoints.Sturm.chain, whose rows the step reads, for the table on the stack with the degrees, which _ratpoints_compute_sturm fills itself, the index of the last row the size of the chain less

one, for the k of line 163; the array of the intervals kept made by Ratpoints.Sturm.iterate, for the array ivlocal on the stack of 1 + ⌊degree / 2⌋ intervals, which suffice assuming Sturm's theorem (Ratpoints.Sturm.size_run_iterate_le_of_chain_eq_some); integers for the GMP integers and Int.sign for mpz_sgn; no mpz_init and mpz_clear, no copy of the caller's intervals on the heap and no failure for lack of memory (RATPOINTS_NO_MEMORY), which the model cannot meet.)

- sturm.c:197-208: modelled by Ratpoints.Sturm.iterate (deviation of Ratpoints.Sturm.iterate: a recursive function of its own, for the nested function of gcc, with the intervals kept passed and returned (Ratpoints.Sturm.addInterval), for the array ivlocal and the pointer iptr that it reaches, and the chain, the index of its last row and the depth of the step as arguments, for the arrays sturm and sturm_degs, k and iter that it reaches; natural numbers for the long counts, depths and the long iter, which is from 0 to 55 (the search clamps it, find_points.c:4682), integers for the long signs, and Int64 for the other longs; the recursion ends since the depth grows towards iter.)

- sturm.c:209-227: modelled by Ratpoints.Sturm.addInterval (deviation of Ratpoints.Sturm.addInterval: a function of its own for the branch of iterate that keeps an interval; the intervals kept as an array, passed and returned, for the array ivlocal on the stack and the pointer iptr of _ratpoints_compute_sturm, which iterate reaches as a nested function of gcc, with no bound 1 + ⌊degree / 2⌋ on their number, the size of ivlocal, which the C code relies on and which holds assuming Sturm's theorem (Ratpoints.Sturm.size_run_iterate_le_of_chain_eq_some); the doubles as their bits (Ratpoints.Double); Int64 for the longs, converted to integers for the int casts of the exponents.)

- sturm.c:228-233: modelled by Ratpoints.Sturm.iterate (deviation of Ratpoints.Sturm.iterate: a recursive function of its own, for the nested function of gcc, with the intervals kept passed and returned (Ratpoints.Sturm.addInterval), for the array ivlocal and the pointer iptr that it reaches, and the chain, the index of its last row and the depth of the step as arguments, for the arrays sturm and sturm_degs, k and iter that it reaches; natural numbers for the long counts, depths and the long iter, which is from 0 to 55 (the search clamps it, find_points.c:4682), integers for the long signs, and Int64 for the other longs; the recursion ends since the depth grows towards iter.)

- sturm.c:234-251: modelled by Ratpoints.Sturm.midpoint (deviation of Ratpoints.Sturm.midpoint: the point returned as a pair, for the locals nm and dem of iterate; Int64 for the longs, >>> on Int64 the arithmetic shift of the C code; the arithmetic of Int64 wrapping where that of the longs overflows, which it does not up to the depth 55 at which the search clamps the step, where every finite end is n / 2 ^ e with |n| ≤ 2 ^ 53.)

- sturm.c:252-260: modelled by Ratpoints.Sturm.signsAt (deviation of Ratpoints.Sturm.signsAt: a function of its own for the evaluation of the chain at the midpoint in iterate, the signs and the count returned, for its locals s0, s1 and cmid; integers for the long signs and a natural number for the long count; the rows as arrays, for the arrays sturm and sturm_degs.)

- sturm.c:261-267: modelled by Ratpoints.Sturm.iterate (deviation of Ratpoints.Sturm.iterate: a recursive function of its own, for the nested function of gcc, with the intervals kept passed and returned (Ratpoints.Sturm.addInterval), for the array ivlocal and the pointer iptr that it reaches, and the chain, the index of its last row and the depth of the step as arguments, for the arrays sturm and sturm_degs, k and iter that it reaches; natural numbers for the long counts, depths and the long iter, which is from 0 to 55 (the search clamps

it, `find_points.c:4682`), integers for the `long` signs, and `Int64` for the other `long`s; the recursion ends since the depth grows towards `iter`.)

- `sturm.c:269-273`: modelled by `Ratpoints.Sturm.computeSturm` (deviation of `Ratpoints.Sturm.computeSturm`: the coefficients, the degree, the depth and the search intervals as arguments, for the fields `cof`, `degree`, `sturm`, `domain` and `num_inter` of the arguments; natural numbers for the `long`s `degree` and `iter`, which is from `0` to `55` (the search clamps it, `find_points.c:4682`); the result as an option of an array, for the `long` that the C code returns and the intervals it writes into `domain` with their number `num_inter`: none for `-1`, the empty array for `0`, with `num_inter = 0`, and else the intervals, their number the return value; the table of the chain made by `Ratpoints.Sturm.chain`, whose rows the step reads, for the table on the stack with the degrees, which `_ratpoints_compute_sturm` fills itself, the index of the last row the size of the chain less one, for the `k` of line 163; the array of the intervals kept made by `Ratpoints.Sturm.iterate`, for the array `ivlocal` on the stack of `1 + ⌊degree / 2⌋` intervals, which suffice assuming Sturm's theorem (`Ratpoints.Sturm.size_run_iterate_le_of_chain_eq_some`); integers for the GMP integers and `Int.sign` for `mpz_sgn`; no `mpz_init` and `mpz_clear`, no copy of the caller's intervals on the heap and no failure for lack of memory (`RATPOINTS_NO_MEMORY`), which the model cannot meet.)

- `sturm.c:274-275`: excluded (memory): the copy of the caller's search intervals on the heap, from which the intersection is made; the model reads the caller's intervals and makes a new array (Ratpoints.Sturm.intersect)

- `sturm.c:276-276`: modelled by `Ratpoints.Sturm.intersect` (deviation of `Ratpoints.Sturm.intersect`: the intervals of the caller read from the array it gives, where the C code reads them from a copy on the heap (lines 274-275, 286) and writes the intersection over them; the intersection returned as a new array, with no bound on its length, for the writes into `ivlist`, the caller's array, which must have room for `1 + ⌊degree / 2⌋` intervals more than it holds (lines 52-53), as the C code relies on and as suffices assuming Sturm's theorem (`Ratpoints.Sturm.size_run_computeSturm_le`), and for the number `num_inter` of the arguments that it sets; natural numbers for the `long` indices.)

- `sturm.c:278-284`: excluded (memory): the failure for lack of memory for the copy of the caller's search intervals (RATPOINTS_NO_MEMORY), with the clears of the table of the chain before the return, which the model cannot meet

- `sturm.c:285-288`: modelled by `Ratpoints.Sturm.intersect` (deviation of `Ratpoints.Sturm.intersect`: the intervals of the caller read from the array it gives, where the C code reads them from a copy on the heap (lines 274-275, 286) and writes the intersection over them; the intersection returned as a new array, with no bound on its length, for the writes into `ivlist`, the caller's array, which must have room for `1 + ⌊degree / 2⌋` intervals more than it holds (lines 52-53), as the C code relies on and as suffices assuming Sturm's theorem (`Ratpoints.Sturm.size_run_computeSturm_le`), and for the number `num_inter` of the arguments that it sets; natural numbers for the `long` indices.)

- `sturm.c:289-322`: modelled by `Ratpoints.Sturm.meet` (deviation of `Ratpoints.Sturm.meet`: a function of its own for the body of the loop, the interval noted and the cursor to move returned, for the write into `ivlist[n]` and the increments of `n`, `n1` and `n2`.)

- `sturm.c:323-323`: modelled by `Ratpoints.Sturm.intersect` (deviation of `Ratpoints.Sturm.intersect`: the intervals of the caller read from the array it gives, where the C code reads them from

a copy on the heap (lines 274-275, 286) and writes the intersection over them; the intersection returned as a new array, with no bound on its length, for the writes into `ivlist`, the caller's array, which must have room for `1 + ⌊degree / 2⌋` intervals more than it holds (lines 52-53), as the C code relies on and as suffices assuming Sturm's theorem (`Ratpoints.Sturm.size_run_computeSturm_le`), and for the number `num_inter` of the arguments that it sets; natural numbers for the `long` indices.)

- sturm.c:324-324: excluded (memory): the release of the copy of the caller's search intervals
- sturm.c:325-325: modelled by `Ratpoints.Sturm.intersect` (deviation of `Ratpoints.Sturm.intersect`: the intervals of the caller read from the array it gives, where the C code reads them from a copy on the heap (lines 274-275, 286) and writes the intersection over them; the intersection returned as a new array, with no bound on its length, for the writes into `ivlist`, the caller's array, which must have room for `1 + ⌊degree / 2⌋` intervals more than it holds (lines 52-53), as the C code relies on and as suffices assuming Sturm's theorem (`Ratpoints.Sturm.size_run_computeSturm_le`), and for the number `num_inter` of the arguments that it sets; natural numbers for the `long` indices.)
- sturm.c:326-332: modelled by `Ratpoints.Sturm.computeSturm` (deviation of `Ratpoints.Sturm.computeSturm`: the coefficients, the degree, the depth and the search intervals as arguments, for the fields `cof`, `degree`, `sturm`, `domain` and `num_inter` of the arguments; natural numbers for the `long`s `degree` and `iter`, which is from `0` to `55` (the search clamps it, `find_points.c:4682`); the result as an option of an array, for the `long` that the C code returns and the intervals it writes into `domain` with their number `num_inter`: none for `-1`, the empty array for `0`, with `num_inter = 0`, and else the intervals, their number the return value; the table of the chain made by `Ratpoints.Sturm.chain`, whose rows the step reads, for the table on the stack with the degrees, which `_ratpoints_compute_sturm` fills itself, the index of the last row the size of the chain less one, for the `k` of line 163; the array of the intervals kept made by `Ratpoints.Sturm.iterate`, for the array `ivlocal` on the stack of `1 + ⌊degree / 2⌋` intervals, which suffice assuming Sturm's theorem (`Ratpoints.Sturm.size_run_iterate_le_of_chain_eq_some`); integers for the GMP integers and `Int.sign` for `mpz_sgn`; no `mpz_init` and `mpz_clear`, no copy of the caller's intervals on the heap and no failure for lack of memory (`RATPOINTS_NO_MEMORY`), which the model cannot meet.)

rp-private.h

- rp-private.h:30-35: excluded (other): the includes of the headers of the C library, among them float.h and math.h of the doubles of the search intervals; the model has no headers
- rp-private.h:41-43: modelled by `Ratpoints.fitsLong` (deviation of `Ratpoints.fitsLong`: the bounds of a `long` of 64 bits, fixed, where the C code does not compile unless an `unsigned long` has exactly 64 bits.)
- rp-private.h:47-47: modelled by `Ratpoints.fitsLong` (deviation of `Ratpoints.fitsLong`: the bounds of a `long` of 64 bits, fixed, where the C code does not compile unless an `unsigned long` has exactly 64 bits.)
- rp-private.h:49-51: modelled by `Ratpoints.rbaLength` (deviation of `Ratpoints.rbaLength`: computed from `Ratpoints.rbaPack`, where each build sets `RBA_LENGTH` by a literal, and the build with 64-bit bit arrays by `LONG_LENGTH`, and its logarithm `RBA_SHIFT` likewise (`Ratpoints.rbaShift`); the shifts by `RBA_SHIFT` and by `LONG_SHIFT` are divisions by `rbaLength`

and by 64, which round down also for a negative value, as the arithmetic shifts of gcc do, and the masks by `RBA_LENGTH - 1` and by `LONG_MASK` Euclidean remainders.)

- `rp-private.h:55-58`: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)
- `rp-private.h:60-60`: excluded (other): the include of the header of the library; the model has no headers
- `rp-private.h:76-84`: excluded (threads): the atomic loads and stores of the fields that the threads of a pool share (RP_ACQUIRE, RP_RELAXED, RP_PUBLISH), and their plain forms for a build without threads; on one thread an atomic load or store is a plain read or write, which the model makes
- `rp-private.h:86-86`: excluded (other): FLOOR, which the C code uses nowhere: find_points.c names it only in a comment
- `rp-private.h:87-87`: modelled by `Ratpoints.Setup.slope`, `Ratpoints.arraySizeAt` (deviation of `Ratpoints.Setup.slope`: integers for the `int`s, and the slope as a natural number; no output under `DEBUG`.) (deviation of `Ratpoints.arraySizeAt`: a `Nat` for the `long` field, so that only `0` stands for the default, which the library takes for every value `≤ 0`; `64` for `LONG_LENGTH`; and the macro `CEIL` of a natural number `x` by `d` as `(x + d - 1) / d`, which is its value for every `x ≥ 0`.)
- `rp-private.h:113-113`: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)
- `rp-private.h:123-123`: excluded (other): the include of the vector intrinsics in the build USE_AVX512, with 512-bit bit arrays; the word operations of the model need none
- `rp-private.h:125-125`: modelled by `Ratpoints.rbaLength` (deviation of `Ratpoints.rbaLength`: computed from `Ratpoints.rbaPack`, where each build sets `RBA_LENGTH` by a literal, and the build with 64-bit bit arrays by `LONG_LENGTH`, and its logarithm `RBA_SHIFT` likewise (`Ratpoints.rbaShift`); the shifts by `RBA_SHIFT` and by `LONG_SHIFT` are divisions by `rbaLength` and by 64, which round down also for a negative value, as the arithmetic shifts of gcc do, and the masks by `RBA_LENGTH - 1` and by `LONG_MASK` Euclidean remainders.)
- `rp-private.h:126-126`: modelled by `Ratpoints.rbaShift` (deviation of `Ratpoints.rbaShift`: the value of the default build, set here, where each build sets it by a literal, and the build with 64-bit bit arrays by `LONG_SHIFT`; the model shifts by none of them: the shifts by `RBA_SHIFT` are divisions by `Ratpoints.rbaLength`.)
- `rp-private.h:127-128`: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)
- `rp-private.h:129-132`: modelled by `Ratpoints.Sieve.andRows`, `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRows`: the bit arrays set to the class word by a loop that

counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`, `sift.c:954`); the moduli of the stage with their rows and their reciprocals as two lists, for the first `sp1` sieve specs and the array `magics` from position `0` on; the value `-w_low - offset` written out in both reductions, for the `long a`, which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_neg_sub`); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the `long`s; a Boolean for the `int small`; no output under `DEBUG`.) (deviation of `Ratpoints.Sieve.andRowsChunked`: no moduli tested by the size of the local array of the sieve specs, for `sp1 == 0`; the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`); the end of the whole chunks the index `full`, for the word number `w_high_full`, which is also the index of the first bit array of the tail, for the pointer `surv` after the chunks; natural numbers and integers for the `long`s.)

- rp-private.h:133-133: modelled by `Ratpoints.Sieve.scanPiece` (deviation of `Ratpoints.Sieve.scanPiece`: indices for the pointers into the survivors, and a scan that holds its position as an index, where `ratpoints` recovers the word number from a pointer difference; the copy `nums` of the bit array a vector of `Ratpoints.rbaPack` words; the sentinel `~zero` of every build as `Ratpoints.rbaPack` words `~~~0`, written one at a time; natural numbers and integers for the `long`s; the counters `cnt` passed and returned, for the field `n_arrays` of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each bit array, where the C code returns from `_ratpoints_sift0` within the walk through the set bits of a word (`Ratpoints.Sieve.walkWord`).)

- rp-private.h:134-134: modelled by `Ratpoints.Sieve.andBitArray` (deviation of `Ratpoints.Sieve.andBitArray`: one definition for every build, on bit arrays given by their indices in arrays of words, each bit array `rbaPack` consecutive words, for the `&` on the vector type of the build.)

- rp-private.h:135-136: modelled by `Ratpoints.Sieve.siftBitArrayWhole` (deviation of `Ratpoints.Sieve.siftBitArrayWhole`: the stride, the first numerator and the steps as natural numbers and integers, the powers `2 ^ k` for the shifts `1L << k`, `RBA_LENGTH << k` and `LONG_LENGTH << k`, the last two of which stay `int`s, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below `2 ^ 63`, stay `long`s (`Ratpoints.Sieve.NumClass.fitsLong_numerator`); the words of the bit array read as entries of the vector, for `EXT0` and `EXT`; natural numbers and integers for the `long`s; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from `_ratpoints_sift0` within the walk (`Ratpoints.Sieve.walkWord`); no counts under `RP_PHASE_COUNTS` and no stop after a phase (`RP_STOP_AFTER`).)

- rp-private.h:137-152: modelled by `Ratpoints.Sieve.isZeroAt` (deviation of `Ratpoints.Sieve.isZeroAt`: one definition for every build, on a bit array given by its index in an array of words, `rbaPack` consecutive words that it compares with `0` one at a time, for the tests of the builds on the

vector type, with vector instructions or by or-ing its words; the test for zero `TESTZ` also stands for its negation `TEST`.)

- rp-private.h:153-155: modelled by `Ratpoints.Sieve.clearLowBits` (deviation of `Ratpoints.Sieve.clearLowBits`: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is the first `Ratpoints.rbaPack` words of an array, for the vector type of the build.)
- rp-private.h:156-159: modelled by `Ratpoints.Sieve.clearHighBits` (deviation of `Ratpoints.Sieve.clearHighBits`: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is given by its index in an array of words, `rbaPack` consecutive words, for a pointer to a value of the vector type of the build.)
- rp-private.h:160-165: modelled by `Ratpoints.chunk` (deviation of `Ratpoints.chunk`: 16, the value of the default build (`USE_AVX`), set here, where the C code chooses the value by the macros that the compiler is given, each build with vector registers taking 16 and the build with 64-bit bit arrays 1 unless `RATPOINTS_CHUNK` is given.)
- rp-private.h:167-167: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)
- rp-private.h:170-170: excluded (other): the include of the vector intrinsics in the build USE_AVX, with 256-bit bit arrays; the word operations of the model need none
- rp-private.h:172-172: modelled by `Ratpoints.rbaLength` (deviation of `Ratpoints.rbaLength`: computed from `Ratpoints.rbaPack`, where each build sets `RBA_LENGTH` by a literal, and the build with 64-bit bit arrays by `LONG_LENGTH`, and its logarithm `RBA_SHIFT` likewise (`Ratpoints.rbaShift`); the shifts by `RBA_SHIFT` and by `LONG_SHIFT` are divisions by `rbaLength` and by 64, which round down also for a negative value, as the arithmetic shifts of gcc do, and the masks by `RBA_LENGTH - 1` and by `LONG_MASK` Euclidean remainders.)
- rp-private.h:173-173: modelled by `Ratpoints.rbaShift` (deviation of `Ratpoints.rbaShift`: the value of the default build, set here, where each build sets it by a literal, and the build with 64-bit bit arrays by `LONG_SHIFT`; the model shifts by none of them: the shifts by `RBA_SHIFT` are divisions by `Ratpoints.rbaLength`.)
- rp-private.h:174-175: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)
- rp-private.h:176-176: modelled by `Ratpoints.Sieve.andBitArray` (deviation of `Ratpoints.Sieve.andBitArray`: one definition for every build, on bit arrays given by their indices in arrays of words, each bit array `rbaPack` consecutive words, for the `&` on the vector type of the build.)
- rp-private.h:177-178: modelled by `Ratpoints.Sieve.siftBitArrayWhole` (deviation of `Ratpoints.Sieve.siftBitArrayWhole`: the stride, the first numerator and the steps as natural numbers and integers, the powers `2 ^ k` for the shifts `1L << k`, `RBA_LENGTH << k` and

`LONG_LENGTH << k`, the last two of which stay `int`s, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below `2 ^ 63`, stay `long`s (`Ratpoints.Sieve.NumClass.fitsLong_numerator`); the words of the bit array read as entries of the vector, for `EXT0` and `EXT`; natural numbers and integers for the `long`s; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from `_ratpoints_sift0` within the walk (`Ratpoints.Sieve.walkWord`); no counts under `RP_PHASE_COUNTS` and no stop after a phase (`RP_STOP_AFTER`).)

- rp-private.h:179-202: modelled by `Ratpoints.Sieve.isZeroAt` (deviation of `Ratpoints.Sieve.isZeroAt`: one definition for every build, on a bit array given by its index in an array of words, `rbaPack` consecutive words that it compares with `0` one at a time, for the tests of the builds on the vector type, with vector instructions or by or-ing its words; the test for zero `TESTZ` also stands for its negation `TEST`.)

- rp-private.h:203-204: modelled by `Ratpoints.Sieve.andRows`, `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRows`: the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`, `sift.c:954`); the moduli of the stage with their rows and their reciprocals as two lists, for the first `sp1` sieve specs and the array `magics` from position `0` on; the value `-w_low - offset` written out in both reductions, for the `long a`, which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_neg_sub`); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the `long`s; a Boolean for the `int small`; no output under `DEBUG`.) (deviation of `Ratpoints.Sieve.andRowsChunked`: no moduli tested by the size of the local array of the sieve specs, for `sp1 == 0`; the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`); the end of the whole chunks the index `full`, for the word number `w_high_full`, which is also the index of the first bit array of the tail, for the pointer `surv` after the chunks; natural numbers and integers for the `long`s.)

- rp-private.h:205-205: modelled by `Ratpoints.Sieve.scanPiece` (deviation of `Ratpoints.Sieve.scanPiece`: indices for the pointers into the survivors, and a scan that holds its position as an index, where `ratpoints` recovers the word number from a pointer difference; the copy `nums` of the bit array a vector of `Ratpoints.rbaPack` words; the sentinel `~zero` of every build as `Ratpoints.rbaPack` words `~~~0`, written one at a time; natural numbers and integers for the `long`s; the counters `cnt` passed and returned, for the field `n_arrays` of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each bit array, where the C code returns from `_ratpoints_sift0` within the walk through the set bits of a word (`Ratpoints.Sieve.walkWord`).)

- rp-private.h:206-213: modelled by `Ratpoints.Sieve.clearLowBits` (deviation of `Ratpoints.Sieve.clearLowBits`: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the

build with 64-bit bit arrays the case of one word; the bit array is the first `Ratpoints.rbaPack` words of an array, for the vector type of the build.)

- rp-private.h:214-221: modelled by `Ratpoints.Sieve.clearHighBits` (deviation of `Ratpoints.Sieve.clearHighBits`: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is given by its index in an array of words, `rbaPack` consecutive words, for a pointer to a value of the vector type of the build.)

- rp-private.h:222-224: modelled by `Ratpoints.chunk` (deviation of `Ratpoints.chunk`: 16, the value of the default build (`USE_AVX`), set here, where the C code chooses the value by the macros that the compiler is given, each build with vector registers taking 16 and the build with 64-bit bit arrays 1 unless `RATPOINTS_CHUNK` is given.)

- rp-private.h:226-226: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- rp-private.h:229-229: excluded (other): the include of the vector intrinsics in the build USE_AVX128, with 128-bit bit arrays; the word operations of the model need none

- rp-private.h:231-231: modelled by `Ratpoints.rbaLength` (deviation of `Ratpoints.rbaLength`: computed from `Ratpoints.rbaPack`, where each build sets `RBA_LENGTH` by a literal, and the build with 64-bit bit arrays by `LONG_LENGTH`, and its logarithm `RBA_SHIFT` likewise (`Ratpoints.rbaShift`); the shifts by `RBA_SHIFT` and by `LONG_SHIFT` are divisions by `rbaLength` and by 64, which round down also for a negative value, as the arithmetic shifts of gcc do, and the masks by `RBA_LENGTH - 1` and by `LONG_MASK` Euclidean remainders.)

- rp-private.h:232-232: modelled by `Ratpoints.rbaShift` (deviation of `Ratpoints.rbaShift`: the value of the default build, set here, where each build sets it by a literal, and the build with 64-bit bit arrays by `LONG_SHIFT`; the model shifts by none of them: the shifts by `RBA_SHIFT` are divisions by `Ratpoints.rbaLength`.)

- rp-private.h:233-234: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- rp-private.h:235-235: modelled by `Ratpoints.Sieve.andRows`, `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRows`: the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`, `sift.c:954`); the moduli of the stage with their rows and their reciprocals as two lists, for the first `sp1` sieve specs and the array `magics` from position `0` on; the value `-w_low - offset` written out in both reductions, for the `long a`, which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_neg_sub`); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the `longs`; a Boolean for the `int small`; no output under

DEBUG.) (deviation of Ratpoints.Sieve.andRowsChunked: no moduli tested by the size of the local array of the sieve specs, for sp1 == 0; the bit arrays set to the class word by a loop that counts up, bit array n - 1 - i at the step i, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector RBA(packed) that get_2adic_info makes once per class (find_points.c:1098) and _ratpoints_sift0 copies into each bit array (bits64, sift.c:519); the end of the whole chunks the index full, for the word number w_high_full, which is also the index of the first bit array of the tail, for the pointer surv after the chunks; natural numbers and integers for the longs.)

- rp-private.h:236-236: modelled by Ratpoints.Sieve.scanPiece (deviation of Ratpoints.Sieve.scanPiece: indices for the pointers into the survivors, and a scan that holds its position as an index, where ratpoints recovers the word number from a pointer difference; the copy nums of the bit array a vector of Ratpoints.rbaPack words; the sentinel ~zero of every build as Ratpoints.rbaPack words ~~~0, written one at a time; natural numbers and integers for the longs; the counters cnt passed and returned, for the field n_arrays of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each bit array, where the C code returns from _ratpoints_sift0 within the walk through the set bits of a word (Ratpoints.Sieve.walkWord).)

- rp-private.h:237-237: modelled by Ratpoints.Sieve.andBitArray (deviation of Ratpoints.Sieve.andBitArray: one definition for every build, on bit arrays given by their indices in arrays of words, each bit array rbaPack consecutive words, for the & on the vector type of the build.)

- rp-private.h:238-239: modelled by Ratpoints.Sieve.siftBitArrayWhole (deviation of Ratpoints.Sieve.siftBitArrayWhole: the stride, the first numerator and the steps as natural numbers and integers, the powers 2 ^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT0 and EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- rp-private.h:241-241: modelled by Ratpoints.Sieve.isZeroAt (deviation of Ratpoints.Sieve.isZeroAt: one definition for every build, on a bit array given by its index in an array of words, rbaPack consecutive words that it compares with 0 one at a time, for the tests of the builds on the vector type, with vector instructions or by or-ing its words; the test for zero TESTZ also stands for its negation TEST.)

- rp-private.h:242-244: modelled by Ratpoints.Sieve.clearLowBits (deviation of Ratpoints.Sieve.clearLowBits: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is the first Ratpoints.rbaPack words of an array, for the vector type of the build.)

- rp-private.h:245-247: modelled by Ratpoints.Sieve.clearHighBits (deviation of Ratpoints.Sieve.clearHighBits: one definition for every build, the loop of the build with

512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is given by its index in an array of words, `rbaPack` consecutive words, for a pointer to a value of the vector type of the build.)

- `rp-private.h:248-250`: modelled by `Ratpoints.chunk` (deviation of `Ratpoints.chunk`: 16, the value of the default build (`USE_AVX`), set here, where the C code chooses the value by the macros that the compiler is given, each build with vector registers taking 16 and the build with 64-bit bit arrays 1 unless `RATPOINTS_CHUNK` is given.)

- `rp-private.h:252-252`: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- `rp-private.h:255-255`: excluded (other): the include of the SSE2 intrinsics in the build USE_SSE, with 128-bit bit arrays; the word operations of the model need none

- `rp-private.h:257-257`: modelled by `Ratpoints.rbaLength` (deviation of `Ratpoints.rbaLength`: computed from `Ratpoints.rbaPack`, where each build sets `RBA_LENGTH` by a literal, and the build with 64-bit bit arrays by `LONG_LENGTH`, and its logarithm `RBA_SHIFT` likewise (`Ratpoints.rbaShift`); the shifts by `RBA_SHIFT` and by `LONG_SHIFT` are divisions by `rbaLength` and by 64, which round down also for a negative value, as the arithmetic shifts of gcc do, and the masks by `RBA_LENGTH - 1` and by `LONG_MASK` Euclidean remainders.)

- `rp-private.h:258-258`: modelled by `Ratpoints.rbaShift` (deviation of `Ratpoints.rbaShift`: the value of the default build, set here, where each build sets it by a literal, and the build with 64-bit bit arrays by `LONG_SHIFT`; the model shifts by none of them: the shifts by `RBA_SHIFT` are divisions by `Ratpoints.rbaLength`.)

- `rp-private.h:259-260`: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- `rp-private.h:261-261`: modelled by `Ratpoints.Sieve.andRows`, `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRows`: the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`, `sift.c:954`); the moduli of the stage with their rows and their reciprocals as two lists, for the first `sp1` sieve specs and the array `magics` from position `0` on; the value `-w_low - offset` written out in both reductions, for the `long a`, which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_neg_sub`); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the `long`s; a Boolean for the `int small`; no output under `DEBUG`.) (deviation of `Ratpoints.Sieve.andRowsChunked`: no moduli tested by the size of the local array of the sieve specs, for `sp1 == 0`; the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies

into each bit array (bits64, sift.c:519); the end of the whole chunks the index full, for the word number w_high_full, which is also the index of the first bit array of the tail, for the pointer surv after the chunks; natural numbers and integers for the longs.)

- rp-private.h:262-262: modelled by Ratpoints.Sieve.scanPiece (deviation of Ratpoints.Sieve.scanPiece: indices for the pointers into the survivors, and a scan that holds its position as an index, where ratpoints recovers the word number from a pointer difference; the copy nums of the bit array a vector of Ratpoints.rbaPack words; the sentinel ~zero of every build as Ratpoints.rbaPack words ~~~0, written one at a time; natural numbers and integers for the longs; the counters cnt passed and returned, for the field n_arrays of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each bit array, where the C code returns from _ratpoints_sift0 within the walk through the set bits of a word (Ratpoints.Sieve.walkWord).)

- rp-private.h:263-263: modelled by Ratpoints.Sieve.andBitArray (deviation of Ratpoints.Sieve.andBitArray: one definition for every build, on bit arrays given by their indices in arrays of words, each bit array rbaPack consecutive words, for the & on the vector type of the build.)

- rp-private.h:264-266: modelled by Ratpoints.Sieve.siftBitArrayWhole (deviation of Ratpoints.Sieve.siftBitArrayWhole: the stride, the first numerator and the steps as natural numbers and integers, the powers 2 ^ k for the shifts 1L << k, RBA_LENGTH << k and LONG_LENGTH << k, the last two of which stay ints, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below 2 ^ 63, stay longs (Ratpoints.Sieve.NumClass.fitsLong_numerator); the words of the bit array read as entries of the vector, for EXT0 and EXT; natural numbers and integers for the longs; the delivery r passed and returned, for the pointers info and wk->quit and the count total of _ratpoints_sift0, and the flag quit tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from _ratpoints_sift0 within the walk (Ratpoints.Sieve.walkWord); no counts under RP_PHASE_COUNTS and no stop after a phase (RP_STOP_AFTER).)

- rp-private.h:267-267: modelled by Ratpoints.Sieve.isZeroAt (deviation of Ratpoints.Sieve.isZeroAt: one definition for every build, on a bit array given by its index in an array of words, rbaPack consecutive words that it compares with 0 one at a time, for the tests of the builds on the vector type, with vector instructions or by or-ing its words; the test for zero TESTZ also stands for its negation TEST.)

- rp-private.h:268-270: modelled by Ratpoints.Sieve.clearLowBits (deviation of Ratpoints.Sieve.clearLowBits: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is the first Ratpoints.rbaPack words of an array, for the vector type of the build.)

- rp-private.h:271-273: modelled by Ratpoints.Sieve.clearHighBits (deviation of Ratpoints.Sieve.clearHighBits: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is given by its index in an array of words, rbaPack consecutive words, for a pointer to a value of the vector type of the build.)

- rp-private.h:274-276: modelled by `Ratpoints.chunk` (deviation of `Ratpoints.chunk`: 16, the value of the default build (`USE_AVX`), set here, where the C code chooses the value by the macros that the compiler is given, each build with vector registers taking 16 and the build with 64-bit bit arrays 1 unless `RATPOINTS_CHUNK` is given.)

- rp-private.h:278-278: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- rp-private.h:281-281: modelled by `Ratpoints.rbaLength` (deviation of `Ratpoints.rbaLength`: computed from `Ratpoints.rbaPack`, where each build sets `RBA_LENGTH` by a literal, and the build with 64-bit bit arrays by `LONG_LENGTH`, and its logarithm `RBA_SHIFT` likewise (`Ratpoints.rbaShift`); the shifts by `RBA_SHIFT` and by `LONG_SHIFT` are divisions by `rbaLength` and by 64, which round down also for a negative value, as the arithmetic shifts of gcc do, and the masks by `RBA_LENGTH - 1` and by `LONG_MASK` Euclidean remainders.)

- rp-private.h:282-282: modelled by `Ratpoints.rbaShift` (deviation of `Ratpoints.rbaShift`: the value of the default build, set here, where each build sets it by a literal, and the build with 64-bit bit arrays by `LONG_SHIFT`; the model shifts by none of them: the shifts by `RBA_SHIFT` are divisions by `Ratpoints.rbaLength`.)

- rp-private.h:283-284: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- rp-private.h:285-285: modelled by `Ratpoints.Sieve.andRows`, `Ratpoints.Sieve.andRowsChunked` (deviation of `Ratpoints.Sieve.andRows`: the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`, `sift.c:954`); the moduli of the stage with their rows and their reciprocals as two lists, for the first `sp1` sieve specs and the array `magics` from position `0` on; the value `-w_low - offset` written out in both reductions, for the `long a`, which stays a `long` where the helper `mod` reduces it outside `small` (`Ratpoints.Sieve.fitsLong_neg_sub`); indices for the pointers into the rows and into the survivors, and the number of whole periods and the rest computed before their loops; natural numbers and integers for the `long`s; a Boolean for the `int small`; no output under `DEBUG`.) (deviation of `Ratpoints.Sieve.andRowsChunked`: no moduli tested by the size of the local array of the sieve specs, for `sp1 == 0`; the bit arrays set to the class word by a loop that counts up, bit array `n - 1 - i` at the step `i`, for the loop that counts down from the last; the class word written into each word of a bit array, for the vector `RBA(packed)` that `get_2adic_info` makes once per class (`find_points.c:1098`) and `_ratpoints_sift0` copies into each bit array (`bits64`, `sift.c:519`); the end of the whole chunks the index `full`, for the word number `w_high_full`, which is also the index of the first bit array of the tail, for the pointer `surv` after the chunks; natural numbers and integers for the `long`s.)

- rp-private.h:286-286: modelled by `Ratpoints.Sieve.scanPiece` (deviation of `Ratpoints.Sieve.scanPiece`: indices for the pointers into the survivors, and a scan that holds its position as an index, where `ratpoints` recovers the word number from a pointer difference; the copy `nums` of the

bit array a vector of `Ratpoints.rbaPack` words; the sentinel `~zero` of every build as `Ratpoints.rbaPack` words `~~~0`, written one at a time; natural numbers and integers for the `long`s; the counters `cnt` passed and returned, for the field `n_arrays` of the sieving thread, which counts one up for each bit array that is not zero (line 1092); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each bit array, where the C code returns from `_ratpoints_sift0` within the walk through the set bits of a word (`Ratpoints.Sieve.walkWord`).)

- rp-private.h:287-287: modelled by `Ratpoints.Sieve.andBitArray` (deviation of `Ratpoints.Sieve.andBitArray`: one definition for every build, on bit arrays given by their indices in arrays of words, each bit array `rbaPack` consecutive words, for the `&` on the vector type of the build.)

- rp-private.h:288-289: modelled by `Ratpoints.Sieve.siftBitArrayWhole` (deviation of `Ratpoints.Sieve.siftBitArrayWhole`: the stride, the first numerator and the steps as natural numbers and integers, the powers `2 ^ k` for the shifts `1L << k`, `RBA_LENGTH << k` and `LONG_LENGTH << k`, the last two of which stay `int`s, while the first numerators of the words, the numerators of bits of the bit arrays of a height bound below `2 ^ 63`, stay `long`s (`Ratpoints.Sieve.NumClass.fitsLong_numerator`); the words of the bit array read as entries of the vector, for `EXT0` and `EXT`; natural numbers and integers for the `long`s; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` of `_ratpoints_sift0`, and the flag `quit` tested after each walk through the set bits of a word and the rest of the bit array left at a stop, where the C code returns from `_ratpoints_sift0` within the walk (`Ratpoints.Sieve.walkWord`); no counts under `RP_PHASE_COUNTS` and no stop after a phase (`RP_STOP_AFTER`).)

- rp-private.h:290-290: modelled by `Ratpoints.Sieve.isZeroAt` (deviation of `Ratpoints.Sieve.isZeroAt`: one definition for every build, on a bit array given by its index in an array of words, `rbaPack` consecutive words that it compares with `0` one at a time, for the tests of the builds on the vector type, with vector instructions or by or-ing its words; the test for zero `TESTZ` also stands for its negation `TEST`.)

- rp-private.h:291-291: modelled by `Ratpoints.Sieve.clearLowBits` (deviation of `Ratpoints.Sieve.clearLowBits`: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is the first `Ratpoints.rbaPack` words of an array, for the vector type of the build.)

- rp-private.h:292-292: modelled by `Ratpoints.Sieve.clearHighBits` (deviation of `Ratpoints.Sieve.clearHighBits`: one definition for every build, the loop of the build with 512-bit bit arrays, of which the macros of the builds with 256-bit and 128-bit bit arrays are unrolled cases and that of the build with 64-bit bit arrays the case of one word; the bit array is given by its index in an array of words, `rbaPack` consecutive words, for a pointer to a value of the vector type of the build.)

- rp-private.h:293-295: modelled by `Ratpoints.chunk` (deviation of `Ratpoints.chunk`: 16, the value of the default build (`USE_AVX`), set here, where the C code chooses the value by the macros that the compiler is given, each build with vector registers taking 16 and the build with 64-bit bit arrays 1 unless `RATPOINTS_CHUNK` is given.)

- rp-private.h:300-300: modelled by `Ratpoints.rbaPack` (deviation of `Ratpoints.rbaPack`: the value of the default build, set here, where the C code chooses the build by the macros

that the compiler is given; the vector type of a bit array becomes `rbaPack` consecutive words of an array of `UInt64`.)

- rp-private.h:306-308: modelled by `Ratpoints.Sieve.isZeroAt` (deviation of `Ratpoints.Sieve.isZeroAt`: one definition for every build, on a bit array given by its index in an array of words, `rbaPack` consecutive words that it compares with `0` one at a time, for the tests of the builds on the vector type, with vector instructions or by or-ing its words; the test for zero `TESTZ` also stands for its negation `TEST`.)
- rp-private.h:311-318: excluded (instrumentation): WIDTH and PRINT_RBA, which print words and bit arrays in hexadecimal in the output for debugging (under DEBUG)
- rp-private.h:321-323: modelled by `Ratpoints.maxBitsInPrime` (deviation of `Ratpoints.maxBitsInPrime`: 8, the value of the default build, which the Makefile of `ratpoints` passes to the compiler (its variable `PRIME_SIZE`), for the value 7 that the header takes when it is given none.)
- rp-private.h:325-361: modelled by `Ratpoints.maxPrimeEven`, `Ratpoints.numPrimes` (deviation of `Ratpoints.maxPrimeEven`: computed from `Ratpoints.maxBitsInPrime`, where the header lists the values for 5 to 10 bits, the powers of 2 that the computation gives, and takes 7 bits for any other value.) (deviation of `Ratpoints.numPrimes`: computed from `Ratpoints.maxBitsInPrime`, where the header lists the values for 5 to 10 bits, the numbers that the computation gives, and takes 7 bits for any other value; the largest such prime, `RATPOINTS_MAX_PRIME`, has no counterpart; the power of 2 above it is `Ratpoints.maxPrimeEven`.)
- rp-private.h:378-379: modelled by `Ratpoints.Sieve.SieveSpec` (deviation of `Ratpoints.Sieve.SieveSpec`: the row for the pointer `ptr` to it, the empty array for the null pointer of a row not built yet (`Ratpoints.Sieve.setRows`, `Ratpoints.Sieve.buildRows`), and indices into the row for the pointers `start` and `end` into it, the end named `stop` (`end` is a keyword of Lean); natural numbers for the `long`s `p` and `offset`, the offset below `2 ^ 32` (`Ratpoints.Sieve.SieveEntry.PackedFor.lt_two_pow_32`).)
- rp-private.h:387-387: modelled by `Ratpoints.rowBias` (deviation of `Ratpoints.rowBias`: the check at compile time that `2 RP_ROW_BIAS - 1 ≤ RP_MULMOD_LIMIT`, an array of negative size otherwise, a theorem checked by `decide` (`Ratpoints.two_mul_rowBias_sub_one_le_mulModLimit`).)
- rp-private.h:404-408: modelled by `Ratpoints.Sieve.mulHi`, `Ratpoints.Sieve.mulMod` (deviation of `Ratpoints.Sieve.mulHi`: the high half of the product from the products of the 32-bit halves, for the product of type `__uint128_t` that gcc provides and Lean core does not.) (deviation of `Ratpoints.Sieve.mulMod`: natural numbers for the `unsigned long` reciprocal, the `long`s `u` and `p`, and the `long` result, which is below `p`; the arguments converted to `UInt64`, as the macro converts `u` and `p` to `unsigned long`s, so that the product `m u` wraps as the product of `unsigned long`s does; the argument `u` a natural number, for the `long` that the C code passes non-negative; the high half of the 128-bit product by `Ratpoints.Sieve.mulHi`; the form with the 128-bit product, which the modelled builds compile (the compiler defines `__SIZEOF_INT128__`, and neither `RP_MULMOD_DIVIDE` nor `RP_NO_INT128` is set), for the division `u % p` of the other builds, which gives the same remainder where the multiplication is exact.)
- rp-private.h:409-410: modelled by `Ratpoints.Sieve.mulDiv`, `Ratpoints.Sieve.mulHi` (deviation of `Ratpoints.Sieve.mulDiv`: natural numbers for the `unsigned long` reciprocal, the `long u` and the `long` result, the arguments converted to `UInt64`, as the macro converts `u` to an `unsigned long`; the high half of the 128-bit product by `Ratpoints.Sieve.mulHi`; no argument `p`, which that form does not read; the form with the 128-bit product, which the modelled builds compile, for the division `u / p` of the other builds, which gives the

same quotient where the multiplication is exact.) (deviation of `Ratpoints.Sieve.mulHi`: the high half of the product from the products of the 32-bit halves, for the product of type `__uint128_t` that gcc provides and Lean core does not.)

- rp-private.h:411-411: modelled by `Ratpoints.Sieve.mulMod` (deviation of `Ratpoints.Sieve.mulMod`: natural numbers for the `unsigned long` reciprocal, the `long`s `u` and `p`, and the `long` result, which is below `p`; the arguments converted to `UInt64`, as the macro converts `u` and `p` to `unsigned long`s, so that the product `m u` wraps as the product of `unsigned long`s does; the argument `u` a natural number, for the `long` that the C code passes non-negative; the high half of the 128-bit product by `Ratpoints.Sieve.mulHi`; the form with the 128-bit product, which the modelled builds compile (the compiler defines `__SIZEOF_INT128__`, and neither `RP_MULMOD_DIVIDE` nor `RP_NO_INT128` is set), for the division `u % p` of the other builds, which gives the same remainder where the multiplication is exact.)

- rp-private.h:413-414: modelled by `Ratpoints.Sieve.mulMod` (deviation of `Ratpoints.Sieve.mulMod`: natural numbers for the `unsigned long` reciprocal, the `long`s `u` and `p`, and the `long` result, which is below `p`; the arguments converted to `UInt64`, as the macro converts `u` and `p` to `unsigned long`s, so that the product `m u` wraps as the product of `unsigned long`s does; the argument `u` a natural number, for the `long` that the C code passes non-negative; the high half of the 128-bit product by `Ratpoints.Sieve.mulHi`; the form with the 128-bit product, which the modelled builds compile (the compiler defines `__SIZEOF_INT128__`, and neither `RP_MULMOD_DIVIDE` nor `RP_NO_INT128` is set), for the division `u % p` of the other builds, which gives the same remainder where the multiplication is exact.)

- rp-private.h:415-416: modelled by `Ratpoints.Sieve.mulDiv` (deviation of `Ratpoints.Sieve.mulDiv`: natural numbers for the `unsigned long` reciprocal, the `long` `u` and the `long` result, the arguments converted to `UInt64`, as the macro converts `u` to an `unsigned long`; the high half of the 128-bit product by `Ratpoints.Sieve.mulHi`; no argument `p`, which that form does not read; the form with the 128-bit product, which the modelled builds compile, for the division `u / p` of the other builds, which gives the same quotient where the multiplication is exact.)

- rp-private.h:417-417: modelled by `Ratpoints.Sieve.mulMod` (deviation of `Ratpoints.Sieve.mulMod`: natural numbers for the `unsigned long` reciprocal, the `long`s `u` and `p`, and the `long` result, which is below `p`; the arguments converted to `UInt64`, as the macro converts `u` and `p` to `unsigned long`s, so that the product `m u` wraps as the product of `unsigned long`s does; the argument `u` a natural number, for the `long` that the C code passes non-negative; the high half of the 128-bit product by `Ratpoints.Sieve.mulHi`; the form with the 128-bit product, which the modelled builds compile (the compiler defines `__SIZEOF_INT128__`, and neither `RP_MULMOD_DIVIDE` nor `RP_NO_INT128` is set), for the division `u % p` of the other builds, which gives the same remainder where the multiplication is exact.)

- rp-private.h:421-421: modelled by `Ratpoints.mulModLimit`

- rp-private.h:426-427: modelled by `Ratpoints.rowBias` (deviation of `Ratpoints.rowBias`: the check at compile time that `2 RP_ROW_BIAS - 1 ≤ RP_MULMOD_LIMIT`, an array of negative size otherwise, a theorem checked by `decide` (`Ratpoints.two_mul_rowBias_sub_one_le_mulModLimit`).)

- rp-private.h:429-444: modelled by `Ratpoints.Sieve.ctz` (deviation of `Ratpoints.Sieve.ctz`: the loop for a compiler without the builtin, where gcc and clang take `__builtin_ctzl`.)

- rp-private.h:450-456: excluded (other): RP_ALWAYS_INLINE and RP_NOINLINE, the inlining attributes that sift.c and find_points.c rely on for speed, empty on a compiler without them; they carry no meaning

- rp-private.h:465-466: modelled by `Ratpoints.Sieve.CheckSpec` (deviation of `Ratpoints.Sieve.CheckSpec`: natural numbers for the `longs` and the `unsigned long`; the table an array of Booleans, for the pointer to the integers `is_f_square` of the sieve entry of the prime.)

- rp-private.h:472-472: modelled by `Ratpoints.stage3Limit` (deviation of `Ratpoints.stage3Limit`: a natural number for the `double` `4294967296.0`, which the C code compares with the `double` product `2 p H`.)

- rp-private.h:476-476: modelled by `Ratpoints.numStrides`

- rp-private.h:494-495: modelled by `Ratpoints.Sieve.NumClass` (deviation of `Ratpoints.Sieve.NumClass`: one word for the bit array `bits`, the vector `RBA(packed)` of `RBA_PACK` copies of the word, which the first stage writes into every word of a bit array; natural numbers for the `longs` `k` and `a0`; the position of the row of offsets in the array of the offsets as an option, for the pointer `offset` into that array, none for `NULL`.)

- rp-private.h:497-498: modelled by `Ratpoints.Sieve.RowInit` (deviation of `Ratpoints.Sieve.RowInit`: the kind of the builder, for the pointer to the function of the type `ratpoints_init_fun`, which for a prime points to the function that the build generates for it; the call through the pointer a match on the kind (`Ratpoints.Sieve.SieveEntry.initRow`, `Ratpoints.Sieve.initFactorRow`).)

- rp-private.h:520-520: modelled by `Ratpoints.maxFactors`

- rp-private.h:523-523: modelled by `Ratpoints.modWords` (deviation of `Ratpoints.modWords`: `64` for `LONG_LENGTH`.)

- rp-private.h:531-534: modelled by `Ratpoints.Sieve.Power` (deviation of `Ratpoints.Sieve.Power`: an array of `m` natural numbers for the `unsigned short` inverses, for an array sized by `RATPOINTS_MAX_PRIME_EVEN`; no count `np`, which `examine_power` stores (`find_points.c:2304`) and nothing reads.)

- rp-private.h:542-555: modelled by `Ratpoints.Sieve.Entry`, `Ratpoints.Sieve.SieveEntry` (deviation of `Ratpoints.Sieve.Entry`: only the fields that the rows of a prime read; the rows of the residues, which `ratpoints` keeps in the field `sieve`, that of `0` from the build and the others made by `init` when a denominator first needs them, are kept in the table of rows of the prime in the store of the run (`Ratpoints.Sieve.EntryRef.prime`, `Ratpoints.Sieve.primeRows`), from which the run reads them and into which it builds them (`Ratpoints.Sieve.setRows`, `Ratpoints.Sieve.buildRows`); the fields `init`, `magic`, `nf`, `factor` and `pw` are those of the sieve entry `Ratpoints.Sieve.SieveEntry`, of which the prime is the single factor, and so are `bias`, `rbainv`, `dinv` and the density `r`.) (deviation of `Ratpoints.Sieve.SieveEntry`: the kinds of modulus as the list of the factors: one prime factor for a prime, whose entry has `nf = 0` and the tables `is_f_square` and `inverses` of its own; one prime-power factor for a prime power, whose entry has `nf = 1` and its tables `pw`; and two or more for a product, whose entry has `nf ≥ 2` and points to the entries `factor` of its factors; no bound `RP_MAX_FACTORS` on the number of factors, which the choice of the moduli keeps when it offers a composite modulus (`Ratpoints.takeFactor`); the rows of the residues, which `ratpoints` keeps in the array `sieve` of the entry, kept in the store of the run (`Ratpoints.Sieve.RowStore`) in the table of rows of the reference `ref`; the reference `ref` to

the entry whose table of rows holds its rows (`Ratpoints.Sieve.EntryRef`), for the pointer to the entry: a prime or a prime power that of its one factor, so that the modulus and the products with it as a factor read one table of rows, as they read one entry in the C code, and a product its own; the kind of its builder `init` (`Ratpoints.Sieve.RowInit`), for the pointer to the function; a natural number for the `unsigned long` reciprocal `magic`, and natural numbers for the `long`s `bias` and `rbainv`; the inverses `dinv` an array, for the array of `RP_NUM_STRIDES` `long`s, which the row offsets of the packing fill (`Ratpoints.Sieve.classOffsets`) with an entry for each stride up to the largest in use, where the entries of the C code beyond it stay unset and unread; the density `r`, which the correction of the moduli during the run reads (`Ratpoints.Sieve.adaptPrimes`), a `Float` for the `double`, and the count `nf` of the factors, an integer for the `long`, `0` for a prime, `1` for a prime power and the number of factors for a product, kept with the list of the factors.)

- rp-private.h:574-574: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `long`s; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)
- rp-private.h:575-575: modelled by `Ratpoints.Sieve.siftPiece` (deviation of `Ratpoints.Sieve.siftPiece`: the arm of the first stage chosen by a test on the constant `Ratpoints.chunk`, for the conditional compilation `#if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16)` (`sift.c:452`, `sift.c:575`, `sift.c:943`), which compiles one arm only, and `tail_leg` with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which

every piece reuses, with one bit array more for the alignment, and whose words `malloc` leaves uninitialized; the piece by its first word number `w0` and its number `n` of bit arrays, for `w_low` and `w_high`, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list `specs` and the end `sp1` of the first stage, for the sieve specs `sieves` and the number `sp1` that the sieving thread keeps, the reciprocals of their moduli as the list `mgs`, for the array `magics` of the arguments, and the thunk `checks`, for the table `checks` of the third stage and its length `nchecks = sp3 - sp2`; the class `cl` as a value, for the pointer `cls`; integers for the `long`s of the test `small`, which stay `long`s, and a Boolean for its `int`; the counters `cnt` passed and returned, for the field `n_words` of the sieving thread, which counts the `n Ratpoints.rbaPack` words of the piece (line 548), the product of `UInt64`s as the C code multiplies the `unsigned long` by `RBA_PACK`; the thunk `w` of the products of the check as an argument, where `_ratpoints_check_point` computes the products when the first survivor of `b` reaches it; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_sift0` keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (`Ratpoints.Sieve.scanPiece`); natural numbers and integers for the `long`s; no measurements of the phases (`RP_TIC`, `RP_TOC`, `RP_PHASE_TIMING`) and no stop after one of them (`RP_STOP_AFTER`).)

- rp-private.h:576-576: modelled by `Ratpoints.Sieve.checkPoint` (deviation of `Ratpoints.Sieve.checkPoint`: the products `w` as a thunk, which the sieve of one denominator makes for each denominator (`Ratpoints.Sieve.sift`) and which is forced when the first numerator of `b` reaches the check, for the array `work` of the sieving thread, which `worker_init` gives it, and its flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the degree `d` and the flags `fl` as arguments, handed down from `Ratpoints.Sieve.sieveRun` through the components that call the check, for the degree and the bits `RATPOINTS_REVERSED`, `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the flags of the search, which `_ratpoints_check_point` reads through its worker (`args->degree`, `args->flags`); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_check_point` keeps from zero and returns, and the three branches of the map of the reversal at each call one definition (`Ratpoints.Delivery.callMapped`); the square root and the test of the remainder by `Nat.sqrt` on `v`, for `mpz_sqrtrem`; integers for the `long`s `a` and `b` (the negations of the map stay `long`s by `Ratpoints.fitsLong_neg`); no measurements of the time of the check and of its products (`RP_CHECK`, `RP_BC_TIC`, `RP_BC_TOC` under `RP_PHASE_TIMING`), and no output under `DEBUG`.)

- rp-private.h:577-578: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_rat-`

`points_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `longs`; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `rp-private.h:579-581`: modelled by `Ratpoints.Sieve.Counters` (deviation of `Ratpoints.Sieve.Counters`: a structure of its own, which the components of the sieve that count pass and return beside the delivery, for the fields of the sieving thread `rp_worker`, which `run_block` clears for each block (`find_points.c:3914-3915`, `Ratpoints.Sieve.runBlock`); `UInt64` for the `unsigned longs`, wrapping as they wrap.)

- `rp-private.h:582-582`: modelled by `Ratpoints.Sieve.runBlock` (deviation of `Ratpoints.Sieve.runBlock`: the decision's entries and reciprocals cut as lists for each block, for the numbers `sp2` and `sp3` that the sieving thread keeps and through which `sift` and `fill_bp_list` read the arrays `sieve_list` and `magics`; the report returned with the delivery and the store, for the fields of the sieving thread that `worker_stat` copies into it after the block; the counters and the index of the decision, which `worker_init` clears before the run (lines 3127-3128), set only here, for each block (lines 3911 and 3914-3915), before they are read; the arguments of the run passed explicitly, for the fields of the run and of the arguments that `run_block` reads; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `run_block` keeps from zero and returns; the array `bp_list` of `sp3_max` entries on the stack, of which `fill_bp_list` fills the first `sp2`, a list made by `Ratpoints.Sieve.siftDen`; natural numbers for the `longs`.)

- `rp-private.h:583-583`: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries

`sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `long`s; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `rp-private.h:584-584`: excluded (threads): the pool, the slot of its block and the flag to abandon it of a thread of the pool; the calling thread, which the model is, has none

- `rp-private.h:585-585`: modelled by `Ratpoints.Sieve.sift` (deviation of `Ratpoints.Sieve.sift`: the height bound converted to `Int64`, which is exact for `H < 2 ^ 63`, and the denominator too, and the bits back to integers; the bounds of the search intervals an array, for the struct `rp_intervals` with its pointers `bd` and `end`, which `sift` reaches through the field `iv` of the sieving thread, which `worker_init` clears (line 3130) and `run_block` sets for each block (line 3913; here `Ratpoints.Sieve.runBlock`, which passes the array of the run on); one call of `Ratpoints.Sieve.setRows`, for the two calls with the flag `shared` a constant, between which the pool of threads chooses, which the model does not have; the store of the tables of rows passed and returned, for the arrays `sieve` of the entries of the arguments, which `sift` reaches through `sieve_list` and which keep the rows built for the rest of the run; the end `sp1` of the first stage, the entries `es` and the reciprocals `mgs` of the first two stages, the entries `es3` of the third stage and their reciprocals `mgs3` as arguments, for the numbers `sp1`, `sp2` and `sp3` that the sieving thread keeps (`worker_init`), the entries `sieve_list[0], …, sieve_list[sp2 - 1]`, the array `magics` of the arguments, which `_ratpoints_sift0` reads, and the entries `sieve_list[sp2], …, sieve_list[sp3 - 1]` and the reciprocals `magics[sp2], …, magics[sp3 - 1]`, which `fill_checks` reads, the cut points of the decision of the block (`Ratpoints.Sieve.runBlock`); the residues `bp` and the class `cl` as values, for the pointers `bp_list` and `cls`, and the array `offsets` of the row offsets of the run as an argument, which the C code reaches through the pointer `offset` of the class; the thunk of `Ratpoints.Sieve.fillChecks`, for the table `checks` of the sieving thread, which `fill_checks` fills, and its flag `stage3_filled`, which `sift` clears; the thunk of the products `w`, for the flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the counters `cnt` passed and returned, for those of the sieving thread, the count `sifts` one up at the start (line 3461); the delivery `r` passed and returned, for the pointers `info` and `wk->quit`, which `worker_init` clears, and the count `total` that `sift` keeps from zero and returns; the arguments of the run passed explicitly, for the pointer `args` of the sieving thread (`rp-private.h:574`), which `worker_init` sets and through which `sift` reads them; no pool, slot and stop flag of a thread of the pool, the flag by which the pool

of threads tells it to abandon its block (`pool`, `slot`, `stop`, `rp-private.h:584`, excluded with the threads); natural numbers and integers for the `long`s; no measurements of the phases (`RP_SIFT_TIC`, `RP_SETUP_TIC` and their `TOC`) and no output under `DEBUG`.)

- `rp-private.h:592-592`: modelled by `Ratpoints.Sieve.sieveInitPower` (deviation of `Ratpoints.Sieve.sieveInitPower`: the tables of the prime power and the reference to its entry (`Ratpoints.Sieve.EntryRef`) as arguments, for the sieve entry `se` whose field `pw` points to the tables and into which the C code stores the row, and the store of the tables of rows of the run passed and returned, for the arrays `sieve` of the entries; the pattern a fresh array, which the loop clears, for the array whose words the function leaves unset until it clears them; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have.)

- `rp-private.h:593-593`: modelled by `Ratpoints.Sieve.sieveInitProduct` (deviation of `Ratpoints.Sieve.sieveInitProduct`: the store of the tables of rows passed and returned, for the entries of the factors that the C code reaches through the pointers `factor` and for the entry `se` into which it stores the row; a fresh vector of `(m + chunk - 1) rbaPack` words, for the space that the builder takes from the buffer of the tables (`ba_next`) after the rows of the factors; the row stored plainly, for the release store of `RP_PUBLISH` in a build with threads, which the model does not have; the number of the factors the length of their list, for the count `nf`.)

- `rp-private.h:596-597`: modelled by `Ratpoints.Sieve.checkPoint` (deviation of `Ratpoints.Sieve.checkPoint`: the products `w` as a thunk, which the sieve of one denominator makes for each denominator (`Ratpoints.Sieve.sift`) and which is forced when the first numerator of `b` reaches the check, for the array `work` of the sieving thread, which `worker_init` gives it, and its flag `compute_bc`, which `sift` sets and `_ratpoints_check_point` clears when it computes the products; the degree `d` and the flags `fl` as arguments, handed down from `Ratpoints.Sieve.sieveRun` through the components that call the check, for the degree and the bits `RATPOINTS_REVERSED`, `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` of the flags of the search, which `_ratpoints_check_point` reads through its worker (`args->degree`, `args->flags`); the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_check_point` keeps from zero and returns, and the three branches of the map of the reversal at each call one definition (`Ratpoints.Delivery.callMapped`); the square root and the test of the remainder by `Nat.sqrt` on `v`, for `mpz_sqrtrem`; integers for the `long`s `a` and `b` (the negations of the map stay `long`s by `Ratpoints.fitsLong_neg`); no measurements of the time of the check and of its products (`RP_CHECK`, `RP_BC_TIC`, `RP_BC_TOC` under `RP_PHASE_TIMING`), and no output under `DEBUG`.)

- `rp-private.h:610-614`: modelled by `Ratpoints.Sieve.siftPiece` (deviation of `Ratpoints.Sieve.siftPiece`: the arm of the first stage chosen by a test on the constant `Ratpoints.chunk`, for the conditional compilation `#if (RATPOINTS_CHUNK > 1) && (RATPOINTS_CHUNK <= 16)` (`sift.c:452`, `sift.c:575`, `sift.c:943`), which compiles one arm only, and `tail_leg` with the chunked one; a fresh array of zeros for each piece, for the survivors array of the sieving thread, which every piece reuses, with one bit array more for the alignment, and whose words `malloc` leaves uninitialized; the piece by its first word number `w0` and its number `n` of bit arrays, for `w_low` and `w_high`, and an index for the pointer to its last bit array; the moduli of the first two stages with their rows and offsets as the list `specs` and the end `sp1` of the first stage, for the sieve specs `sieves` and the number `sp1` that the sieving thread keeps, the reciprocals of their moduli as the list `mgs`, for the array `magics` of the arguments, and the thunk `checks`, for the table `checks` of the third stage and its length `nchecks = sp3 - sp2`;

the class `cl` as a value, for the pointer `cls`; integers for the `long`s of the test `small`, which stay `long`s, and a Boolean for its `int`; the counters `cnt` passed and returned, for the field `n_words` of the sieving thread, which counts the `n Ratpoints.rbaPack` words of the piece (line 548), the product of `UInt64`s as the C code multiplies the `unsigned long` by `RBA_PACK`; the thunk `w` of the products of the check as an argument, where `_ratpoints_check_point` computes the products when the first survivor of `b` reaches it; the delivery `r` passed and returned, for the pointers `info` and `wk->quit` and the count `total` that `_ratpoints_sift0` keeps from zero and returns, and its returns at a stop within the walk through the set bits of a word passed on by the scan (`Ratpoints.Sieve.scanPiece`); natural numbers and integers for the `long`s; no measurements of the phases (`RP_TIC`, `RP_TOC`, `RP_PHASE_TIMING`) and no stop after one of them (`RP_STOP_AFTER`).)

- `rp-private.h:617-617`: excluded (interface): the prototype of _ratpoints_compute_sturm, which find_points.c calls; the model has no headers (Ratpoints.Sturm.computeSturm)

`ratpoints.h`

- `ratpoints.h:31-31`: excluded (other): the include of gmp.h, the header of GMP; the model has no headers
- `ratpoints.h:33-33`: modelled by `Ratpoints.maxDegree`
- `ratpoints.h:42-42`: excluded (other): RATPOINTS_DEFAULT_SP1, the fixed end of the first stage of the versions 2.x, which the C code reads nowhere: the library chooses the end of the first stage from the curve (sieving_info)
- `ratpoints.h:43-43`: excluded (other): RATPOINTS_DEFAULT_SP2, the fixed end of the second stage of the versions 2.x, which the C code reads nowhere
- `ratpoints.h:92-94`: modelled by `Ratpoints.survivorsPerWord`
- `ratpoints.h:95-97`: modelled by `Ratpoints.sp2Extra`
- `ratpoints.h:120-122`: modelled by `Ratpoints.sp2U0`
- `ratpoints.h:150-152`: modelled by `Ratpoints.sp3PerSurvivor`
- `ratpoints.h:153-155`: modelled by `Ratpoints.sp3PerDenom`
- `ratpoints.h:164-166`: modelled by `Ratpoints.sp3Coprime`
- `ratpoints.h:192-194`: modelled by `Ratpoints.costTable`
- `ratpoints.h:210-212`: modelled by `Ratpoints.compositeMax`
- `ratpoints.h:213-218`: modelled by `Ratpoints.costBp`
- `ratpoints.h:224-226`: modelled by `Ratpoints.costSetup`
- `ratpoints.h:241-243`: modelled by `Ratpoints.costCall`
- `ratpoints.h:258-260`: modelled by `Ratpoints.costLine`
- `ratpoints.h:261-263`: modelled by `Ratpoints.costPhase2`

- ratpoints.h:270-272: modelled by Ratpoints.costSurvivor
- ratpoints.h:280-282: modelled by Ratpoints.costCheck
- ratpoints.h:312-314: modelled by Ratpoints.checkStep
- ratpoints.h:315-317: modelled by Ratpoints.checkLimb
- ratpoints.h:318-320: modelled by Ratpoints.checkCall
- ratpoints.h:321-323: modelled by Ratpoints.checkRoot
- ratpoints.h:329-331: modelled by Ratpoints.checkReference
- ratpoints.h:333-340: modelled by Ratpoints.defaultNumPrimes
- ratpoints.h:341-343: modelled by Ratpoints.defaultSturm
- ratpoints.h:345-352: modelled by Ratpoints.defaultMaxForbidden
- ratpoints.h:354-356: modelled by Ratpoints.defaultArraySize
- ratpoints.h:359-359: modelled by Ratpoints.Options (deviation of Ratpoints.Options: a structure for the fields of ratpoints_args that hold these options: Booleans for the bits RATPOINTS_NO_CHECK, RATPOINTS_NO_Y and RATPOINTS_NO_REVERSE of the flags, noCheck and noY true when their bits are set, reverse true when RATPOINTS_NO_REVERSE is clear, so that all three defaults are those of the cleared flags (main.c, line 352); of the other bits of the flags, which main.c clears with these, the Jacobi switch is that of the options of the sieve (Ratpoints.SieveOptions.jacobi), and RATPOINTS_VERBOSE is the print channel of the search (Ratpoints.findPointsVerbose), which the program makes from Ratpoints.Cli.Output.verbose; an integer for args->sturm; a list of pairs of doubles (Ratpoints.Double) for the array domain of the type ratpoints_interval with the number num_inter of its intervals, the empty list for num_inter = 0, so that no number of intervals is negative and the room in the array for the intervals of the Sturm step is no concern of the caller; integers for the longs b_low and b_high; of the fields that share their lines in ratpoints.h, sp1, sp2, num_primes and max_forbidden are read by the options of the sieve (Ratpoints.SieveOptions), and sp3 is the end of the third stage of the choice (Ratpoints.Params.sp3), corrected during the run (Ratpoints.Sieve.Correction).)
- ratpoints.h:362-362: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0

stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `ratpoints.h:363-364`: modelled by `Ratpoints.Options` (deviation of `Ratpoints.Options`: a structure for the fields of `ratpoints_args` that hold these options: Booleans for the bits `RATPOINTS_NO_CHECK`, `RATPOINTS_NO_Y` and `RATPOINTS_NO_REVERSE` of the flags, `noCheck` and `noY` true when their bits are set, `reverse` true when `RATPOINTS_NO_REVERSE` is clear, so that all three defaults are those of the cleared flags (`main.c`, line 352); of the other bits of the flags, which `main.c` clears with these, the Jacobi switch is that of the options of the sieve (`Ratpoints.SieveOptions.jacobi`), and `RATPOINTS_VERBOSE` is the print channel of the search (`Ratpoints.findPointsVerbose`), which the program makes from `Ratpoints.Cli.Output.verbose`; an integer for `args->sturm`; a list of pairs of doubles (`Ratpoints.Double`) for the array `domain` of the type `ratpoints_interval` with the number `num_inter` of its intervals, the empty list for `num_inter = 0`, so that no number of intervals is negative and the room in the array for the intervals of the Sturm step is no concern of the caller; integers for the `long`s `b_low` and `b_high`; of the fields that share their lines in `ratpoints.h`, `sp1`, `sp2`, `num_primes` and `max_forbidden` are read by the options of the sieve (`Ratpoints.SieveOptions`), and `sp3` is the end of the third stage of the choice (`Ratpoints.Params.sp3`), corrected during the run (`Ratpoints.Sieve.Correction`).)

- `ratpoints.h:365-367`: modelled by `Ratpoints.SieveOptions` (deviation of `Ratpoints.SieveOptions`: integers for the `long`s of the fields, doubles by their bits (`Ratpoints.Double`) for the doubles, as the program reads doubles (`Ratpoints.Cli.scanDouble?`), and a Boolean for the bit `RATPOINTS_NO_JACOBI` of the flags, which `main.c` clears with the others; the bit of the reversal is in `Ratpoints.Options`.)

- `ratpoints.h:368-368`: modelled by `Ratpoints.Params` (deviation of `Ratpoints.Params`: the parameters as a structure of their own, for the fields of the arguments that `find_points_work` normalizes and `sieving_info` sets (`sp1`, `sp2`, `sp3`, `max_forbidden`, `array_size`, `run_words`,

`run_denoms`, `adapt_at`, `check_rel`; `sp3_max` as the length of `Ratpoints.SievingInfo.entries`) and the local `pn_lim` of `sieving_info` at the end (`lookedAt`); the Jacobi switch a Boolean (`Ratpoints.Params.jacobi`), true when the bit `RATPOINTS_NO_JACOBI` of the flags is clear, for that bit of `args->flags`, which `main.c` clears with the others; a natural number for the `long` field `array_size` (`Ratpoints.Params.arraySize`), `0` for the default, which the library takes for every value `≤ 0` (`Ratpoints.Params.arraySizeAt`); `Float`s for the doubles and a `UInt64` for the `unsigned long`.)

- `ratpoints.h:369-369`: modelled by `Ratpoints.Options` (deviation of `Ratpoints.Options`: a structure for the fields of `ratpoints_args` that hold these options: Booleans for the bits `RATPOINTS_NO_CHECK`, `RATPOINTS_NO_Y` and `RATPOINTS_NO_REVERSE` of the flags, `noCheck` and `noY` true when their bits are set, `reverse` true when `RATPOINTS_NO_REVERSE` is clear, so that all three defaults are those of the cleared flags (`main.c`, line 352); of the other bits of the flags, which `main.c` clears with these, the Jacobi switch is that of the options of the sieve (`Ratpoints.SieveOptions.jacobi`), and `RATPOINTS_VERBOSE` is the print channel of the search (`Ratpoints.findPointsVerbose`), which the program makes from `Ratpoints.Cli.Output.verbose`; an integer for `args->sturm`; a list of pairs of doubles (`Ratpoints.Double`) for the array `domain` of the type `ratpoints_interval` with the number `num_inter` of its intervals, the empty list for `num_inter = 0`, so that no number of intervals is negative and the room in the array for the intervals of the Sturm step is no concern of the caller; integers for the `long`s `b_low` and `b_high`; of the fields that share their lines in `ratpoints.h`, `sp1`, `sp2`, `num_primes` and `max_forbidden` are read by the options of the sieve (`Ratpoints.SieveOptions`), and `sp3` is the end of the third stage of the choice (`Ratpoints.Params.sp3`), corrected during the run (`Ratpoints.Sieve.Correction`).)

- `ratpoints.h:370-376`: excluded (threads): the number of threads that sieve; the model sieves on the calling thread, with the same output for every number of threads

- `ratpoints.h:377-383`: modelled by `Ratpoints.Outcome` (deviation of `Ratpoints.Outcome`: a structure for what the C code returns and writes back into the arguments: the data of the callback, which the caller has behind its pointer `info`, the count, which `find_points` returns as a `long` that also carries its error codes (`Ratpoints.findPointsWith` returns the refusals apart), Booleans for three bits of `args->flags`, and the search intervals as a list, for the array `args->domain` and the number `args->num_inter` that the search writes back and the program reads after it; nothing else written back: neither the reversed coefficients and the degree (`args->cof`, `args->degree`), which nothing reads after the search; nor the other internal bits of the flags, `RATPOINTS_CHECK_DENOM`, `RATPOINTS_USE_JACOBI` and `RATPOINTS_NO_REVERSE_AUTO`, which the program does not read either, the bits of the input coming back as the caller set them (`find_points.c:4606`), which the program takes from its own options and settings (`Ratpoints.Options`, `Ratpoints.SieveOptions.jacobi`, `Ratpoints.Cli.Output.verbose`); nor `args->sp3`, which `find_points_work` zeroes on entry (line 4551) and leaves as the search leaves it, unlike `sp2`, which it restores (line 4561): the end of the third stage of the last block sifted (line 5163), that of the choice after the returns between the choice and the sieve, where `sp3_used` stays `0`, and `0` after the returns before the choice and after the exits of the choice when the curve has no points modulo a prime (`find_points.c:2614`, `find_points.c:2998`), which come before line 3017 sets it; nor the private fields that `sieving_info` and `adapt_primes` write, `adapt_at`, `run_words`, `run_denoms`, `sp3_max` and `check_rel`, which nothing reads before `sieving_info` sets them again in the next search; natural numbers for the `long`s `sp1_used`, `sp2_used` and `sp3_used`.)

- `ratpoints.h:385-388`: excluded (memory): the private fields of the arguments that hold

memory: the scratch integers of the search with their number (work, work_length), and the buffers of the sieve entries, of the bit arrays of the rows and of the integers of the tables of the primes, each with the pointer to its free space (se_buffer, se_next, ba_buffer, ba_next, int_buffer, int_next), which find_points_init allocates (find_points.c:272-375, excluded as interface); the model computes with Lean's integers and makes the entries, the rows and the tables as fresh values

- ratpoints.h:389-389: modelled by `Ratpoints.SievingInfo` (deviation of `Ratpoints.SievingInfo`: a structure for what `sieving_info` writes into the arguments, the arrays `sieve_list` and `magics` (line 389) and `forb_ba` and `forbidden` (line 391) as lists, for the arrays with their counts, the bit `RATPOINTS_CHECK_DENOM` of the flags as a Boolean and the fields of the result (`Ratpoints.Params`), and into the entries, the arrays `sieve` as the store of the tables of rows; no field for the array `stage3_list` (line 389), the buffer of the table of the third stage that the sieving thread fills for each denominator (`find_points.c:5127`), which the model replaces by the thunk of the checks (`Ratpoints.Sieve.fillChecks`, `Ratpoints.Sieve.sift`).)

- ratpoints.h:390-390: modelled by `Ratpoints.Setup.setupUs1` (deviation of `Ratpoints.Setup.setupUs1`: the leading coefficient read from the list, for the integer `work[0]` that the caller sets to `|c_d|` (lines 4911-4912); lists, for the arrays `den_info` and `divisors` of the arguments (`ratpoints.h:390`) with their terminating zeros; `none` for a factorization that is not complete or too many divisors, where `setup_us1` leaves the flag `RATPOINTS_USE_SQUARES1` clear, and the lists returned, where it sets the flag; no output under `DEBUG`.)

- ratpoints.h:391-391: modelled by `Ratpoints.SievingInfo` (deviation of `Ratpoints.SievingInfo`: a structure for what `sieving_info` writes into the arguments, the arrays `sieve_list` and `magics` (line 389) and `forb_ba` and `forbidden` (line 391) as lists, for the arrays with their counts, the bit `RATPOINTS_CHECK_DENOM` of the flags as a Boolean and the fields of the result (`Ratpoints.Params`), and into the entries, the arrays `sieve` as the store of the tables of rows; no field for the array `stage3_list` (line 389), the buffer of the table of the third stage that the sieving thread fills for each denominator (`find_points.c:5127`), which the model replaces by the thunk of the checks (`Ratpoints.Sieve.fillChecks`, `Ratpoints.Sieve.sift`).)

- ratpoints.h:392-394: excluded (memory): the private fields of the arguments that hold memory: the buffer of the words of the forbidden divisors beyond the table with its length (forb_words, forb_words_len), the buffer of the rows as allocated, with the primes and the bit arrays it has room for (ba_buffer_na, ba_buffer_primes, ba_buffer_arrays), and the buffer of the prime powers (pw_buffer); the model makes the words, the rows and the records of the prime powers as fresh values (Ratpoints.Sieve.forbiddenRow, Ratpoints.Sieve.examinePower)

- ratpoints.h:395-397: modelled by `Ratpoints.Params` (deviation of `Ratpoints.Params`: the parameters as a structure of their own, for the fields of the arguments that `find_points_work` normalizes and `sieving_info` sets (`sp1`, `sp2`, `sp3`, `max_forbidden`, `array_size`, `run_words`, `run_denoms`, `adapt_at`, `check_rel`; `sp3_max` as the length of `Ratpoints.SievingInfo.entries`) and the local `pn_lim` of `sieving_info` at the end (`lookedAt`); the Jacobi switch a Boolean (`Ratpoints.Params.jacobi`), true when the bit `RATPOINTS_NO_JACOBI` of the flags is clear, for that bit of `args->flags`, which `main.c` clears with the others; a natural number for the `long` field `array_size` (`Ratpoints.Params.arraySize`), `0` for the default, which the library takes for every value `≤ 0` (`Ratpoints.Params.arraySizeAt`); `Float`s for the doubles and a `UInt64` for the `unsigned long`.)

- ratpoints.h:398-398: excluded (threads): the pool of threads of the arguments; the model has no threads

- ratpoints.h:399-400: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- ratpoints.h:403-405: modelled by Ratpoints.Options (deviation of Ratpoints.Options: a structure for the fields of ratpoints_args that hold these options: Booleans for the bits RATPOINTS_NO_CHECK, RATPOINTS_NO_Y and RATPOINTS_NO_REVERSE of the flags, noCheck and noY true when their bits are set, reverse true when RATPOINTS_NO_REVERSE is clear, so that all three defaults are those of the cleared flags (main.c, line 352); of the other bits of the flags, which main.c clears with these, the Jacobi switch is that of the options of the sieve (Ratpoints.SieveOptions.jacobi), and RATPOINTS_VERBOSE is the print channel of the search (Ratpoints.findPointsVerbose), which the program makes from Rat-

points.Cli.Output.verbose; an integer for args->sturm; a list of pairs of doubles (Ratpoints.Double) for the array domain of the type ratpoints_interval with the number num_inter of its intervals, the empty list for num_inter = 0, so that no number of intervals is negative and the room in the array for the intervals of the Sturm step is no concern of the caller; integers for the longs b_low and b_high; of the fields that share their lines in ratpoints.h, sp1, sp2, num_primes and max_forbidden are read by the options of the sieve (Ratpoints.SieveOptions), and sp3 is the end of the third stage of the choice (Ratpoints.Params.sp3), corrected during the run (Ratpoints.Sieve.Correction).)

- ratpoints.h:406-406: modelled by Ratpoints.Params (deviation of Ratpoints.Params: the parameters as a structure of their own, for the fields of the arguments that find_points_work normalizes and sieving_info sets (sp1, sp2, sp3, max_forbidden, array_size, run_words, run_denoms, adapt_at, check_rel; sp3_max as the length of Ratpoints.SievingInfo.entries) and the local pn_lim of sieving_info at the end (lookedAt); the Jacobi switch a Boolean (Ratpoints.Params.jacobi), true when the bit RATPOINTS_NO_JACOBI of the flags is clear, for that bit of args->flags, which main.c clears with the others; a natural number for the long field array_size (Ratpoints.Params.arraySize), 0 for the default, which the library takes for every value ≤ 0 (Ratpoints.Params.arraySizeAt); Floats for the doubles and a UInt64 for the unsigned long.)

- ratpoints.h:407-407: modelled by Ratpoints.Cli.Output (deviation of Ratpoints.Cli.Output: a structure for the global variables; Option String for the strings and the format, none for the null pointer, and Lean strings for the C strings of the command line, which the Lean runtime decodes as UTF-8 (bytes that do not form UTF-8 are read as U+FFFD, one for each malformed sequence as the Lean runtime cuts them, where main.c copies the bytes); the bit RATPOINTS_VERBOSE of args->flags a field here, from which the program makes the print channel of the library (Ratpoints.Cli.run).)

- ratpoints.h:409-411: modelled by Ratpoints.findPointsVerbose (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1),

or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- ratpoints.h:414-414: modelled by `Ratpoints.CheckFlags`, `Ratpoints.Outcome` (deviation of `Ratpoints.CheckFlags`: a structure with a Boolean for each of the three bits, for the bits of the word `flags` of the arguments, which `_ratpoints_check_point` reads through its worker and `find_points_work_1` directly; the bits `RATPOINTS_NO_CHECK` and `RATPOINTS_NO_Y` copied from the options of the search (`Ratpoints.Options`), and `RATPOINTS_REVERSED` from the reversal check (`Ratpoints.Setup.reversal`).) (deviation of `Ratpoints.Outcome`: a structure for what the C code returns and writes back into the arguments: the data of the callback, which the caller has behind its pointer `info`, the count, which `find_points` returns as a `long` that also carries its error codes (`Ratpoints.findPointsWith` returns the refusals apart), Booleans for three bits of `args->flags`, and the search intervals as a list, for the array `args->domain` and the number `args->num_inter` that the search writes back and the program reads after it; nothing else written back: neither the reversed coefficients and the degree (`args->cof`, `args->degree`), which nothing reads after the search; nor the other internal bits of the flags, `RATPOINTS_CHECK_DENOM`, `RATPOINTS_USE_JACOBI` and `RATPOINTS_NO_REVERSE_AUTO`, which the program does not read either, the bits of the input coming back as the caller set them (`find_points.c:4606`), which the program takes from its own options and settings (`Ratpoints.Options`, `Ratpoints.SieveOptions.jacobi`, `Ratpoints.Cli.Output.verbose`); nor `args->sp3`, which `find_points_work` zeroes on entry (line 4551) and leaves as the search leaves it, unlike `sp2`, which it restores (line 4561): the end of the third stage of the last block sifted (line 5163), that of the choice after the returns between the choice and the sieve, where `sp3_used` stays `0`, and `0` after the returns before the choice and after the exits of the choice when the curve has no points modulo a prime (`find_points.c:2614`, `find_points.c:2998`), which come before line 3017 sets it; nor the private fields that `sieving_info` and `adapt_primes` write, `adapt_at`, `run_words`, `run_denoms`, `sp3_max` and `check_rel`, which nothing reads before `sieving_info` sets them again in the next search; natural numbers for the `long`s `sp1_used`, `sp2_used` and `sp3_used`.)

- ratpoints.h:415-415: modelled by `Ratpoints.searchPoints` (deviation of `Ratpoints.searchPoints`: the range of the denominators, the search intervals and the flag `sturm_empty` as arguments, for the fields `b_low`, `b_high`, `domain` and `num_inter` of the arguments and the local flag of `find_points_work_1`; the flags of the denominators as Booleans, for the bits of `args->flags`; the divisors of the leading coefficient with their records as lists, for the arrays `args->den_info` and `args->divisors` of the arguments; the square test `lcfsq` of line 4906

made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local flag quit of find_points_work_1 and the count total, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), for the bits that find_points_work_1 sets in args->flags, the intervals that it leaves in args->domain and the fields sp1_used, sp2_used and sp3_used that the part after the choice sets (Ratpoints.chooseAndSearch); the report under RATPOINTS_VERBOSE printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under DEBUG.)

- ratpoints.h:416-417: modelled by Ratpoints.Outcome (deviation of Ratpoints.Outcome: a structure for what the C code returns and writes back into the arguments: the data of the callback, which the caller has behind its pointer info, the count, which find_points returns as a long that also carries its error codes (Ratpoints.findPointsWith returns the refusals apart), Booleans for three bits of args->flags, and the search intervals as a list, for the array args->domain and the number args->num_inter that the search writes back and the program reads after it; nothing else written back: neither the reversed coefficients and the degree (args->cof, args->degree), which nothing reads after the search; nor the other internal bits of the flags, RATPOINTS_CHECK_DENOM, RATPOINTS_USE_JACOBI and RATPOINTS_NO_REVERSE_AUTO, which the program does not read either, the bits of the input coming back as the caller set them (find_points.c:4606), which the program takes from its own options and settings (Ratpoints.Options, Ratpoints.SieveOptions.jacobi, Ratpoints.Cli.Output.verbose); nor args->sp3, which find_points_work zeroes on entry (line 4551) and leaves as the search leaves it, unlike sp2, which it restores (line 4561): the end of the third stage of the last block sifted (line 5163), that of the choice after the returns between the choice and the sieve, where sp3_used stays 0, and 0 after the returns before the choice and after the exits of the choice when the curve has no points modulo a prime (find_points.c:2614, find_points.c:2998), which come before line 3017 sets it; nor the private fields that sieving_info and adapt_primes write, adapt_at, run_words, run_denoms, sp3_max and check_rel, which nothing reads before sieving_info sets them again in the next search; natural numbers for the longs sp1_used, sp2_used and sp3_used.)

- ratpoints.h:418-418: modelled by Ratpoints.searchPoints (deviation of Ratpoints.searchPoints: the range of the denominators, the search intervals and the flag sturm_empty as arguments, for the fields b_low, b_high, domain and num_inter of the arguments and the local flag of find_points_work_1; the flags of the denominators as Booleans, for the bits of args->flags; the divisors of the leading coefficient with their records as lists, for the arrays args->den_info and args->divisors of the arguments; the square test lcfsq of line 4906 made on the polynomial searched (Ratpoints.Setup.isSquare), where the C code computes it before the reversal check (line 4634) and clears it in case 2 (line 4770): the same value for an even degree, the only one that reads it; the array size normalized for the height bound where the estimate of the run reads it (Ratpoints.arraySizeAt); the test on point_at_infty of the 2-adic exit recomputed, as Ratpoints.Sieve.pointsAtInfinity decides it; the 2-adic exit returns the delivery with the message, nothing delivered, where find_points_work_1 returns the count 0; the delivery passed, for the pointer info, the local

flag `quit` of `find_points_work_1` and the count `total`, and the result with the bits of the flags written back, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), for the bits that `find_points_work_1` sets in `args->flags`, the intervals that it leaves in `args->domain` and the fields `sp1_used`, `sp2_used` and `sp3_used` that the part after the choice sets (`Ratpoints.chooseAndSearch`); the report under `RATPOINTS_VERBOSE` printed through the print channel, the lines of the tests on the denominators together after the divisor set-up, which prints nothing; no output under `DEBUG`.)

- `ratpoints.h:421-424`: modelled by `Ratpoints.Options.normalize` (deviation of `Ratpoints.Options.normalize`: the normalized options returned, for the fields of the arguments that the C code normalizes in place and restores in `find_points_work` (lines 4542-4545 and 4560-4563), which the caller does not see either; no clamp of the number of the search intervals, which a list cannot make negative; the bit `RATPOINTS_NO_REVERSE_AUTO`, which the C code sets in the flags and the reversal check tests with `RATPOINTS_NO_REVERSE` (line 4739), as the switch `reverse` cleared; integers for the `long`s.)

- `ratpoints.h:427-430`: modelled by `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args-`

`>flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- `ratpoints.h:436-437`: modelled by `Ratpoints.Answer`, `Ratpoints.findPointsVerbose` (deviation of `Ratpoints.Answer`: the result of the call returned as a value, for the `int` that `process` returns, the `int` that it writes through the pointer `quit`, which the library clears before the search and tests after each call, so that it is clear at every call and the callback takes no argument for it, and the data that it may change through the pointer `info`; an integer for the `int`, with no bound, so that the count of a search has no range obligation (`Ratpoints.Delivery`); a Boolean for the `int *quit`.) (deviation of `Ratpoints.findPointsVerbose`: one function for `find_points`, `find_points_work` and the start of `find_points_work_1`, which the library calls in turn, and the rest of which is `Ratpoints.findPointsWork`. The refusals of the program, in its order and with its errors (`Ratpoints.Refusal`), returned apart from the outcome with the data of the callback, for the negative codes of the library (`RATPOINTS_NON_SQUAREFREE` and `RATPOINTS_BAD_ARGS`; `RATPOINTS_WORK_LENGTH_TOO_SMALL` and `RATPOINTS_NO_MEMORY` cannot arise), which share the `long` that `find_points` returns with the count, the caller keeping its data behind the pointer `info`: fewer than two coefficients are refused on the length of `c`, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that `5 0` with `H = 0` is error 4, as the program gives it. The degree with the leading zeros dropped is `Ratpoints.BruteForce.degree c`. A natural number `H` for the `long height` of the arguments (`ratpoints.h:362`), so that `H = 0` stands for the heights below 1 that line 4658 refuses; a height that is not a `long`, `2 ^ 63` or more, is refused with error 4, as the program refuses it when it reads the height (`Ratpoints.Cli.heightBound`), where the library cannot be given one, its height being a `long`. The end of the type `ratpoints_args` (`ratpoints.h:399-400`) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (`Ratpoints.Options`, `Ratpoints.SieveOptions`, `Ratpoints.Params`, `Ratpoints.SievingInfo`, `Ratpoints.Outcome`), of the divisor set-up (`Ratpoints.Setup.setupUs1`), or excluded. A list for the array of GMP integers and its scratch integers `work`, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (`Ratpoints.Outcome`), which are `0` unless the search sets them, as `find_points_work` zeroes them on entry with the working field `sp3`, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (`Ratpoints.Answer`), and its data a value, for the pointer `info`; the bit `RATPOINTS_VERBOSE` of the flags as the print channel `print`, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (`Ratpoints.Options`), copied into the flags of the check (`Ratpoints.CheckFlags`) with the flag of the reversal, for the bits of `args->flags` that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (`Ratpoints.Sieve.pointsAtInfinity`, `Ratpoints.Setup.reversal`, `Ratpoints.sturmRegion`, `Ratpoints.searchPoints`); no failure for lack of memory, which the model cannot meet.)

- ratpoints.h:439-439: excluded (interface): the prototype of find_points_init, which a caller of the library calls before find_points_work and whose definition (find_points.c:272-375) is excluded as interface: the model allocates nothing

- ratpoints.h:441-442: modelled by Ratpoints.Answer, Ratpoints.findPointsVerbose (deviation of Ratpoints.Answer: the result of the call returned as a value, for the int that process returns, the int that it writes through the pointer quit, which the library clears before the search and tests after each call, so that it is clear at every call and the callback takes no argument for it, and the data that it may change through the pointer info; an integer for the int, with no bound, so that the count of a search has no range obligation (Ratpoints.Delivery); a Boolean for the int *quit.) (deviation of Ratpoints.findPointsVerbose: one function for find_points, find_points_work and the start of find_points_work_1, which the library calls in turn, and the rest of which is Ratpoints.findPointsWork. The refusals of the program, in its order and with its errors (Ratpoints.Refusal), returned apart from the outcome with the data of the callback, for the negative codes of the library (RATPOINTS_NON_SQUAREFREE and RATPOINTS_BAD_ARGS; RATPOINTS_WORK_LENGTH_TOO_SMALL and RATPOINTS_NO_MEMORY cannot arise), which share the long that find_points returns with the count, the caller keeping its data behind the pointer info: fewer than two coefficients are refused on the length of c, as the program refuses them, where the library refuses an empty list as a negative degree and one coefficient as the degree 0 (a longer list of degree 0 fails the test of the leading zeros first); and the height bound, which the program checks before it calls the library, comes before the test of the leading zeros, which the library makes first, so that 5 0 with H = 0 is error 4, as the program gives it. The degree with the leading zeros dropped is Ratpoints.BruteForce.degree c. A natural number H for the long height of the arguments (ratpoints.h:362), so that H = 0 stands for the heights below 1 that line 4658 refuses; a height that is not a long, 2 ^ 63 or more, is refused with error 4, as the program refuses it when it reads the height (Ratpoints.Cli.heightBound), where the library cannot be given one, its height being a long. The end of the type ratpoints_args (ratpoints.h:399-400) with its first fields, whose other fields are those of the options, the parameters, the result of the choice of the moduli and the outcome (Ratpoints.Options, Ratpoints.SieveOptions, Ratpoints.Params, Ratpoints.SievingInfo, Ratpoints.Outcome), of the divisor set-up (Ratpoints.Setup.setupUs1), or excluded. A list for the array of GMP integers and its scratch integers work, with nothing written back to the caller but the data of the callback, the count, three bits of the flags, the search intervals and the numbers of the moduli used (Ratpoints.Outcome), which are 0 unless the search sets them, as find_points_work zeroes them on entry with the working field sp3, so that there is no state between calls and the reset of the flags clears nothing; the callback a function that returns what the call gives (Ratpoints.Answer), and its data a value, for the pointer info; the bit RATPOINTS_VERBOSE of the flags as the print channel print, a function that prints a text into the data of the callback, which holds the standard output of the caller, so that the text comes between the points as on the standard output of the C program, and which prints nothing when the bit is clear; the options of the output as fields of the options (Ratpoints.Options), copied into the flags of the check (Ratpoints.CheckFlags) with the flag of the reversal, for the bits of args->flags that the mask of line 4606 keeps; the points at infinity and the square test of the leading coefficient are decided where they are used (Ratpoints.Sieve.pointsAtInfinity, Ratpoints.Setup.reversal, Ratpoints.sturmRegion, Ratpoints.searchPoints); no failure for lack of memory, which the model cannot meet.)

- ratpoints.h:444-444: excluded (memory): the prototype of find_points_clear, which frees

what find_points_init allocated and whose definition (find_points.c:377-416) is excluded as memory: the model allocates nothing

primes.h

- primes.h:29-40: modelled by `Ratpoints.primeTable`

gen_find_points_h.c

- gen_find_points_h.c:30-31: excluded (other): the includes of the generator of find_points.h
- gen_find_points_h.c:33-44: modelled by `Ratpoints.Sieve.inversesMod` (deviation of `Ratpoints.Sieve.inversesMod`: a function of `p`, computed when it is called, for the row of `p` in the table `inverses` that the build generates and that `find_points.c` includes; the trial stops before `p` and gives `0` when it finds nothing, where `inv_mod_p` would not stop, which never happens for a prime `p` and a residue other than `0`.)
- gen_find_points_h.c:46-67: excluded (other): the head of main of the generator of find_points.h and the guard it writes into the header against a header generated for other values of RBA_PACK, RATPOINTS_CHUNK and RATPOINTS_NUM_PRIMES; the model computes its tables from its constants
- gen_find_points_h.c:69-88: modelled by `Ratpoints.Sieve.squaresMod` (deviation of `Ratpoints.Sieve.squaresMod`: a function of `p`, computed when it is called, for the row of `p` in the table `squares` that the build generates and that `find_points.c` includes.)
- gen_find_points_h.c:90-100: modelled by `Ratpoints.Sieve.inversesMod` (deviation of `Ratpoints.Sieve.inversesMod`: a function of `p`, computed when it is called, for the row of `p` in the table `inverses` that the build generates and that `find_points.c` includes; the trial stops before `p` and gives `0` when it finds nothing, where `inv_mod_p` would not stop, which never happens for a prime `p` and a residue other than `0`.)
- gen_find_points_h.c:102-116: modelled by `Ratpoints.Sieve.sieves0` (deviation of `Ratpoints.Sieve.sieves0`: a function of `p`, computed when it is called, for the row of `p` in the table `sieves0` that the build generates and that `find_points.c` includes; its first `(p + chunk - 1) rbaPack` words, for the row of `RBA_PACK (RATPOINTS_MAX_PRIME_EVEN + RATPOINTS_CHUNK - 1)` words whose other words are `0` and never read.)
- gen_find_points_h.c:118-120: modelled by `Ratpoints.Sieve.forbiddenRow` (deviation of `Ratpoints.Sieve.forbiddenRow`: one function for the two codes that make these words, the array `work` of the generator of `find_points.h`, for the primes of the table, and the words that `sieving_info` makes in the buffer `forb_words` for the forbidden divisors beyond the table (`Ratpoints.Sieve.forbidden`), with the positions `p i` for `i < 64`, for the positions `m = 0, p, 2 p, …` below `64 p` there; a fresh array of `p` words, for the array `work` of `RATPOINTS_MAX_PRIME` words and for the space in the buffer; divisions and remainders for the shifts and masks by `LONG_SHIFT` and `LONG_MASK`.)
- gen_find_points_h.c:122-134: modelled by `Ratpoints.Sieve.sieves0` (deviation of `Ratpoints.Sieve.sieves0`: a function of `p`, computed when it is called, for the row of `p` in the table `sieves0` that the build generates and that `find_points.c` includes; its first `(p + chunk`

- 1) `rbaPack` words, for the row of `RBA_PACK (RATPOINTS_MAX_PRIME_EVEN + RATPOINTS_CHUNK - 1)` words whose other words are `0` and never read.)

- gen_find_points_h.c:136-137: excluded (other): the end of main of the generator of find_points.h

gen_init_sieve_h.c

- gen_init_sieve_h.c:29-46: excluded (other): the includes and the head of main of the generator of init_sieve.h, a global array work that it never uses, and the guard it writes into the header against a header generated for another RATPOINTS_NUM_PRIMES; the model makes the rows from its constants

- gen_init_sieve_h.c:48-62: modelled by `Ratpoints.Sieve.sieveInit` (deviation of `Ratpoints.Sieve.sieveInit`: a function of `p`, for the entry of `p` in the table of the pointers to the functions that the build generates, one for each prime, and that `init.c` includes; the kind of the function, for the pointer to it.)

- gen_init_sieve_h.c:64-65: excluded (other): the end of main of the generator of init_sieve.h

main.c

- main.c:29-37: excluded (other): the includes of the program; the model has no headers

- main.c:43-48: modelled by `Ratpoints.Cli.banner` (deviation of `Ratpoints.Cli.banner`: the text is that of `ratpoints-lean`.)

- main.c:54-54: modelled by `Ratpoints.Cli.coefficients` (deviation of `Ratpoints.Cli.coefficients`: a list of `Int`s for the global array `c` of `RATPOINTS_MAX_DEGREE + 1` GMP integers; the integers beyond these, which `main.c` reads into the variable `fff` only to check them, are read like the others.)

- main.c:55-55: excluded (memory): the GMP integers tmp and tmp2 for intermediate results, which the program initializes and releases and the model does not need

- main.c:57-64: modelled by `Ratpoints.Cli.Output` (deviation of `Ratpoints.Cli.Output`: a structure for the global variables; `Option String` for the strings and the format, `none` for the null pointer, and Lean strings for the C strings of the command line, which the Lean runtime decodes as UTF-8 (bytes that do not form UTF-8 are read as U+FFFD, one for each malformed sequence as the Lean runtime cuts them, where `main.c` copies the bytes); the bit `RATPOINTS_VERBOSE` of `args->flags` a field here, from which the program makes the print channel of the library (`Ratpoints.Cli.run`).)

- main.c:66-66: modelled by `Ratpoints.Cli.Intervals` (deviation of `Ratpoints.Cli.Intervals`: a structure for the two locals of `read_input` and the two global arrays, whose entries are zero (the doubles) and `0` (the exact ends, as `mpq_init` sets them, `main.c:205`) at the start; pairs of doubles for the entries of the type `ratpoints_interval`; a natural number for the `long num_inter`.)

- main.c:71-71: modelled by `Ratpoints.Cli.ExactEnd` (deviation of `Ratpoints.Cli.ExactEnd`: an `Int` for the `int inf`, and a rational number of core for the `mpq_t q`, `0` by default as `mpq_init` sets it.)

- main.c:73-73: modelled by Ratpoints.Cli.Intervals (deviation of Ratpoints.Cli.Intervals: a structure for the two locals of read_input and the two global arrays, whose entries are zero (the doubles) and 0 (the exact ends, as mpq_init sets them, main.c:205) at the start; pairs of doubles for the entries of the type ratpoints_interval; a natural number for the long num_inter.)
- main.c:77-88: modelled by Ratpoints.Cli.usage (deviation of Ratpoints.Cli.usage: the program is called ratpoints-lean, and the lines below the first are indented to match.)
- main.c:94-103: excluded (other): the prototypes of the functions of the program; the model has no forward declarations
- main.c:110-113: modelled by Ratpoints.Cli.Data (deviation of Ratpoints.Cli.Data: the text printed so far, for the standard output, to which process prints; an array for the pointer ends and a natural number for the long num_inter.)
- main.c:116-120: modelled by Ratpoints.Cli.cmpEndX (deviation of Ratpoints.Cli.cmpEndX: the sign as -1, 0 or 1, for the value of mpq_cmp_si, of which main.c reads only the sign; integers for the longs.)
- main.c:126-137: modelled by Ratpoints.Cli.inRegion (deviation of Ratpoints.Cli.inRegion: natural numbers for the long indices, an array for the pointer e.)
- main.c:139-142: modelled by Ratpoints.Cli.process (deviation of Ratpoints.Cli.process: the data a value, for the pointer info, and returned with the weight and the flag, for the int *quit that the library passes, which is clear at every call; the text appended to that of the data, for the standard output, and no fflush.)
- main.c:150-156: modelled by Ratpoints.Cli.process (deviation of Ratpoints.Cli.process: the data a value, for the pointer info, and returned with the weight and the flag, for the int *quit that the library passes, which is clear at every call; the text appended to that of the data, for the standard output, and no fflush.)
- main.c:157-177: modelled by Ratpoints.Cli.formatPoint (deviation of Ratpoints.Cli.formatPoint: a recursion over the characters, for the loop over the C string; an Int for the GMP integer y and for the longs x and z, which print alike.)
- main.c:178-182: modelled by Ratpoints.Cli.escape (deviation of Ratpoints.Cli.escape: one function for the two switches of process and print_string, which agree.)
- main.c:183-186: modelled by Ratpoints.Cli.formatPoint (deviation of Ratpoints.Cli.formatPoint: a recursion over the characters, for the loop over the C string; an Int for the GMP integer y and for the longs x and z, which print alike.)
- main.c:187-189: modelled by Ratpoints.Cli.process (deviation of Ratpoints.Cli.process: the data a value, for the pointer info, and returned with the weight and the flag, for the int *quit that the library passes, which is clear at every call; the text appended to that of the data, for the standard output, and no fflush.)
- main.c:195-198: modelled by Ratpoints.Cli.run (deviation of Ratpoints.Cli.run: the text written once at the end (Ratpoints.Cli.main), where main.c prints as it goes; the bit RATPOINTS_VERBOSE of the flags, which the program sets under -v and leaves clear under -q (Ratpoints.Cli.input), as the print channel that it passes the library, and one that prints nothing otherwise.)

- main.c:199-199: excluded (interface): the declaration of the arguments of the library, ratpoints_args args, whose fields the model passes as the arguments of the search with a callback
- main.c:201-207: excluded (memory): the initialization of the GMP integers of the coefficients, of the rational ends of the intervals and of tmp and tmp2 at the start of main
- main.c:210-214: modelled by `Ratpoints.Cli.input` (deviation of `Ratpoints.Cli.input`: `read_input` never returns a value other than 0, so the usage that `main.c` prints then (lines 212 and 213) has no counterpart; the largest denominator `-1` of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer `args->domain` to the array `domain` and the number `args->num_inter`.)
- main.c:215-220: modelled by `Ratpoints.Cli.preamble`
- main.c:221-222: modelled by `Ratpoints.Cli.run` (deviation of `Ratpoints.Cli.run`: the text written once at the end (`Ratpoints.Cli.main`), where `main.c` prints as it goes; the bit `RATPOINTS_VERBOSE` of the flags, which the program sets under `-v` and leaves clear under `-q` (`Ratpoints.Cli.input`), as the print channel that it passes the library, and one that prints nothing otherwise.)
- main.c:224-224: modelled by `Ratpoints.Cli.preamble`
- main.c:232-234: excluded (memory): the allocation of the data that the program passes to the callback process, and error 10 when it fails
- main.c:235-242: modelled by `Ratpoints.Cli.Data.ofOutput` (deviation of `Ratpoints.Cli.Data.ofOutput`: the format the one that `Ratpoints.Cli.input` settled, `""` if none.)
- main.c:246-247: modelled by `Ratpoints.Cli.Data.ofOutput` (deviation of `Ratpoints.Cli.Data.ofOutput`: the format the one that `Ratpoints.Cli.input` settled, `""` if none.)
- main.c:248-248: modelled by `Ratpoints.Cli.run` (deviation of `Ratpoints.Cli.run`: the text written once at the end (`Ratpoints.Cli.main`), where `main.c` prints as it goes; the bit `RATPOINTS_VERBOSE` of the flags, which the program sets under `-v` and leaves clear under `-q` (`Ratpoints.Cli.input`), as the print channel that it passes the library, and one that prints nothing otherwise.)
- main.c:251-252: modelled by `Ratpoints.Cli.run` (deviation of `Ratpoints.Cli.run`: the text written once at the end (`Ratpoints.Cli.main`), where `main.c` prints as it goes; the bit `RATPOINTS_VERBOSE` of the flags, which the program sets under `-v` and leaves clear under `-q` (`Ratpoints.Cli.input`), as the print channel that it passes the library, and one that prints nothing otherwise.)
- main.c:254-255: excluded (memory): the release of the data that the program passed to the callback process, at the end of its block
- main.c:256-261: modelled by `Ratpoints.Cli.Failure.ofRefusal` (deviation of `Ratpoints.Cli.Failure.ofRefusal`: no error 10, since the model has no failure for lack of memory; the refusals of the coefficients and of the height (`Ratpoints.Refusal.tooManyCoefficients`, `Ratpoints.Refusal.tooFewCoefficients`, `Ratpoints.Refusal.incorrectHeight`) mapped to the errors 3, 5 and 4 with which `read_input` refuses those inputs before the library is called, where line 260 gives error 9 for the code `RATPOINTS_BAD_ARGS` with which the library refuses those it can be given (lines 4557-4558, 4633 and 4658 of `find_points.c`); no command line reaches them; no `mpz_clear`.)

- main.c:265-265: modelled by Ratpoints.Cli.run (deviation of Ratpoints.Cli.run: the text written once at the end (Ratpoints.Cli.main), where main.c prints as it goes; the bit RATPOINTS_VERBOSE of the flags, which the program sets under -v and leaves clear under -q (Ratpoints.Cli.input), as the print channel that it passes the library, and one that prints nothing otherwise.)
- main.c:267-267: modelled by Ratpoints.Cli.run (deviation of Ratpoints.Cli.run: the text written once at the end (Ratpoints.Cli.main), where main.c prints as it goes; the bit RATPOINTS_VERBOSE of the flags, which the program sets under -v and leaves clear under -q (Ratpoints.Cli.input), as the print channel that it passes the library, and one that prints nothing otherwise.)
- main.c:268-276: modelled by Ratpoints.Cli.report
- main.c:279-283: excluded (memory): the release of tmp and tmp2, of the GMP integers of the coefficients and of the rational ends of the intervals at the end of main
- main.c:284-285: modelled by Ratpoints.Cli.run (deviation of Ratpoints.Cli.run: the text written once at the end (Ratpoints.Cli.main), where main.c prints as it goes; the bit RATPOINTS_VERBOSE of the flags, which the program sets under -v and leaves clear under -q (Ratpoints.Cli.input), as the print channel that it passes the library, and one that prints nothing otherwise.)
- main.c:299-305: modelled by Ratpoints.Cli.heightEnd (deviation of Ratpoints.Cli.heightEnd: a natural number for the long height, at most 2 ^ 63 - 1.)
- main.c:309-313: modelled by Ratpoints.Cli.cmpEnd (deviation of Ratpoints.Cli.cmpEnd: the sign as -1, 0 or 1, for the value of mpq_cmp, of which main.c reads only the sign.)
- main.c:315-316: modelled by Ratpoints.Cli.input (deviation of Ratpoints.Cli.input: read_input never returns a value other than 0, so the usage that main.c prints then (lines 212 and 213) has no counterpart; the largest denominator -1 of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer args->domain to the array domain and the number args->num_inter.)
- main.c:317-317: modelled by Ratpoints.Cli.Intervals (deviation of Ratpoints.Cli.Intervals: a structure for the two locals of read_input and the two global arrays, whose entries are zero (the doubles) and 0 (the exact ends, as mpq_init sets them, main.c:205) at the start; pairs of doubles for the entries of the type ratpoints_interval; a natural number for the long num_inter.)
- main.c:318-318: modelled by Ratpoints.Cli.coefficients (deviation of Ratpoints.Cli.coefficients: a list of Ints for the global array c of RATPOINTS_MAX_DEGREE + 1 GMP integers; the integers beyond these, which main.c reads into the variable fff only to check them, are read like the others.)
- main.c:319-319: modelled by Ratpoints.Cli.Intervals (deviation of Ratpoints.Cli.Intervals: a structure for the two locals of read_input and the two global arrays, whose entries are zero (the doubles) and 0 (the exact ends, as mpq_init sets them, main.c:205) at the start; pairs of doubles for the entries of the type ratpoints_interval; a natural number for the long num_inter.)

- main.c:330-332: excluded (interface): the initialization of the fields of the arguments of the library that the model has no counterpart for: the pointer to the global array c of the coefficients, whose entries read_input fills later (line 368), and the degree and the height, which it sets later (lines 374 and 378) and the model passes as values

- main.c:333-336: modelled by `Ratpoints.Cli.input` (deviation of `Ratpoints.Cli.input`: `read_input` never returns a value other than 0, so the usage that `main.c` prints then (lines 212 and 213) has no counterpart; the largest denominator `-1` of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer `args->domain` to the array `domain` and the number `args->num_inter`.)

- main.c:337-347: modelled by `Ratpoints.SieveOptions` (deviation of `Ratpoints.SieveOptions`: integers for the `long`s of the fields, doubles by their bits (`Ratpoints.Double`) for the doubles, as the program reads doubles (`Ratpoints.Cli.scanDouble?`), and a Boolean for the bit `RATPOINTS_NO_JACOBI` of the flags, which `main.c` clears with the others; the bit of the reversal is in `Ratpoints.Options`.)

- main.c:348-348: modelled by `Ratpoints.Options` (deviation of `Ratpoints.Options`: a structure for the fields of `ratpoints_args` that hold these options: Booleans for the bits `RATPOINTS_NO_CHECK`, `RATPOINTS_NO_Y` and `RATPOINTS_NO_REVERSE` of the flags, `noCheck` and `noY` true when their bits are set, `reverse` true when `RATPOINTS_NO_REVERSE` is clear, so that all three defaults are those of the cleared flags (`main.c`, line 352); of the other bits of the flags, which `main.c` clears with these, the Jacobi switch is that of the options of the sieve (`Ratpoints.SieveOptions.jacobi`), and `RATPOINTS_VERBOSE` is the print channel of the search (`Ratpoints.findPointsVerbose`), which the program makes from `Ratpoints.Cli.Output.verbose`; an integer for `args->sturm`; a list of pairs of doubles (`Ratpoints.Double`) for the array `domain` of the type `ratpoints_interval` with the number `num_inter` of its intervals, the empty list for `num_inter = 0`, so that no number of intervals is negative and the room in the array for the intervals of the Sturm step is no concern of the caller; integers for the `long`s `b_low` and `b_high`; of the fields that share their lines in `ratpoints.h`, `sp1`, `sp2`, `num_primes` and `max_forbidden` are read by the options of the sieve (`Ratpoints.SieveOptions`), and `sp3` is the end of the third stage of the choice (`Ratpoints.Params.sp3`), corrected during the run (`Ratpoints.Sieve.Correction`).)

- main.c:349-350: modelled by `Ratpoints.SieveOptions` (deviation of `Ratpoints.SieveOptions`: integers for the `long`s of the fields, doubles by their bits (`Ratpoints.Double`) for the doubles, as the program reads doubles (`Ratpoints.Cli.scanDouble?`), and a Boolean for the bit `RATPOINTS_NO_JACOBI` of the flags, which `main.c` clears with the others; the bit of the reversal is in `Ratpoints.Options`.)

- main.c:351-351: excluded (threads): the default of the number of threads, one: the calling thread sieves; the model has no threads

- main.c:352-352: modelled by `Ratpoints.Options`, `Ratpoints.SieveOptions` (deviation of `Ratpoints.Options`: a structure for the fields of `ratpoints_args` that hold these options: Booleans for the bits `RATPOINTS_NO_CHECK`, `RATPOINTS_NO_Y` and `RATPOINTS_NO_REVERSE` of the flags, `noCheck` and `noY` true when their bits are set, `reverse` true when `RATPOINTS_NO_REVERSE` is clear, so that all three defaults are those of the cleared flags (`main.c`, line 352); of the other bits of the flags, which `main.c` clears with these, the Jacobi switch is that of the options of the sieve (`Ratpoints.SieveOptions.jacobi`), and `RATPOINTS_VERBOSE` is the print channel of the search (`Ratpoints.findPointsVerbose`), which the program makes

from `Ratpoints.Cli.Output.verbose`; an integer for `args->sturm`; a list of pairs of doubles (`Ratpoints.Double`) for the array `domain` of the type `ratpoints_interval` with the number `num_inter` of its intervals, the empty list for `num_inter = 0`, so that no number of intervals is negative and the room in the array for the intervals of the Sturm step is no concern of the caller; integers for the `long`s `b_low` and `b_high`; of the fields that share their lines in `ratpoints.h`, `sp1`, `sp2`, `num_primes` and `max_forbidden` are read by the options of the sieve (`Ratpoints.SieveOptions`), and `sp3` is the end of the third stage of the choice (`Ratpoints.Params.sp3`), corrected during the run (`Ratpoints.Sieve.Correction`).) (deviation of `Ratpoints.SieveOptions`: integers for the `long`s of the fields, doubles by their bits (`Ratpoints.Double`) for the doubles, as the program reads doubles (`Ratpoints.Cli.scanDouble?`), and a Boolean for the bit `RATPOINTS_NO_JACOBI` of the flags, which `main.c` clears with the others; the bit of the reversal is in `Ratpoints.Options`.)

- `main.c:357-357`: excluded (memory): the initialization of the GMP integer fff, into which the program reads the coefficients beyond the bound to check them
- `main.c:362-375`: modelled by `Ratpoints.Cli.coefficients` (deviation of `Ratpoints.Cli.coefficients`: a list of `Int`s for the global array `c` of `RATPOINTS_MAX_DEGREE + 1` GMP integers; the integers beyond these, which `main.c` reads into the variable `fff` only to check them, are read like the others.)
- `main.c:378-379`: modelled by `Ratpoints.Cli.heightBound`
- `main.c:380-380`: modelled by `Ratpoints.Cli.input` (deviation of `Ratpoints.Cli.input`: `read_input` never returns a value other than 0, so the usage that `main.c` prints then (lines 212 and 213) has no counterpart; the largest denominator `-1` of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer `args->domain` to the array `domain` and the number `args->num_inter`.)
- `main.c:383-390`: modelled by `Ratpoints.Cli.Output` (deviation of `Ratpoints.Cli.Output`: a structure for the global variables; `Option String` for the strings and the format, `none` for the null pointer, and Lean strings for the C strings of the command line, which the Lean runtime decodes as UTF-8 (bytes that do not form UTF-8 are read as U+FFFD, one for each malformed sequence as the Lean runtime cuts them, where `main.c` copies the bytes); the bit `RATPOINTS_VERBOSE` of `args->flags` a field here, from which the program makes the print channel of the library (`Ratpoints.Cli.run`).)
- `main.c:393-400`: modelled by `Ratpoints.Cli.checkOptions` (deviation of `Ratpoints.Cli.checkOptions`: the argument of `-t`, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of `-f`, `-fs`, `-fm` and `-fe` kept as values, for the copies that `main.c` allocates, with error 10 when the allocation fails, which the model cannot meet.)
- `main.c:401-413`: modelled by `Ratpoints.Cli.Intervals.lower`
- `main.c:414-416`: modelled by `Ratpoints.Cli.checkOptions` (deviation of `Ratpoints.Cli.checkOptions`: the argument of `-t`, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of `-f`, `-fs`, `-fm` and `-fe` kept as values, for the copies that `main.c` allocates, with error 10 when the allocation fails, which the model cannot meet.)
- `main.c:417-436`: modelled by `Ratpoints.Cli.Intervals.upper`

- main.c:437-532: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:533-577: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:578-593: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:594-601: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:602-637: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:638-649: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:650-668: modelled by Ratpoints.Cli.checkOptions (deviation of Ratpoints.Cli.checkOptions: the argument of -t, the number of threads, checked and then ignored, since the model sieves the blocks of the run in turn on one thread, which gives the output of every number of threads; the strings of -f, -fs, -fm and -fe kept as values, for the copies that main.c allocates, with error 10 when the allocation fails, which the model cannot meet.)
- main.c:669-669: modelled by Ratpoints.Cli.Intervals.complete
- main.c:673-678: modelled by Ratpoints.Cli.Intervals.complete
- main.c:679-679: modelled by Ratpoints.Cli.input (deviation of Ratpoints.Cli.input: read_input never returns a value other than 0, so the usage that main.c prints then (lines 212 and 213) has no counterpart; the largest denominator -1 of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer args->domain to the array domain and the number args->num_inter.)
- main.c:681-685: modelled by Ratpoints.Cli.defaultFormat

- main.c:688-688: modelled by `Ratpoints.Cli.input` (deviation of `Ratpoints.Cli.input`: `read_input` never returns a value other than 0, so the usage that `main.c` prints then (lines 212 and 213) has no counterpart; the largest denominator `-1` of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer `args->domain` to the array `domain` and the number `args->num_inter`.)
- main.c:690-690: excluded (memory): the release of the GMP integer fff at the end of read_input
- main.c:691-692: modelled by `Ratpoints.Cli.input` (deviation of `Ratpoints.Cli.input`: `read_input` never returns a value other than 0, so the usage that `main.c` prints then (lines 212 and 213) has no counterpart; the largest denominator `-1` of line 336, which line 380 replaces by the height bound, not set; the search intervals a list of the intervals read, for the pointer `args->domain` to the array `domain` and the number `args->num_inter`.)
- main.c:700-717: modelled by `Ratpoints.Cli.intOfWord?`
- main.c:724-728: modelled by `Ratpoints.Cli.onlyWord?`
- main.c:733-740: modelled by `Ratpoints.Cli.longOfString?` (deviation of `Ratpoints.Cli.longOfString?`: an `Int` for a `long`.)
- main.c:746-752: modelled by `Ratpoints.Cli.scanDouble?` (deviation of `Ratpoints.Cli.scanDouble?`: the NaN refused by its syntax, which `Ratpoints.Cli.isDoubleWord` does not accept, for the test `isnan` of the double that `strtod` reads.)
- main.c:761-763: modelled by `Ratpoints.Cli.outward` (deviation of `Ratpoints.Cli.outward`: the exact value of `x` (`Ratpoints.Double.toRat`) compared with `v` as rational numbers of core, for the GMP rational `t`, `mpq_set_d` and `mpq_cmp`; no `mpq_init` and `mpq_clear`.)
- main.c:765-765: excluded (memory): the initialization of the GMP rational t of outward
- main.c:766-773: modelled by `Ratpoints.Cli.outward` (deviation of `Ratpoints.Cli.outward`: the exact value of `x` (`Ratpoints.Double.toRat`) compared with `v` as rational numbers of core, for the GMP rational `t`, `mpq_set_d` and `mpq_cmp`; no `mpq_init` and `mpq_clear`.)
- main.c:774-774: excluded (memory): the release of the GMP rational t of outward
- main.c:775-776: modelled by `Ratpoints.Cli.outward` (deviation of `Ratpoints.Cli.outward`: the exact value of `x` (`Ratpoints.Double.toRat`) compared with `v` as rational numbers of core, for the GMP rational `t`, `mpq_set_d` and `mpq_cmp`; no `mpq_init` and `mpq_clear`.)
- main.c:798-806: modelled by `Ratpoints.Cli.scanEnd` (deviation of `Ratpoints.Cli.scanEnd`: the end and the double returned, or none, for the `exact_end` and the double that `scan_end` writes through pointers with its return value, the end of an infinity with `q = 0`, which `scan_end` leaves as it is (it is not read); the parts of the word read by `Ratpoints.Cli.floatParts` from the word of `Ratpoints.Cli.onlyWord?`, which is `s` without its white space; rational numbers of core for the GMP integers and rationals `m`, `p` and `e->q`, without `mpz_init`, `mpz_clear`, `malloc` and `free` (and the error 10 when `malloc` fails, which the model cannot meet).)
- main.c:808-819: modelled by `Ratpoints.Cli.scanEnd` (deviation of `Ratpoints.Cli.scanEnd`: the end and the double returned, or none, for the `exact_end` and the double that `scan_end` writes through pointers with its return value, the end of an infinity with `q = 0`, which

scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init, mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)

- main.c:820-821: excluded (memory): the allocation of the string of the digits of scan_end, and error 10 when it fails, which the model cannot meet
- main.c:822-834: modelled by Ratpoints.Cli.mantissaDigits (deviation of Ratpoints.Cli.mantissaDigits: a recursion over the characters, for the loop over the C string and the array digits, whose length is the number n of the digits.)
- main.c:835-847: modelled by Ratpoints.Cli.scanEnd (deviation of Ratpoints.Cli.scanEnd: the end and the double returned, or none, for the exact_end and the double that scan_end writes through pointers with its return value, the end of an infinity with q = 0, which scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init, mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)
- main.c:848-848: excluded (memory): the initialization of the GMP integer m of scan_end
- main.c:849-849: modelled by Ratpoints.Cli.scanEnd (deviation of Ratpoints.Cli.scanEnd: the end and the double returned, or none, for the exact_end and the double that scan_end writes through pointers with its return value, the end of an infinity with q = 0, which scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init, mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)
- main.c:850-850: excluded (memory): the release of the string of the digits of scan_end
- main.c:851-865: modelled by Ratpoints.Cli.scanEnd (deviation of Ratpoints.Cli.scanEnd: the end and the double returned, or none, for the exact_end and the double that scan_end writes through pointers with its return value, the end of an infinity with q = 0, which scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init, mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)
- main.c:867-867: excluded (memory): the initialization of the GMP integer p of scan_end
- main.c:868-873: modelled by Ratpoints.Cli.scanEnd (deviation of Ratpoints.Cli.scanEnd: the end and the double returned, or none, for the exact_end and the double that scan_end writes through pointers with its return value, the end of an infinity with q = 0, which scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init,

mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)

- main.c:874-874: excluded (memory): the release of the GMP integer p of scan_end
- main.c:875-876: modelled by Ratpoints.Cli.scanEnd (deviation of Ratpoints.Cli.scanEnd: the end and the double returned, or none, for the exact_end and the double that scan_end writes through pointers with its return value, the end of an infinity with q = 0, which scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init, mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)
- main.c:877-877: excluded (memory): the release of the GMP integer m of scan_end
- main.c:878-880: modelled by Ratpoints.Cli.scanEnd (deviation of Ratpoints.Cli.scanEnd: the end and the double returned, or none, for the exact_end and the double that scan_end writes through pointers with its return value, the end of an infinity with q = 0, which scan_end leaves as it is (it is not read); the parts of the word read by Ratpoints.Cli.floatParts from the word of Ratpoints.Cli.onlyWord?, which is s without its white space; rational numbers of core for the GMP integers and rationals m, p and e->q, without mpz_init, mpz_clear, malloc and free (and the error 10 when malloc fails, which the model cannot meet).)
- main.c:891-905: modelled by Ratpoints.Cli.joinIntervals (deviation of Ratpoints.Cli.joinIntervals: a list for the array args->domain and the number args->num_inter of its intervals, written back in place by join_intervals, and natural numbers for the long indices.)
- main.c:913-949: modelled by Ratpoints.Cli.printPoly (deviation of Ratpoints.Cli.printPoly: a fold over the coefficients, for the loop; no fflush.)
- main.c:952-960: modelled by Ratpoints.Cli.printString (deviation of Ratpoints.Cli.printString: a recursion over the characters, for the loop over the C string.)
- main.c:961-965: modelled by Ratpoints.Cli.escape (deviation of Ratpoints.Cli.escape: one function for the two switches of process and print_string, which agree.)
- main.c:966-968: modelled by Ratpoints.Cli.printString (deviation of Ratpoints.Cli.printString: a recursion over the characters, for the loop over the C string.)
- main.c:972-976: modelled by Ratpoints.Cli.Message.text (deviation of Ratpoints.Cli.Message.text: the banner of ratpoints-lean (Ratpoints.Cli.banner); a constructor for each message that ratpoints-lean prints, with its data, for the number n and the arguments of the library: message 1, which the program prints when read_input fails, which it never does, is left out; the loops over the intervals of messages 3 and 8 as Ratpoints.Cli.showIntervals, the conversion %lf of message 8 being %f; the difference sp3_used - sp2_used of message 4 an integer, for the long; no fflush.)
- main.c:977-978: excluded (other): the text of message 1, the banner and the usage, which main prints only when read_input returns a value other than 0, which it never does: its only return is at line 691, and every other failure exits through error

- main.c:979-1033: modelled by `Ratpoints.Cli.Message.text` (deviation of `Ratpoints.Cli.Message.text`: the banner of `ratpoints-lean` (`Ratpoints.Cli.banner`); a constructor for each message that `ratpoints-lean` prints, with its data, for the number `n` and the arguments of the library: message 1, which the program prints when `read_input` fails, which it never does, is left out; the loops over the intervals of messages 3 and 8 as `Ratpoints.Cli.showIntervals`, the conversion `%lf` of message 8 being `%f`; the difference `sp3_used - sp2_used` of message 4 an integer, for the `long`; no `fflush`.)
- main.c:1036-1048: modelled by `Ratpoints.Cli.Failure.message` (deviation of `Ratpoints.Cli.Failure.message`: the banner and the usage are those of `ratpoints-lean`.)
- main.c:1049-1049: excluded (memory): the message of error 10, for lack of memory, which the model cannot meet
- main.c:1050-1053: modelled by `Ratpoints.Cli.Failure.message` (deviation of `Ratpoints.Cli.Failure.message`: the banner and the usage are those of `ratpoints-lean`.)
- main.c:1054-1056: modelled by `Ratpoints.Cli.Failure.status`